Abstract: This document describes the detailed SCAMPI Architecture. It complements deliverable D1.1 “High-level SCAMPI architecture and components”, which provided an overview of the SCAMPI architecture and deliverable D1.2 “SCAMPI architecture and component design” which provided a more detailed view and thorough treatment of the Monitoring Application Programming Interface (MAPI).

This deliverable focuses on providing a detailed software architecture for the SCAMPI monitoring system as described in section 5. Based on a MAPI daemon (mapid) and specialized hardware, the proposed architecture combines monitoring needs received from different clients and implements them in the most efficient way.

<table>
<thead>
<tr>
<th>Contractual Date of Delivery</th>
<th>October 31st 2003</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Date of Delivery</td>
<td>November 13th 2003</td>
</tr>
<tr>
<td>Deliverable Security Class</td>
<td>Public</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
- **Masaryk University** Principal Contractor, Czech Republic
- **Siemens** Principal Contractor, Germany
## Contents

1 Introduction ........................................................................................................ 9
   1.1 Motivation - What is the problem? ......................................................... 9
   1.1.1 The Wild Spread of Cyber-Attacks .................................................. 9
   1.1.2 Friendly Fire on the Internet ............................................................... 11
   1.1.3 The Need for Internet Traffic Monitoring ......................................... 12
   1.1.4 The Benefits of Internet Traffic Monitoring ..................................... 12
   1.2 The Challenges of Network Monitoring .................................................. 14
   1.2.1 A Babel of Monitoring Tools and Languages ..................................... 14
   1.2.2 The network speed challenge ............................................................. 15
   1.3 Roadmap ................................................................................................. 17

2 Network Monitoring Applications ................................................................. 19
   2.1 Packet Capture ....................................................................................... 20
      2.1.1 Libpcap Interface ............................................................................ 20
      2.1.2 Packet Capture Application ............................................................ 20
   2.2 Traffic Summary Reporting .................................................................... 20
      2.2.1 Accounting Application .................................................................. 20
      2.2.2 Host Tracking Application ............................................................... 22
      2.2.3 Flow Record Export Application ..................................................... 23
      2.2.4 Flow-based Reporting ..................................................................... 23
   2.3 Threshold Alerting ................................................................................... 24
   2.4 Quality of Service Monitoring ............................................................... 24
   2.5 Security Applications ............................................................................. 26
      2.5.1 Intrusion Detection Application ....................................................... 26
      2.5.2 Denial of Service (DoS) Attack Detection Application ..................... 26

3 Overall System Architecture .......................................................................... 29
   3.1 Hardware ............................................................................................... 29
   3.2 Software ................................................................................................. 29

4 Hardware Architecture ..................................................................................... 31
   4.1 SCAMPI monitoring adapter ................................................................. 31
      4.1.1 Introduction ..................................................................................... 31
      4.1.2 COMBO6 motherboard ................................................................ 31
      4.1.3 Interface card ................................................................................ 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.4</td>
<td>Firmware</td>
<td>33</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Driver</td>
<td>34</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Feasibility study</td>
<td>35</td>
</tr>
<tr>
<td>4.1.7</td>
<td>Combo6 adapter to Combo6 driver interface</td>
<td>36</td>
</tr>
<tr>
<td>4.1.8</td>
<td>Booting</td>
<td>37</td>
</tr>
<tr>
<td>4.1.9</td>
<td>CAM</td>
<td>37</td>
</tr>
<tr>
<td>4.1.10</td>
<td>I/O queues</td>
<td>37</td>
</tr>
<tr>
<td>4.1.11</td>
<td>comboctl utility</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>The DAG card</td>
<td>41</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Timestamp format</td>
<td>42</td>
</tr>
<tr>
<td>4.2.2</td>
<td>The DAG API</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Intelligent Routers</td>
<td>43</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Overview</td>
<td>43</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Traffic Filtering and Mirroring Overview</td>
<td>44</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Configuring Traffic Filtering and Mirroring</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>The IXP1200 network processor</td>
<td>47</td>
</tr>
<tr>
<td>4.4.1</td>
<td>APIs</td>
<td>48</td>
</tr>
<tr>
<td>4.5</td>
<td>Commodity Interfaces</td>
<td>49</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Commodity Interfaces in Promiscuous Mode</td>
<td>49</td>
</tr>
<tr>
<td>4.5.2</td>
<td>The commodity adapter API</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Software Structure</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>The MAPI daemon</td>
<td>51</td>
</tr>
<tr>
<td>5.1.1</td>
<td>mapidcom</td>
<td>51</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Admission Control and Resource Costs</td>
<td>53</td>
</tr>
<tr>
<td>5.1.3</td>
<td>The libscampi</td>
<td>55</td>
</tr>
<tr>
<td>5.1.4</td>
<td>The drivers</td>
<td>57</td>
</tr>
<tr>
<td>5.1.5</td>
<td>mapidlib</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>Kernel-based Software</td>
<td>59</td>
</tr>
<tr>
<td>5.2.1</td>
<td>The combo6 driver</td>
<td>59</td>
</tr>
<tr>
<td>5.3</td>
<td>SNMP</td>
<td>62</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Management functions</td>
<td>62</td>
</tr>
<tr>
<td>5.3.2</td>
<td>SNMP Access</td>
<td>64</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Using the SCAMPI MIB</td>
<td>68</td>
</tr>
<tr>
<td>5.4</td>
<td>Monitoring in IPv6 networks</td>
<td>69</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Monitoring DWDM links and other link layer technologies</td>
<td>69</td>
</tr>
<tr>
<td>6</td>
<td>MAPI Definition</td>
<td>71</td>
</tr>
<tr>
<td>6.1</td>
<td>Passive Monitoring MAPI</td>
<td>71</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Creating and Terminating Network Flows</td>
<td>71</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Reading packets from a flow</td>
<td>73</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Applying functions to Network Flows</td>
<td>73</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Packet formats</td>
<td>73</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Flow Records</td>
<td>75</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Loadable Functions</td>
<td>79</td>
</tr>
<tr>
<td>6.1.7</td>
<td>Some More Complex Monitoring Examples</td>
<td>82</td>
</tr>
<tr>
<td>6.1.8</td>
<td>Implementation issues for MAPI</td>
<td>84</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 *The geographic coverage of the sapphire worm on January 25th 2003.* The photograph is courtesy of www.caida.org. ......................................................... 10

1.2 *Traffic Distribution of the network of the University of Wisconsin for the week 7-13 Sept. 2003.* The photograph is courtesy of www.stats.net.wisc.edu .......... 13

1.3 *Connectivity and speed of GEANT: the European Research Network.* The photograph is courtesy of GEANT. .......................................................... 16

2.1 Accounting architecture ................................................................. 21

2.2 Flow-based reporting ............................................................... 23

2.3 Architecture for QoS monitoring in a CDN .................................. 25

2.4 A content distribution network as a QoS monitoring application .......................................................................................... 26

3.1 *The SCAMPI network monitoring system.* .................................. 30

4.1 COMBO6 motherboard ................................................................. 32

4.2 Structure of the adapter firmware ................................................. 33

4.3 Parallelized approach for 10 Gb/s processing .................................. 35

4.4 I/O queues (a) ............................................................................. 39

4.5 I/O queues (b) ............................................................................. 40

4.6 DAG architecture ........................................................................ 41

4.7 Main components of the IXP1200 ................................................. 47

4.8 *SCAMPI monitoring system based on commodity-only network interfaces.* ................................................................. 49

5.1 *The main modules of the MAPI monitoring system.* ............... 52

5.2 SCAMPI MIB ............................................................................. 65
LIST OF FIGURES
Chapter 1

Introduction

1.1 Motivation - What is the problem?

1.1.1 The Wild Spread of Cyber-Attacks

As networks get faster and as network-centric applications get more complex, our understanding of the Internet continues to diminish. Nowadays, we frequently discover, to our surprise, that there exist new aspects of Internet behavior that are either unknown or poorly understood. A couple of years ago, for example, the world was surprised to learn that more than 4,000 Denial-of-Service (DoS) attacks are being launched on the Internet every week [19]. This surprising result attracted the interest of public and of media together. Although at that time lots of people had heard about Denial-of-service attacks, most of them did not really know that their magnitude was so high. Most of them they were not simply aware of the wars raging on the Internet.

To make matters worse, besides DoS attacks, malicious self-replicating programs (called worms in the colorful language of computers) continue to plague our networks, to multiply rapidly, and to have the ability to cause damage of unprecedented magnitude. For example, on the 15th of January 2003 at 05:29, the Sapphire (Slammer) Worm was launched on the Internet. Targeting a vulnerability in the software of SQL servers, the worm rapidly infected more than 70,000 computers running such servers. Interestingly enough, it took the worm no more than 30 minutes to infect most of the computers that could be infected. Actually, during its peak expansion the worm was doubling in size every 8.5 seconds. Figure 1.1 shows that geographic coverage of the sapphire worm half an hour (at 06:00) after the worm was released. We see that the worm infected close to 75,000 computers practically all over the Globe even including areas such as the Fidji islands and Greenland. Indeed, Sapphire was the first worm which demonstrated that worms can spread all over the Globe within a time interval, in which human intervention and response is either limited or better yet not possible at all.

Worms like the Sapphire are usually called flash worms because they have the potential to conquer the entire Internet within minutes before any human intervention is possible. Interestingly enough, besides rapidly spreading worms, there also exist slowly spreading worms: the stealth worms. Stealth worms capitalize on the fact that automatic worm detection systems usually detect the spread of new worms - not the worms themselves. This is because new worms contain code unknown to worm detection systems, and thus can not be detected. Their spread however, is usually exemplified by a sudden increase in traffic and/or the existence of peculiar traffic patterns, that can be detected. Stealth worms

\[1\] http://www.cnn.com/2001/TECH/internet/05/24/dos.study.idg/
http://www.nytimes.com/2001/05/24/technology/24HACK.html
spread very slowly trying to elude automatic worm detection systems. Masquerading as ordinary traffic, stealth worms attach themselves to popular programs, such as peer-to-peer file sharing systems, and propagate along with the ordinary traffic of such programs.

Despite our knowledge of worms, a large number of them still roam the Internet. Recently, for example, the Blaster worm infected more than 400,000 computers. Although not as rapidly spreading as Sapphire, Blaster is a unique worm in the following sense: after its original release on the Internet, Blaster was quickly followed by Welcia, a good worm whose goal was to combat Blaster. Welcia enters computers through the same security hole that Blaster exploits and attempts to download and install the software patch that would close the security hole and that would make the computer immune to Blaster. After patching the local computer, Welcia attempts to invade and patch the other computers within the same Intranet. Unfortunately, by being over-eager at searching for other vulnerable machines to patch, Welcia eventually leads to a situation equivalent to an internal Denial-of-Service attack: the Welcia-infected computers continually probe all the rest of the computers in the Intranet trying to find whether they are vulnerable to Blaster. To make matters worse, once Welcia installs itself in a computer, it is not programmed to remove itself before 2004. As a result, it was not Blaster, but Welcia that caused more damage in corporate computers. To summarize, over the last few months we have seen Blaster spreading through the Internet, Welcia chasing after Blaster, and security administrators chasing after both worms.

In addition to self-replicating worms, viruses are increasingly starting to represent a significant cyber-threat as well. Recent viruses were able to gain access to passwords, bank accounts, and important personal information. While worms are self-replicating programs that multiply without any human intervention, viruses usually depend on human help in order to multiply. Masquerading themselves as interesting content, malicious viruses attach themselves to innocent looking email messages prompting the user to “click” on them. Once uses click on the attachment, the malicious executable starts executing and conquering the local computer. One of the most widely spread viruses, the BugBear-B,
1.1. MOTIVATION - WHAT IS THE PROBLEM?

recently hit several computers on the Internet where it installed a keyboard logger, a snooping program, that was able to steal passwords and gain access to secret information. Keyboard loggers are able to steal confidential information, such as credit card numbers, despite the fact that users may take all security precautions, such as using secure socket layers or a similar information encryption mechanism for all their communication with the outside world. This is because secure socket layers protect the information from snoopers that reside outside the user’s personal computer, but not from snoopers that have penetrated the user’s personal computer. Indeed, keyboard loggers are able to steal confidential information before it reaches the secure socket layer, and thus before it is encrypted.

Before users managed to recover from the effects of BugBear-B, a new virus started spreading: Sobig.F. Sobig.F has been quite effective and has sent more than 100 million email messages during its first week of activity causing more than one billion dollars of damage. However, this billion-dollar damage is rather small compared to what these cyber-attacks can potentially do. So far, the average victim of a cyberattack lost no more than a few hours of work trying to find what is wrong with the system and to upgrade it with the appropriate patches. We should not be fooled, however, by the mediocre damage that the current worms have done. These worms and viruses are powerful programs that can cause massive damage of unprecedented effect. For example, these worms have the ability to destroy (even temporarily) our Internet infrastructure. Such devastating worms have been described in detail in the literature and are known as “Warhol” worms. After spreading in less than 15 minutes to most of its potential victims, a Warhol worm would install itself in startup scripts so that it is always started when the machine reboots. It would then overwrite random pieces of non system files while at the same time it would restore the file modification times to their original value making it difficult to find which files have been altered. Although altering random pieces of randomly selected files is slower than deleting the entire files, it is more difficult to be detected by system administrators. After establishing itself to several millions of computers, the Warhol worm would start a massive Distributed Denial-of-Service attack to major sites including antivirus sites that may contain the patch for the Warhol worm, thus hindering all legitimate users from patching their computers. Effectively, the described Warhol worm would shut down the normal operation for most computers connected to the Internet.

1.1.2 Friendly Fire on the Internet

Besides malicious attacks, there also exist non-malicious incidents on the Internet that may have a big negative impact on its operation. For example, it was recently discovered that a “combination of Microsoft software features and misconfigurations was essentially causing a slowly-paced massive distributed denial of service (DDoS) attack on the root name server system.” Actually, it was found that personal computers running the Windows operating system were sending updates for private addresses to root Domain Name Servers (DNSs) without any particular reason. These unnecessary updates were consuming a large percentage of root name server resources, which is exactly the same effect caused by massive Distributed Denial of Service attacks. Obviously, as an increasingly larger percentage of computers runs software of a single vendor, such incidents will become increasingly common in the future. Indeed, when most computers run the same software, even a small un-noticeable software bug can be magnified and have unexpected results.

4http://www.cnn.com/2002/TECH/internet/10/01/hln.wired.bugbear.virus/
5http://theregister.co.uk/content/56/32760.html

scampi@ist-scampi.org

November 13th, 2003
1.1.3 The Need for Internet Traffic Monitoring

All the above mentioned incidents, whether malicious or not, (i) demonstrate our relatively poor understanding of the Internet, and (ii) show the need for better Internet traffic monitoring. Better Internet traffic monitoring will not only improve our understanding of the Internet but it will help us improve its operation by

- providing early warning for new cyberattacks during the first seconds of their spread
- isolating newly discovered cyberattacks during the first minutes of their spread, based on cooperation with firewalls or other network protection systems
- identifying the sources of unexpected performance problems, such as the friendly-fire problem described above
- providing better traffic characterization, even for applications that want to elude characterization, such as peer-to-peer systems

1.1.4 The Benefits of Internet Traffic Monitoring

Besides dealing with the detection of cyber-security threats, better Internet Traffic Monitoring can help us tackle a variety of problems. For example, one of the most frequent requests of network administrators is to find out the applications that generate the largest amounts of traffic that flows through the monitored networks. Unfortunately, current network traffic monitoring infrastructures usually do not provide this kind of information, i.e. current monitoring infrastructures do not provide the necessary information needed to find which applications generate most traffic within a network. This is because several of the current monitoring systems are based on flow-level statistics, such as those provided by NetFlow [28], IPFIX, and related systems and proposals, which associate applications with static well known ports. Such monitoring systems assume, for example, that each application is associated with only one static well-known port. Although this is true for several of the traditional applications, most of the emerging applications, such as peer-to-peer systems and video conferencing systems, use a variety of dynamically generated ports. Therefore, traditional ways of monitoring usually end up unable to monitor the network. Let’s take for example figure 1.2 that shows the distribution of traffic among various applications for the University of Wisconsin for the week of 7-13 September 2003. The legend reads that 1.2% of the outgoing traffic is attributed to KaZaa, 0.5% of the outgoing traffic is attributed to Gnutella, 14.4% of the outgoing traffic is attributed to HTTP (the web), etc. What is most important to see, however, is that the penultimate line of the figure legend states that 69.4% of the outgoing traffic is attributed to “Other” applications, that is, applications besides peer-to-peer systems (such as KaZaa, Gnutella, eDonkey, Napster), besides web, besides ftp, besides Usenet news (nnntp), besides email (smtp), besides ICMP, besides real audio/video, etc. It is surprising to see that close to 70% of the outgoing traffic of the University of Wisconsin does not belong to any of the known popular sources of traffic. It is interesting to see that close to 70% of the outgoing traffic of a University with well-monitored networks is attributed to applications that we do not currently know. Actually, this surprising effect is probably due to the fact that the methodology used to categorize traffic into applications is based on traditional monitoring methods that associate applications with static ports, while an emerging applications of popular applications use dynamic ports. Thus, applications that use dynamic ports, such as KaZaA, seem to consume only 1% of the traffic while previous measurements, form the era when those applications used static ports reported that peer-to-peer applications consumed
1.1. MOTIVATION - WHAT IS THE PROBLEM?

Figure 1.2: Traffic Distribution of the network of the University of Wisconsin for the week 7-13 Sept. 2003. The photograph is courtesy of www.stats.net.wisc.edu

the largest portion of traffic [15]. Indeed, analysis based on passive monitoring tools reveals that peer-to-peer applications consume three times more traffic than analysis based on static port numbers [27]. As a result, in the above example, traditional monitoring methods cannot account for the 70% of the traffic, and in the end, network administrators are not able to see which applications consume which percentage of the traffic: in the above example they do not know who is using 70% of the network's traffic.

Besides finding the applications that generate the largest amounts of traffic in the networks, system administrators are also interested to find out performance bottlenecks in applications running in their domain. For example, users frequently complain that accessing a web server is unusually slow, and turn to network administrators to find what is the problem. The problem may be attributed to a slow server, a buggy protocol implementation, a slow network, a slow client, a lossy connection, etc. Traditional traffic monitoring systems usually provide little information in order to pinpoint this kind of performance problem. For example, the mentioned flow-level statistics, although may provide some insight, are in general of little help in this kind of problem. Advanced network monitoring systems try to solve the above mentioned problem using active monitoring. In active monitoring, servers periodically send a sequence of messages to receivers, record the time when the messages arrive to their destination, and based on these measurements, they are able to find network properties such as bandwidth, latency, loss rate, etc. Although such active monitoring systems may provide insight towards the solution of the performance problem, in many cases that are still inadequate. For example, if the server belongs to a different administrative domain which may not want to cooperate in measurements, such
as cnn.com or abc.com, active monitoring systems can not take accurate measurements of traffic arriving to this server. Similarly, active monitoring systems can not usually find out why a particular client experiences a particular problem: they can only find out the average performance of source-destination pair. Therefore, even after using active monitoring and flow-level statistics, a user may not have the necessary information to deal with the performance problems reported.

1.1.4.1 The SCAMPI Approach

SCAMPI is a step towards improving our understanding of the Internet and towards solving difficult performance and security problems. Besides understanding and improving the Internet as stated in [21], 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.

1.2 The Challenges of Network Monitoring

1.2.1 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.
1.2. THE CHALLENGES OF NETWORK MONITORING

(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 [28]. 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 [12]. Intrusion detection systems, such as snort [26], have been implemented on top of the libpcap [16] 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.

1.2.2 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 [2], 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 1.3). As if this exponential increase were not enough, network monitoring applications continue to become more 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.

1.2.3 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
Figure 1.3: Connectivity and speed of GEANT: the European Research Network. The photograph is courtesy of GEANT.
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 [16], Berkeley Packet Filters [17], and Linux Socket Filters [11]) 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 [12, 18]. 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 bus from much of their load.

### 1.3 Roadmap

The rest of the deliverable is organized as follows: Chapter 2 outlines the network monitoring applications SCAMPI will support. Chapter 3 presents the overall software and hardware structure of the SCAMPI monitoring system. Chapter 4 presents the alternatives for the hardware infrastructure ranging from special-purpose interfaces, to commodity interfaces, to intelligent routers. The chapter focuses especially on the description of the combo6-based SCAMPI monitoring board. Chapter 5 presents the overall software structure of the system, focusing especially on the MAPID daemon, the “heart” of the software architecture. Chapter 6 presents the definition of MAPI, a somewhat improved version of what has been presented in Deliverable D1.2, and finally chapter 7 summarizes and concludes the deliverable. In addition, this deliverable provides several appendices that complement the main document or act as reference manuals.
Chapter 2

Network Monitoring Applications

This chapter describes the monitoring applications that are being developed as part of the SCAMPI project. We will focus on five different types of applications, i.e. pure packet capture, traffic summary reports, threshold alerting, QoS monitoring and security. An overview is given in Table 2.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Application</th>
<th>Flow Operation</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Capture</td>
<td>legacy applications on top of libpcap</td>
<td>classification</td>
<td>full frames or selected headers</td>
</tr>
<tr>
<td></td>
<td>storing selected packet parts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traffic Summary</td>
<td>Netflow-like flow records and related accounting support</td>
<td>Flow Record Statistics</td>
<td>Flow Records</td>
</tr>
<tr>
<td>Threshold Alerting</td>
<td>Threshold checking on traffic entering a node</td>
<td>Flow Statistics</td>
<td>Event</td>
</tr>
<tr>
<td>Quality of Service</td>
<td>Passive Flow QoS Monitoring</td>
<td>flow-based sampling</td>
<td>statistics</td>
</tr>
<tr>
<td>Security</td>
<td>Intrusion detection</td>
<td>signature-based sampling</td>
<td>full frames</td>
</tr>
<tr>
<td></td>
<td>DoS detection</td>
<td>Flow Statistics</td>
<td>Events or Statistics</td>
</tr>
</tbody>
</table>

Table 2.1: Application Overview

*Pure packet capture* will be supported in order to comply with legacy applications that run for example on top of libpcap [16]. In this case, an important method to reduce the volume of the captured traffic stream will be classification and filtering of the packets in the stream. Packet capture will output full captured frames or selected headers that can be stored to disk for later analysis. *Traffic summary* applications on the other hand will capture traffic and export flow records. These NetFlow-like flow records will for example be very useful in accounting applications. When monitoring networks for *threshold alerting*, flow statistics are collected. If a certain threshold for example is exceeded, an asynchronous event is sent to the application. In *QoS monitoring*, statistics about captured flows are collected and are sent back to the application. Flow-based sampling will allow the application to keep up with high-speed networks. In *security* applications such as Intrusion Detection Systems (IDS),
signature-based sampling will be used to filter packets that contain some specified pattern out the stream. The full frame of these sampled packets is sent to the application for analysis. Finally, the Denial-of-Service detection application will monitor network traffic characteristics in order to detect sudden and abnormal changes of traffic statistics that are indications of security attacks.

We chose to implement these applications, not only because of our interest in such applications and their importance in network management and traffic engineering, but also because they focus on different capabilities of the system. All applications use different techniques to deal with high-speed networks. Packet capture focuses on the elimination of packets that are of no interest to the higher layer application running on top of packet capture. Traffic summary reduces the size of the flow records, without losing any valuable information, in order to deal for example with the limitations of memory capacity and PCI bus speed. Threshold alerting in QoS monitoring gather statistics of the flows and use sampling methods to reduce the captured network flow. Security applications, such as Intrusion Detection Systems, often need to analyse all captured data. In this case, efficient algorithms are needed in both the monitoring platform and the application.

2.1 Packet Capture

Pure packet capture will filter selected packets on a monitored circuit and store full frames or selected headers to a disk for a follow-up analysis. It will also support legacy applications that run for example on top of libpcap.

2.1.1 Libpcap Interface

The libpcap interface to the MAPI will translate libpcap functions to MAPI calls. This interface will allow any application written on top of libpcap to be executed on a SCAMPI enabled platform. To support these legacy applications, a modified libpcap library will be developed. Using this library in combination with legacy applications, the performance and functionality of the SCAMPI platform can be compared to traditional libpcap-based systems.

2.1.2 Packet Capture Application

Packet Capture is a very basic monitoring application. It simply captures packets from the wire and saves them to disk. Some filtering can be done on the incoming stream of packets to obtain only those packets that the application is interested in. Furthermore, only certain parts of the packet can be processed or saved to disk. If the user is only interested in the headers of the packets belonging to a certain flow, the application can be configured to capture only the requested information.

The Packet Capture application will be implemented on top of the MAPI. It will be possible to apply a chain of filters to the incoming packetstream and save the full frames or selected headers to disk for a follow-up analysis.

2.2 Traffic Summary Reporting

2.2.1 Accounting Application

Internet and Application Service Providers bill their customers based on their actual traffic or network usage. An accounting application over SCAMPI will provide to an ISP the ability to accurately measure
2.2. TRAFFIC SUMMARY REPORTING

Various characteristics of network traffic, improve their services and provide advanced billing mechanisms and policies. The accounting application should be able to receive or gather input data from many SCAMPI probes, in order to provide (if needed) aggregated statistics or a more comprehensive "picture" of usage for the whole network. This application will also provide necessary information in order to find customers whose traffic volume results in a degradation of service quality, or customers who experience such service quality degradation.

The process of cooperation/integration of SCAMPI platform, the accounting application and the components of a billing system is schematically represented in the figure 2.1.

The accounting application will be based on the SCAMPI platform and the input will be the response from MAPI on specific requests e.g

- traffic volume that corresponds to a range of IPs (quantitative parameter)
- jitter on packets with high priority between a source and destination host (qualitative parameter)

The output of accounting application will be one or more Service Detail Records (SDRs). An SDR is a record with all the necessary information for the usage of a service or the values of QoS parameters needed by a billing system. In Table 2.2 there is a high level description of the most important fields of an SDR.

The rating system is the process that incorporates the definition of tariff elements, the algorithms to implement various rate plans and the configuration of Service Level Agreement (SLAs). The billing
system produces invoices for customers (charge or credit) based on the service and the respective pricing policy. The rating and billing software solutions could be supplementary products in the SCAMPI monitoring platform, in the sense that they can take as input, the output data of the monitoring process and exploit its advantages and re-adjust accordingly the policies applied to the networked based services.

The structure of Forthnet’s existing billing system (FOBIA), follows the scheme presented above. Therefore with the appropriate parameterization it can be integrated with the accounting application by defining SDRs in the proper way. Tests will be conducted on real traffic flowing across FORTHnet network.

SCAMPI and applications over it can provide the means to measure efficiently performance and behaviour at high speed, in order to feed billing components with the meaningful events and billable information. Henceforth, the perspective role of SCAMPI is to empower and encourage the adoption of modern economic and product models in the telecommunications and networking market, based on the emphasis to

1. qualitative measurable performance rather than quantitative criteria.
2. enable the customer validate the received quality.

### 2.2.2 Host Tracking Application

In the last years, many European countries approved laws that require ISPs to:

- maintain traffic traces for some time (6 months or more)
- provide facilities for tracking activities performed by customers

If needed (e.g. in case of an investigation), ISPs must provide the government (e.g. police) a secure interface (e.g. via HTTPS on leased lines not connected to the Internet) to the system that maintains traffic traces.

Many traffic monitoring applications provide some traffic accounting facilities. Most of these applications have been designed for measuring traffic for billing purposes and do not provide facilities for tracking activities.

Traffic flows (e.g. NetFlow) are commonly used by ISPs for network monitoring. High-end routers used to run the backbone are usually able to emit network flows. However, in some cases routers are not able to emit flows when running at high speeds because flow probes have limited resource access (mainly CPU) or because flow generation is performed by the main router CPU and not by the ASICs core used to route packets. The consequence is that it is necessary to develop both a software probe and a collector:

<table>
<thead>
<tr>
<th>ID</th>
<th>Unique Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Specific Info</td>
<td>IP address-IP range-Phone number-User name</td>
</tr>
<tr>
<td>Service Info</td>
<td>Parameter measured or monitored</td>
</tr>
<tr>
<td>Quantity</td>
<td>Traffic volume - Parameter value - Call duration</td>
</tr>
<tr>
<td>Time Period</td>
<td>The date/time period (start-end) of measurements</td>
</tr>
<tr>
<td>Status</td>
<td>The status of SDR</td>
</tr>
</tbody>
</table>

Table 2.2: SDR Fields
2.2. TRAFFIC SUMMARY REPORTING

Flow-based reporting

- the probe will be used whenever network routers cannot cope with the network speed
- the collector will be responsible for collecting flows generated both by the routers and the software probes

The probe will be based on MAPI in order to take advantage of the SCAMPI platform. The collector will be based on CAPI (Collector API), a new API defined by SCAMPI that eases the creation of collector applications. CAPI is responsible for collecting flows, storing them persistently and providing facilities for performing queries on the flows for the purpose of traffic accounting and activity tracking.

2.2.3 Flow Record Export Application

NetFlow is today one of the most commonly used technologies for monitoring network usage and collecting information about network traffic. The work of collecting flow records is usually done by routers which export the flow records to some collector. One problem with this is that as the network speed increases, most routers are not able to do full flow analysis and have to use sampling to keep up.

The flow record exporter application will use the SCAMPI platform to export flow records using the IPFIX protocol and will be used at high speeds where routers can not deliver flow records without using sampling. 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 IPFIX export.

2.2.4 Flow-based Reporting

Flow-based reports uses the information available in flow records to produce a broad range of reports showing various information about the network traffic. Users normally accesses the reports through some web based interface which allows the users to select the report type and time period.

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

Figure 2.2 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.

The web interface will allow users to browse the reports. This interface will be implemented so that it will be easy to support new types of reports. There will also be access control support so that access to some or all reports can be restricted.
CHAPTER 2. NETWORK MONITORING APPLICATIONS

At the end of the project this application will collect flow records from the Flow record exporter application and generate reports based on traffic from a production network. A successful implementation will be able to generate reports based on the traffic from a 10Gbit/s production network.

2.3 Threshold Alerting

Threshold alerting is a common part of network management platforms and is supported in both SNMP (through traps) and COPS (through the COPS reporting mechanisms). There are several uses for these type of alerts which all share a common requirement to the SCAMPI platform: the applications are not generally interested in monitoring information, but want to receive information asynchronously.

In order to configure a threshold alerting application we need to supply the MAPI with a monitoring job that contains:

- zero or more classification rules to narrow the overall traffic volume.
- a metering or counting function.
- a threshold checking function.

By using a blocking read on the results of the latter, we can be notified whenever a threshold has been exceeded.

The above mechanism will be incorporated as part of a short-term traffic engineering research. In this context we need to split traffic entering from a interface on two paths according to a given weight distribution. A first remark is that in this case we have taken a router and scampi device together in one device (using an ad-hoc extension of the SCAMPI software). Secondly, the sampler and metering blocks remain static through the lifetime of the system (only the mapping to an output can be changed). The main goal of this application is to show the functionality of the event-based reaction to monitoring, and this is accomplished by building a feedback loop between the monitoring of the buckets, through the MAPI to a local tunnel management point, and a possible resulting reconfiguration (re-mapping).

2.4 Quality of Service Monitoring

QoS-monitoring analyses the behaviour of a specified (e.g. SLS monitoring) or random stream (e.g. CoS monitoring) throughout a system under observation (ranging from a single link to a concatenation of ISPs).

We will develop and implement a two-layered architecture for QoS monitoring, i.e. a QoS monitoring layer and an application layer. The monitoring layer, belonging to a single Internet Service Provider (ISP), will provide end-to-end QoS statistics of the observed network to the application layer. These statistics include packet loss, network throughput and delay. Any application in the upper layer can request these end-to-end QoS statistics from the monitoring layer.

In the current client-server architecture, a single server serves multiple clients. This approach however has some significant drawbacks. The quality of the offered services often degrades when for example (i) the server cannot handle the load, (ii) the intermediate network gets congested or (iii) the experienced latency is too high. To tackle the performance problems of this classical client-server approach, a lot of alternative techniques have already been developed. In order to reduce the latency

\[1\] this traffic engineering functionality is commonly referred to as tunnel management and is applicable to several areas like MPLS-based TE, where traffic is mapped on an LSP or Ethernet-based TE where traffic is mapped to a VLAN/spanning tree.

scampi@ist-scampi.org 24 November 13th, 2003
for example, caches can be placed in the network, close to the end user. Instead of serving clients using only a single server, server farms in combination with load balancing can increase the service rate of the incoming requests. Despite such efforts, end users still often encounter poor service quality when retrieving bandwidth-intensive content such as audio and video from the Internet. Therefore, Content Distribution Networks have been introduced. In a Content Distribution Network (CDN), the content is replicated to different surrogate servers at the edges of the network. This way, the content only has to pass a few nodes in order to reach the end user, resulting in better quality of service. The CDN will use a Replica Placement Algorithm (RPA) to decide where to replicate which content. Likewise, Content Server Selection Algorithms are used to direct client requests to an optimal server (i.e. a server that can deliver the service with the highest quality).

However, the main problem in the current CDNs is that both algorithms rely in most cases on some raw measurements such as round-trip-time and hop-count. CDNs simply do not have enough information about the current state of the network resources and content servers, resulting in non-optimal decisions of the current algorithms. In this application, we will develop a QoS monitoring layer that offers the requested statistics to, for instance, a CDN application.

Figure 2.3 depicts the architecture we use for QoS monitoring. The Network layer will provide network QoS statistics to the application layer, e.g. a CDN application, above. Based on this information, the CDN layer can replace its content, i.e. run Replica Placement Algorithms, and assign clients to certain content servers, i.e. Content Server Selection. The network layer is part of a single ISP, which provides (on-demand) end-to-end QoS statistics to the application layer, without revealing any sensitive network topology information to the outside world. This way, an application can obtain end-to-end statistics between two access points of the network.

The integration of the monitoring layer with the SCAMPI architecture is illustrated in Figure 2.4. Based on hashing-based sampling (trajectory sampling) or classification, the QoS parameters are measured. This information will be configured in the monitoring agents (i.e. the access points) throughout the MAPI. The results of the individual observation points are correlated in a centralized database.
2.5 Security Applications

2.5.1 Intrusion Detection Application

Network-based Intrusion Detection is a research and development area that aims to improve the security of our cyberinfrastructure through the early detection of intrusion attempts. Network-based Intrusion Detection is usually deployed in the core (or the edge) of the Internet in order to identify possible intrusions as they are being launched. After a possible intrusion is identified, all the information regarding the intrusion is being logged, and the administrators of the system are (optionally) being alerted. The administrators, in turn, may take corrective measures to reduce the effects of this intrusion and possibly patch the security hole that led to the intrusion.

Network-based Intrusion Detection Systems are usually based on a set of rules (also called signatures). Each possible type of intrusion is described by one or more rules. For example, the following rule describes an attempt by an outsider to become super-user (i.e. root) in one of the local systems: \textit{OUTSIDE\_NETWORK} $\rightarrow$ \textit{LOCAL\_NETWORK} TCP 23 content "su root". The above rule states that if a packet is sent from a computer located in the \textit{OUTSIDE\_NETWORK} (an alias for all computers outside the monitored organization) towards a computer in the \textit{LOCAL\_NETWORK} (an alias for all computers in the monitored organization) on port 23 (the telnet port) using the protocol TCP, and the payload of the packet contains the substring "su root", then this is a possible intrusion attempt. In a Network-based Intrusion Detection System each packet is checked against every rule. If the packet matches a rule, it is logged and the administrators may be notified.

We plan to write an open-source Intrusion Detection application using the MAPI interface. We will capitalize on our experience with similar publicly available applications, such as SNORT [26], in order to optimize the application for high-speed networks. The application will detect intrusions based on a well-known set of rules, such as the one distributed by SNORT.

2.5.2 Denial of Service (DoS) Attack Detection Application

A Denial of Service attack detection application has the goal of detecting attacks that attempt to disrupt the ability of a provider to offer service, at a particular performance level, to legitimate users. DoS attacks typically involve flooding a particular network node or server with traffic at a rate much higher than the node or server can handle. Recent measurements indicate that over 90% of DoS attacks involve TCP traffic, with the most common being TCP SYN attacks. TCP SYN attacks involve sending SYN packets, thus starting the TCP handshake, without sending corresponding FIN packets, hence without
completing it and thereby consuming bandwidth and memory resources. Other DoS attacks include UDP flood attacks, fragmentation attacks, and ICMP attacks. Key objectives of a DoS attack detection application are the high detection probability, at an early stage of the attack, and with a low false alarm rate.

A common feature of DoS attacks is that they lead to changes in measurable characteristics of a network traffic flow. Such characteristics can include the type and size of packets, the number of half open connections, and the rate of packets associated with a particular application or port number. Based on this property of changes in network characteristics, DoS attack detection applications are commonly based on anomaly detection models, where the behavior of a measurable network characteristic is compared to its normal behavior, in order to detect significant and abrupt deviations. An advantage of anomaly detection systems is that they do not require any a priori specification of attack signatures, hence they can detect new types of attacks. One approach for describing normal behavior is to use a static characterization (also known as operational model), possibly based on measuring the behavior at some past time; such an approach has the disadvantage of not adapting to changes and trends of normal traffic, which may eventually lead to an increased false alarm rate. Another approach is to adapt to the traffic’s normal behavior through continuous monitoring; however, this advantage can turn out to be a disadvantage, if exploited by the attacker to slowly train the detection system to accept a behavior containing an attack. Finally, there are two categories of anomaly detection procedures that differ in how measurements are processed: sequential processing and batch processing. Sequential procedures involve continuous processing of measurements in order to detect changes as soon as they occur, whereas batch processing procedures involve processing of a batch of data collected within some time interval, hence result in a fixed detection delay.

The DoS attack detection application will be based on time series models, which take into account the time correlations of network traffic measurements. Such an approach still leaves open a number of possible procedures for actually detecting when a change has occurred, including standard deviation models that detect a deviation beyond some number of standard deviations and change point detection (or hypothesis testing) models. For the latter, two specific algorithms will be implemented: the Generalized Likelihood Ratio (GLR) algorithm and the Cumulative Sum (CUSUM) algorithm. All the above methods require measurements of a particular network variable. As indicated above, because the majority of DoS attacks involve TCP traffic, our focus will be on variables related to TCP, such as TCP SYN packets received in some measurement interval, number of half open connections (i.e. difference of SYN and FIN packets received in some measurement interval), rate of TCP requests, and packet sizes.

s will be on variables related to TCP, such as TCP SYN packets received in some measurement interval, number of half open connections (i.e. difference of SYN and FIN packets received in some measurement interval), rate of TCP requests, and packet sizes.
Chapter 3

Overall System Architecture

3.1 Hardware

The overall system architecture of the SCAMPI network monitoring system is shown in Figure 3.1. The system is composed of a 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 through 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\(^2\).

Cutting-edge systems will employ the SCAMPI adapter based on the combo6 card that can not only capture packets at high-speed, but it can also provide filtering and processing capabilities on them.

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).\(^3\)

3.2 Software

Besides the monitoring hardware, the SCAMPI system will also include a significant amount of monitoring software. The software will provide the monitoring applications with a powerful Monitoring Application Programming Interface (MAPI), that will enable them to express their monitoring needs in a device-independent way. the main abstraction provided by MAPI is the network flow. Although flows have been used before in network monitoring systems, MAPI gives flows a first-class status.

---

\(^1\)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 3.1.
Contrary to flows defined in previous monitoring systems, MAPI network flows have a *name*, which enables them to be acted upon. For example, users can create network flows, delete network flows, apply functions to network flows, count the packets of network flows, and so on. For example, traditional flow-based systems like NeTraMet[5] and Cisco’s NetFlow provide a little more than flow-level traffic summaries. To make matters worse users of NetFlow and similar systems have little if any flexibility in defining network flows. On the contrary, network monitoring application programmers are able to define complex network flows on top of MAPI. For example, a network flow may be defined to be composed of “all packets destined to my web server”. Another more complex flow may be composed of “all packets that contain the signature of CODERED virus”. Few, if any, other monitoring interfaces provide similar expressive abilities.

Besides providing powerful abstractions, MAPI also allows for efficient implementation. For example, most of the MAPI code is based around a MAPI daemon (*mapid*) which receives requests from different monitoring applications, synthesizes them and implements them in the most efficient way. In addition, the MAPI daemon is mostly implemented in user-space, avoiding costly operating system overheads associated with kernel implementations such as those provided by Linux Socket Filters for example [11]. The MAPI daemon, in close cooperating with the monitoring hardware maps a memory region in user-space and requests the monitoring hardware to write the captured packets in this memory region. Thus, the MAPI daemon is able to read all the captured packets directly from user-space without invoking the operating system kernel and without copying the packets from kernel space to user space.
Chapter 4

Hardware Architecture

4.1 SCAMPI monitoring adapter

4.1.1 Introduction

An important part of the SCAMPI project is design, development and manufacturing of our own specialized monitoring adapter. The SCAMPI monitoring adapter consists of three levels - the hardware, the firmware (i.e., FPGA program) and the driver. As the hardware is complicated and its flexibility was one of the primary design goals, it is split into two cards - the universal COMBO6 motherboard and the interface card connected as an application daughter board. In the course of the SCAMPI project we intend to develop two versions of the monitoring adapter - the Phase I adapter operating at 1 Gb/s (Gigabit Ethernet), primarily for development and testing purposes and the Phase II adapter operating at 10 Gb/s (10 Gigabit Ethernet), as the final version. We will describe all important components of both versions of the SCAMPI monitoring adapter in the following sections.

4.1.2 COMBO6 motherboard

The COMBO6 (C0mmunication Multiport BOard for IPv6) motherboard has been initially developed as activity of the 6NET project (IST-2001-32603). The original purpose is IPv6 routing. However, as the hardware and the firmware design of COMBO6 is modular and highly flexible, it can be adopted for monitoring purposes.

For the Phase I adapter we will use COMBO6 mainboard developed as part of the 6NET project complemented with the timestamp unit and with modified firmware. For the Phase II adapter, a new version of COMBO6.1 mainboard will be developed adapted for operation at 10 Gb/s. Design characteristics that allow us to reliably achieve operation at 10 Gb/s speed are discussed in section 4.1.6.

COMBO6 is a universal highly flexible card consisting of the following blocks:

- FPGA chip VIRTEX II (produced by Xilinx)
- memory modules SRAM, CAM and DDRAM
- PCI bus interface
- daughter board interface
- auxiliary circuits
VIRTEX II is a pin compatible family of chips with different number of gates. There is no need for board redesign if firmware becomes larger and requires more gates. The firmware is stored in RAM and therefore it can be easily changed. The change of the firmware takes only in the order of several milliseconds. The CAM is used for quick and efficient packet filtering. Received packets are stored in DDRAM, which can be mapped through PCI bus to the system memory space. Therefore, we can avoid unnecessary copying of whole packet when an application needs to read only several bytes from the packet. The physical appearance of COMBO6 is illustrated in Fig. 4.1.

The main change from Phase I to Phase II will be installation of the new CAM designed for PCK block and increased number of SRAM chips which is necessary for higher throughput of HFE and LUP blocks. The second change will be a new PCI interface block.

### 4.1.3 Interface card

The daughter card contains network interface module(s), including input queues. An important function of the interface card is to assign timestamp to each received packet.

For the Phase I adapter we will use COMBO-4MT interface card equipped with four 1 Gb/s (Gigabit Ethernet) ports, which was developed as part of the 6NET project.

A new component will be the precise timestamp unit, connected via the feature card interface. The main part of the timestamp unit is the real-time clock based on a temperature compensated oscillator. It can be synchronized by an external PPS signal (e.g., from a GPS receiver) or by the operating system via NTP protocol. The resolution of the clock is about 10 ns, which means that every packet of a 10 Gb/s stream obtains a unique timestamp. Absolute accuracy of the clock is determined by the synchronization method. It is better than 5 microseconds when PPS is available and better than 1 ms when the clock is synchronized to NTP server on a local network. Without any synchronization, the quality of the clock is determined by the oscillator, whose frequency stability is better than 0.5 ppm (i.e., 0.00005 %) in the temperature range from 0 to 40 degrees of Celsius.

For the Phase II adapter a new interface card operating at 10 Gb/s (10 Gigabit Ethernet) will be developed. We will consider using interchangeable transceivers provided availability of these components. The card will contain another VIRTEX II Pro chip, with a much smaller number of gates than the main VIRTEX II at motherboard. The timestamp unit (TSU) will be implemented directly on the interface card, rather than as an add-on component.
4.1.4 Firmware

The firmware of the VIRTEX II chip is the most important and complicated part of the monitoring adapter. It is split into functional blocks which are separately designed. Some blocks can be used repeatedly in different projects. Recycling of blocks already designed (e.g., for IP header parsing) simplifies the VHDL design process.

Some of the blocks (such as HFE and LUP, which will be described later) are implemented as machines we call ‘nanoprocessors’ running dedicated programs. The nanoprocessors complexity lies between a Finite State Machine (FSM) and RISC processors. Nanoprocessors have limited instruction sets, the nanoprogram is interpreted by a firmware block stored either in FPGA’s BlockRAM or external SRAM. Instruction sets are designed especially for each nanoprocessor. The advantage of the nanoprocessor approach is the possibility to change block functionality at run time, as opposed to the case of FSMs. There is no need to rewrite the source code (e.g., in VHDL), synthesize it, place and route the design, and download the configuration data into the FPGA. It also makes the code design smaller and more efficient. This differs from partial reconfiguration which changes the firmware of the FPGA. With partial reconfiguration, it is necessary to make all design development steps. The structure of the firmware is illustrated in Fig. 4.2.

**HFE** Header field extractor is a preprocessing unit which extracts valid data from IP and TCP headers and stores them in a unified header structure suitable for further processing.

**LUP** Lookup processor matches patterns in a packet header. The first step is matching header with CAM (272 bits wide) for IP addresses, flags, etc. The results is a pointer to the tree parsing automaton. As the second step, the automaton compares header fields with specified constants, such as port ranges, etc. The available timeslot allows to do sequence of nine such comparisons.

**TSU** Timestamp unit assigns high resolution timestamps to packets derived from local the clock. The clock may or may not be synchronized by external PPS (Pulse Per Second) signal. Timestamp is represented by a 64-bit value in a fixed point format, where 32 bits represent the fraction of a second.

**DRAM** Dynamic RAM stores received packets. It is directly accessible from the user space.

**SAU** Sampler unit is designed to reduce data exchange rate over the PCI bus when only samples are required. The unit provides both deterministic (i.e., each n-th packet is passed through) and probabilistic sampling (i.e., a packet is passed through with probability 1/n).
CHAPTER 4. HARDWARE ARCHITECTURE

**STU** Statistic unit supports statistics computing. It counts packets and calculates $\sum x$ and $\sum x^2$ (where $x$ is the length of the packet) for each of up to 256 categories of subflows defined by any 8 bits of the header.

**PCK** Payload checker performs content-based filtering. It can search up to 500 substrings of 16 bytes in the packet payload.

**PCI** PCI Controller

For the Phase II adapter, SAU and STU blocks will be added, LUP block will be more advanced, supporting multiple concurrent monitoring applications and filtering blocks (HFE and LUP) will be implemented in several instances in order to process full 10 Gb/s speed.

The following monitoring functions will be supported in adapter firmware:

- Phase I adapter:
  - One filter based on a combination of header fields (Ethernet, IP, TCP, UDP) expressed in tcpdump syntax.

- Phase II adapter: Multiple filters (one per open socket) based on sequence of:
  - filter based on a combination of header fields
  - deterministic and probabilistic packet sampling
  - filters based on searching payload for multiple 16-byte strings

### 4.1.5 Driver

The driver, which runs in kernel space, provides interface between user space and the physical adapter. We decided to write driver for Linux, which is now a preferred platform for most high-performance networking end hosts. Drivers for other platforms can be written later based on demand. The main functions of the driver include:

- initialization of the adapter
- downloading of the FPGA program
- transfer and optional transformation of control information and data
- translation of user level filter description into nanoprogram instructions for HFE and LUP

As part of the development work, a special driver will also be developed for the commodity Ethernet adapter (such as 3COM) with the same interface as the final driver for the SCAMPI adapter. This special driver will enable faster development of higher-level software.

For the Phase II adapter driver will be upgraded to support functionality of new firmware blocks.
In this section we describe design features of individual components that enable operation at 10 Gb/s. As the COMBO6.1 mainboard (but this also applies to COMBO6 mainboard) performs pipelined processing in cascaded blocks, we can study throughput of each block independently. The shortest time required for packet processing is 50 ns, which represents 64 byte packets arriving at 10 Gb/s.

Blocks of the path from the input up to the filtering unit are most critical from the throughput viewpoint. They have to operate always at the full line speed. Other blocks deal with the data that already passed filters and therefore their throughput can be lower if necessary for technical reasons. Input circuits include transceiver, MAC chip and preprocessor unit, which is implemented in a subsidiary VIRTEX II Pro chip. Specification of this components permits operation at the full line speed.

**HFE and LUP blocks**

Simulation showed that HFE and LUP blocks can process more than two millions of packets per second. It is enough for 1 Gb/s link as the time distance between two packets is not lower than 500 ns. The process of filtering will be parallelized. Ten filtering units will serve the input in cycle. All units share one CAM. Access to CAM takes 40 ns, therefore the speed of searching in CAM is sufficient. Instructions for LUP are stored in SRAM with access speed 10 ns. It implies that one SRAM can serve four LUP units and together three SRAMs have to be installed on the Phase II adapter. The 500 ns time limit is enough long for one CAM access and for subsequent processing of nine instructions of LUP nanoprocessor. Details can be found in [9]. The parallelized processing is illustrated in Fig. 4.3.

**STU block**

There is only one time critical operation in the STU block, which is the square computation. VIRTEX II provides a multiplication unit taking about 18 ns for each operation. As only one square computation per packet is required, STU is capable to process full 10 Gb/s traffic.
CHAPTER 4. HARDWARE ARCHITECTURE

PCK block

The PCK block is based on CAM organized as 272 bits in 8000 rows (the same type as CAM of LUP). Each pattern is up to 16 bytes long. The pattern is stored in CAM in the form of all shifted pictures within a 32-byte grid. It implies that each pattern occupies from 16 (for 16 byte long pattern) to 32 (for 1 byte long pattern) rows. Full CAM can contain at least 256 different patterns. The algorithm requires one CAM access (40 ns) for each 32 bytes of payload, therefore the PCK processing speed is about 5 Gb/s. The PCK block is equipped with input queue which contains pointers to packet body stored in DDRAM. For reliable payload checking, the HFE and LUP blocks have to filter out about one half of the traffic.

PCI bus

The current version of the COMBO6 mainboard, which will be used in Phase I, is equipped with the standard 32 bit / 33 MHz PCI interface. Its maximum theoretical throughput is 132 MB/s (i.e., about 1 Gb/s). The practically achievable throughput is about 100 MB/s. PCI 2.2 specifies the 64 bit / 66 MHz bus, whose throughput is four times higher than for 32 bit / 33 MHz PCI. The problem is that we recognize PCI 64 bit / 66 MHz as a non-perspective solution and even more there is a lack of suitable PCI 64 bit / 66 MHz chipsets for reasonable price. For this reason we prefer to implement the standard PCI-X in Phase II. PCI-X (version 1.0) supports 64 bit / 133 MHz bus with a theoretical throughput of 1064 MB/s (i.e., about 8 Gb/s). The main goal of the SCAMPI monitoring adapter is to support processing in hardware in order to override the limited bus speed. Therefore, the PCI bus of the COMBO6.1 adapter is not the bottleneck of the monitoring system.

4.1.6.1 Chips and modules

FPGA  Virtex II on motherboard, Virtex II Pro on 10 Gbps interface card

Transceiver  We prefer following module: 10GBASE LR 1310 nm DFB laser, other available modules are: XFP Optical Transceivers for 10Gig Serial IGF-32111 TRx 10G 600M XFP LC STD TEMP, four-channel CDWM Transceiver BLAZE LX4-XENBLAZE LX4-XEN

MAC  LVDS output: VSC7320, VSC7321, IFX1810x, XAUI output: BBTX100

PCI bus  PCI 32/66: PLX 9656, PCI 64/66 or PCI-X: At the moment, it is hard to find a chip which really operates at the 64/66 speed. The PLX 9656 provides 64/66 on the PCI bus, unfortunately on the custom side it has just 32/66. Similarly, there is not available any chip for PCI-X. Another solution is to find a vendor which produce PCI-X ASIC. The best solutions would be to use IP core for FPGA, however commercially available cores are very expensive and out of budget of this project. We consider to write our own PCI core.

4.1.7 Combo6 adapter to Combo6 driver interface

Conceptually, the PC-side interface of the Combo6 adapter serves for several functions. The most important of them are loading of FPGA image, loading of packet filter, getting statistics and getting packets. Technically, the communication takes place through a set of control, address and data registers and a PCI bus.
4.1.8 Booting

The booting process is controlled by a register whose bits correspond to individual control signals to or from FPGA. The booting process consists of the following steps:

- Using the PCI bridge on Combo6 (PLX9054) we disconnect FPGA from the local bus, giving control to CPLD.
- Writing to CPLD’s control register we generate a high-low-high pulse on FSPROG signal of given FPGA.
- We wait for FSINIT signal to go high.
- Now we can move configuration data to the data register, CPLD translates our writes to transaction on FPGA configuration bus.
- We wait for the FSDONE signal to go high.
- FPGA is now booted and ready for operation, using PLX we disconnect CPLD from local bus and give control to FPGA.

4.1.9 CAM

Communication with the Content Addressable Memory (CAM), which performs packet filtering, takes place through one control register, three address registers and three data registers. Each register is 32 bits wide, but not all bits of all registers currently carry significant information. The structure of the registers is shown in Table 4.1.

Notes:
- The CAM has 8192 lines and the line address is specified by the address register in bits marked by underscore.
- The CAM data bus width is 68 bits. This defines granularity of all read/write operations. In the code, we call this ‘physical cell’.
- ’Line width’ is configurable (within some constraints). Throughout the code, we refer to this as ’logical cell’ - this is what the user is concerned about (user access granularity). We chose logical width of 272 bits, this gives us 2048 Unified Packet Header patterns to match against.

4.1.10 I/O queues

A principal abstraction used for communication between the adapter and the software is an I/O queue. There is one output packet queue and multiple input packet queues used for transmission and reception of packets. There is also one input management queue and one output management queue used for communicating commands and events to and from the adapter. The queues are implemented by several data structures, see Fig. 4.4 and 4.5. At the top of the hierarchy (going bottom-up in the figure), there is a set of control registers. The iqca register points to I/O queue configuration structure. This structure includes the input and output packet and management queues, described by I/O queue descriptor structures. Each of these structures includes a pointer to an array of I/O block structures, which implement the ring of the corresponding queue. The adapter and the software use these rings as a producer and
CHAPTER 4. HARDWARE ARCHITECTURE

| CAM Registers | | |
|----------------|----------------|
| Register No.  | Purpose                     | Content                        |
| 0             | Command register            | commands for CAM               |
| 1             | Address register            | lower 32 bits of 68-bit address |
| 2             | Address register            | middle 32 bits of 68-bit address |
| 3             | Address register            | upper 4 bits of 68-bit address  |
| 4             | Data register               | lower 32 bits of 68-bit data   |
| 5             | Data register               | middle 32 bits of 68-bit data  |
| 6             | Data register               | upper 4 bits of 68-bit data    |

<table>
<thead>
<tr>
<th>Control register</th>
<th>Command</th>
<th>Register No.</th>
<th>Register Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>0</td>
<td>00000201</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>0</td>
<td>00000000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address registers</th>
<th>Specifies</th>
<th>Register No.</th>
<th>Register Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask area</td>
<td>1</td>
<td>0008</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>Data area</td>
<td>1</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>00000000</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: CAM registers

consumer, their role depends on the direction of data flow (input or output). Each I/O block structure represents one packet, command or event and includes pointers to scatter/gather (S/G) segment descriptors, which finally include address and length of the packet data. All S/G descriptors within an I/O block point to buffers of fixed size configurable on per-queue basis. This allows for efficient memory usage, for example, all JUMBO packets can be directed to one queue.

4.1.11 comboctl utility

Variables in the Combo6 card bus (PCI) address space can be read or set by comboctl utility. If a list of variables is specified on the command line, then comboctl prints the current value of those variables for the specified device. If the -a flag is specified, all variables for the device are printed. If the -w flag is specified comboctl attempts to set the specified variables to the given values.

If -X flag followed by a filename is used, variable names or assignments are read from the specified file (one per line) and handled as if they were specified on the command line. If -S flag followed by a script name is used, the script is executed inside TCL interpreter supporting Combo6 variables.

The -d flag can be used to give an Combo6 special device file, the default is /dev/scampi/0. If -D flag is specified, default variable definitions (given in environment or compiled-in) are printed on standard output. Variable names can be modified by adding ".( number )" at the end, such name is interpreted as if original variable had offset increased by number is its definition.

Use of -L flag ensures that another comboctl invocation will not interfere with the sequence of read/write operations by locking against parallel access in the device driver. The -q flag suppresses all printouts except error messages, the -v flag sets verbose (debug) mode.

For example, to determine whether parity error occurred on PCI bus interface, you can use the following command:
### 4.1. SCAMPI MONITORING ADAPTER

#### S/G segment descriptor:

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>32b</td>
<td>address</td>
</tr>
<tr>
<td>len</td>
<td>16b</td>
<td>length</td>
</tr>
<tr>
<td>pad</td>
<td>16b</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### I/O block structure:

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cflags</td>
<td>16b</td>
<td>control (type independent) flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- owner (HW/SW)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- alert request (HW interrupts after processing the block)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- data store type (plain data/data+desc)</td>
</tr>
<tr>
<td>buflen</td>
<td>16b</td>
<td>length of buffers</td>
</tr>
<tr>
<td>pad</td>
<td>32b</td>
<td>N/A</td>
</tr>
<tr>
<td>srcid</td>
<td>8b</td>
<td>number of HW component</td>
</tr>
<tr>
<td>classid</td>
<td>24b</td>
<td>id of packet class (replication rule)</td>
</tr>
<tr>
<td>tstamp</td>
<td>32b</td>
<td>timestamp</td>
</tr>
<tr>
<td>packlen</td>
<td>16b</td>
<td>packet/data block length</td>
</tr>
<tr>
<td>pad</td>
<td>16b</td>
<td>N/A</td>
</tr>
<tr>
<td>dflags</td>
<td>32b</td>
<td>data-related flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- L2 error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- L3 error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- carrier events</td>
</tr>
<tr>
<td>bufs</td>
<td>5x64b</td>
<td>S/G segment descriptors</td>
</tr>
</tbody>
</table>

#### Buffer queue (for input):

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufs</td>
<td>Nx64b</td>
<td>S/G segment descriptors</td>
</tr>
</tbody>
</table>

#### Configuration of buffer queues:

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qnum</td>
<td>8b</td>
<td>number of buffer queues</td>
</tr>
<tr>
<td>pad</td>
<td>24b</td>
<td>N/A</td>
</tr>
<tr>
<td>qnum times:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blen</td>
<td>16b</td>
<td>buffer length</td>
</tr>
<tr>
<td>pad</td>
<td>16b</td>
<td>N/A</td>
</tr>
<tr>
<td>bnum</td>
<td>16b</td>
<td>number of buffer slots in queue</td>
</tr>
<tr>
<td>pad</td>
<td>16b</td>
<td>N/A</td>
</tr>
<tr>
<td>qaddr</td>
<td>32b</td>
<td>buffer queue array address</td>
</tr>
</tbody>
</table>

Figure 4.4: I/O queues (a)
### I/O queue:

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blks</td>
<td>N×64B</td>
<td>array of I/O blocks</td>
</tr>
</tbody>
</table>

### I/O queue descriptor

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qid</td>
<td>16b</td>
<td>id of queue</td>
</tr>
<tr>
<td>pad</td>
<td>16b</td>
<td>N/A</td>
</tr>
<tr>
<td>bnum</td>
<td>16b</td>
<td>number of I/O blocks in queue</td>
</tr>
<tr>
<td>pad</td>
<td>16b</td>
<td>N/A</td>
</tr>
<tr>
<td>qaddr</td>
<td>32b</td>
<td>I/O queue array address</td>
</tr>
<tr>
<td>(timer)</td>
<td>32b</td>
<td>limit value of interrupt delay timer</td>
</tr>
</tbody>
</table>

### I/O queue configuration:

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qnum</td>
<td>16b</td>
<td>number of packet input queues</td>
</tr>
<tr>
<td>pad</td>
<td>16b</td>
<td>N/A</td>
</tr>
<tr>
<td>oomgmt</td>
<td>96b</td>
<td>management output I/O queue descriptor (IQD)</td>
</tr>
<tr>
<td>imgmt</td>
<td>96b</td>
<td>management input IQD</td>
</tr>
<tr>
<td>opack</td>
<td>96b</td>
<td>packet output IQD</td>
</tr>
<tr>
<td>ipack</td>
<td>N×96b</td>
<td>packet input IQDs</td>
</tr>
</tbody>
</table>

### Control registers:

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iqca</td>
<td>32b</td>
<td>address of I/O queue configuration structure</td>
</tr>
<tr>
<td>timer</td>
<td>32b</td>
<td>limit value of interrupt delay timer</td>
</tr>
<tr>
<td>intinfo</td>
<td>32b</td>
<td>address of interrupt information structure (TBD)</td>
</tr>
<tr>
<td>command</td>
<td>32b</td>
<td>control of DMA machinery</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- initialize</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- start running</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- timer activate</td>
</tr>
<tr>
<td>status</td>
<td>32b</td>
<td>state of DMA machinery</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- up/down</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- paused/running</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- timer active</td>
</tr>
</tbody>
</table>

---

Figure 4.5: I/O queues (b)
4.2 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 64bit timestamp is added to each captured packet.

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

Figure 4.6 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. New DAG cards with on board processing capabilities are under development.

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 transfered 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 \texttt{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, \texttt{dag\_nextpkt}, that can be

```bash
comboctl pci.config.status.perr_detected

and to set duplex mode on the third port:

comboctl -w ports.port3.mii.bmcr.fdx=1
```
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 in the DAG header indicating which interface the packet was captured by.

### 4.2.1 Timestamp format

The 32 most significant bits of the DAG timestamp are the same as a Unix `time_t` and is set from the host’s system clock. The least significant 32 bits represents the fractional part of the timestamp in the specified second. This gives a resolution of $2^{-32} = 0.23ms$ 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.

### 4.2.2 The DAG API

The DAG API provides programmers an easy-to-use interface for reading packets from DAG cards. The main purpose of the API is to provide users access to captured packet through an efficient zero-copy memory mapped method. The following functions are defined by the API:

- **dag_open** returns a file descriptor to the DAG device node.
- **dag_close** closes the access to the DAG device node.
- **dag_configure** used to configure various parameters like maximum number of bytes that should be captured of each packet by the card.
- **dag_mmap** returns the base address of the circular buffer used to store packets.
- **dag_start** starts a measurement session
- **dag_stop** stops a measurement session
- **dag_offset** this function takes as parameter a pointer to the last packet processed by the user and returns a pointer to the last new packet written to the circular buffer by the DAG card.
- **dag_nextpkt** fast and efficient inline function that works on the pointer returned by dag_offset to provide easy per packet processing capabilities.

The following is a simple program showing how the DAG API is used to read packets from a DAG card and write them to disk.

```c
// open the dag device to capture packets
dagfd=dag_open(dagname);
// mmap the packet buffer in user space using pointer buf
char * buf = dag_mmap(dagfd);
//start capturing packets
```

scampi@ist-scampi.org 42 November 13th, 2003
4.3 INTELLIGENT ROUTERS

dag_start(dagfd);
oldoffset=0;

// read packets from the card and dump them in an output file
for(;;){

// read captured packets
offset=dag_offset(dagfd,&oldoffset,0);

// the device captured packets with total size equal to "diff"
diff=offset-oldoffset;

// dump the captured buffers into the output file
// the captured packets start at buf+oldoffset
write(STDOUT_FILENO,buf+oldoffset,diff);

oldoffset=offset;
}

4.3 Intelligent Routers

4.3.1 Overview

Modern routers are based on a two level architecture:

- switching plane, performed in ASICs, that performs all the switching operations;
- control plane, implemented on a embedded PC with a Unix-like OS, that controls and instruments all the routing operations.

As ASICs deliver enough power for both switching and monitoring traffic, the control plane can instrument the router to perform in ASIC operations including:

- Traffic filtering
- Traffic forwarding, mirroring and conversion across various media and protocols (e.g. IP-over-ATM to IP-over-Ethernet).
- Traffic accounting.

The advantage of monitoring traffic on the switch are multiple:

- All the operations are performed at wire speed.
- Media type independence.
- There is no need to add an external PC-based probe for analysing traffic.

scampi@ist-scampi.org 43 November 13th, 2003
Unfortunately, as switches need to give priority to their main task (i.e. packet switching), the monitoring capabilities provided are limited to a limited set of counters whose value is incremented when an incoming/outgoing packet matches the filter associated with the counter.

Due to these limitations, in the context of the SCAMPI project, intelligent routers are used as follows: the router filters and mirrors (also converts in right media type) the traffic towards a PC-based probe that performs sophisticated traffic accounting and measurement.

As the most advanced intelligent routers currently available are manufactured by Juniper Networks, the rest of the document will focus just on this family of routers.

### 4.3.2 Traffic Filtering and Mirroring Overview

Juniper routers base the filtering and mirroring capabilities on the firewall component. Each interface can be configured to apply a firewall filter to each incoming/outgoing packet as follows:

```plaintext
[edit]
interfaces {
    interface-name {
        unit logical-unit-number {
            family family-name {
                filter {
                    input filter-name;
                    output filter-name;
                }
            }
        }
    }
}
```

The `input` statement contains the name of the firewall filter to be evaluated when packets are received on the interface. The `output` statement contains the name of the firewall filter to be evaluated when packets are transmitted on the interface. When a filter is applied to an interface, such filter is evaluated against all the data packets passing through that interface. The filter definition is the following:

```plaintext
firewall {
    family family-name {
        filter filter-name {
            interface-specific
            term term-name {
                from {
                    match-conditions;
                }
                then {
                    action;
                }
            }
        }
    }
}
```

The \textit{family-name} statement can be set to \texttt{inet} for IPv4 and \texttt{inet6} for IPv6. The filter-name is an alphanumeric name (e.g. spam-filter) that is used to identify the filter in the previous interface statement. Each firewall filter consists of one or more terms. A term defines a filter statement that is evaluated in the order in which are specified in the configuration. The \texttt{from} statement specifies the match conditions. The \texttt{then} statement specifies the action to perform in case of match. Match conditions include (but are not limited to):

- source/destination port
- source/destination IP address
- packet type/protocol/length
- ICMP type/code
- VLAN id
- TCP flags
- fragments (IPv4 only)
- next-header and traffic-class (IPv6 only).

Action conditions include (but are not limited to):

- accept
- discard
- reject
- count
- logical
- sample

Among the above actions, the two most important actions are:

\textbf{count}, that allows packet to be counted into the Juniper Firewall MIB, in a variable named filter-name, accessible via SNMP.

\textbf{sample}, that allows packets to be sent either to destination or to a mirror interface with/without the specified sample rate.

A simple filter definition for mirroring/sampling broadcast traffic is the following:

```
firewall {
    family inet {
        filter broadcast-access-control {
            from {
                protocol udp;
            }
            then {
```
4.3.3 Configuring Traffic Filtering and Mirroring

The configuration of traffic filtering and mirroring is usually performed via command line. However in the context of SCAMPI this means of configuration is not really effective as those facilities are not configured once by a network administrator but rather by applications at runtime via the MAPI API.

In order to dynamically (re)configure the router, the best solution is to make mapid communicate in JUNOScript, an Extensible Markup Language (XML) application that Juniper Networks routing platforms used to exchange information with client applications. The JUNOScript API is a programmatic interface that simplifies the development of applications or libraries that need to interact with the router configuration. A basic JUNOScript session works as follows:

- The client application establishes a connection to the JUNOScript server running into the router, and opens the JUNOScript session.
- The JUNOScript server and client application exchange initialization tags, used to determine if they are using compatible versions of the routing software and the JUNOScript API.
- The client application sends one or more requests to the JUNOScript server and parses its responses.
- The client application closes the JUNOScript session and the connection to the JUNOScript server.
- The JUNOScript API can be accessed from both C and Perl. Months ago the SCAMPI consortium has released a JUNOScript-enabled libpcap in order to demonstrate the feasibility of this approach. Based on this experience, the MAPI may also take advantage of JUNOScript for transparently setting filters and configuring mirroring into the Juniper router.
4.4 The IXP1200 network processor

Like the SCAMPI Combo6 card, the Intel IXP network processors provide a platform that is capable of processing data at a very early stage, before the data hits the PCI bus. A design for supporting MAPI on an IXP1200 is described in this section. The IXP1200 contains a single StrongARM processor (running Linux) and 6 microengines (running no operating system whatsoever). Figure 4.7 shows the main components of an architecture where a number of IXP1200 boards are plugged into the PCI backplane of a host.

A general-purpose processor on a host is connected to one or more IXP1200 evaluation boards plugged into the PC's PCI slots. Each of the boards contains a single IXP1200, on-board DRAM and SRAM (256MB and 8MB respectively) and two Gigabit Ethernet ports. The board drawn in Figure 4.7 is based on the Radisys ENP-2506 IXP1200 board.

The IXP1200 itself contains a two-level processing hierarchy, consisting of a StrongARM control processor and six independent RISC processors, known as microengines (MEs). On the IXP, the StrongARM runs the Linux operating system, while the MEs do not run any operating system. Each ME supports 4 hardware contexts that have their own PCs and register sets (allowing it to switch between contexts at zero cycle overhead). Each ME runs its own code from a small (1K) instruction store.

MEs control the transfer of network packets to SDRAM in the following way. Ethernet frames are received at the MACs and transferred over the proprietary IX bus to an IXP buffer in 64 byte chunks, known as mpackets. From these buffers (which Intel has called RFIFOs, even though they are not real FIFOs at all) the mpackets can be transferred to SDRAM. MEs can subsequently read the packet data from SDRAM in their registers in order to process it.
4.4.1 APIs

When the IXP is used for implementing the MAPI, code will need to be provided in all levels of the processing hierarchy, sketched in Figure 4.7: on the MEs, on the StrongARM (Linux kernel), and on the host processor (both kernel and userspace). The following components can be distinguished:

mapIXPdrv: this module provides functions that support MAPI more or less directly (e.g. \texttt{mapidrv\_create\_flow}, \texttt{mapidrv\_aply\_function}, etc.).

IXP\texttt{lib}: this is the library that provides all functions needed for map\texttt{i}XP\texttt{drv} functions to be implemented (e.g. \texttt{ixp1200\_get\_next\_packet}, \texttt{ixp1200\_apply\_function}, etc.).

IXP\texttt{drv}: the kernel driver for the IXP. It provides all the functions that are needed to implement IX-\texttt{P}lib, including the implementation of memory mapping the memroy of the IXP board to the application, etc.

IXP: the code on the network processor itself. It consists of code on the StrongARM and code on the MEs, responsible for receiving the packets and processing them.

Most relevant for a systems programmer implementing the MAPI is the API exported by IXP\texttt{lib}. Here the following functions are provided:

- \texttt{int ixp1200\_open (char *devname)}
  opens the IXP1200 device (returns file descriptor);

- \texttt{int ixp1200\_close (int fd)}
  closes the IXP1200 device;

- \texttt{flow\_t ixp1200\_create\_flow (int fd, mode m):}
  create all state needed for a flow of MAPI mode \texttt{RAW}, \texttt{COOKED}, or \texttt{HIERARCHICAL} (returns flow descriptor);

- \texttt{int ixp1200\_close\_flow (flow\_t f)}
  destroy all state associated with a flow;

- \texttt{ixp1200\_apply\_function (flow\_t f, function func, ...)}
  add a packet processing function to the linear list of functions associated with this flow;

- \texttt{ixp1200\_remove\_function (flow\_t f, function func, ...)}
  remove a packet processing function to the linear list of functions associated with this flow;

- \texttt{int ixp1200\_connect (flow\_t f)}
  instantiate a flow (perform all admission checks, if relevant);

- \texttt{int ixp1200\_activate (flow\_t f)}
  start/activate a flow;

- \texttt{int ixp1200\_pause (flow\_t f)}
  stop/pause a flow;

- \texttt{int ixp1200\_loop (flow\_t f, int count, callback\_t callback)}
  register a callback;
4.5. COMMODITY INTERFACES

SCAMPI monitoring based on commodity adapters

Figure 4.8: SCAMPI monitoring system based on commodity-only network interfaces.

- `packet *ixp1200_get_next_packet(flow_t f, int copy)`
  get the next packet from the buffer (either by copy or by reference);

- `management_info *ixp1200_get_management_info (int info_type)`
  get information about the status of the monitoring hardware (the components of the IXP1200).

- `int ixp1200_advance_read_position(flow_t f, int offset)`
  Indicate to the IXP that the flow has processed offset packets, so the buffer space may be reused;

- `function_mem_list_t ixp1200_get_function_mem(flow_t f)`
  for each function applied to the flow, get a pointer to memory to be used for direct communication with this function (if applicable). For example, a packet counter may provide a pointer to read-only memory where the counters are updated, while a NETFLOW function may provide memory where flow records are stored.

- `int ixp1200_mempcpy_atomic(flow_t f, void *dst, void *src, int len)`
  Atomically copy a chunk of memory from or to memory areas belonging to functions;

4.5 Commodity Interfaces

4.5.1 Commodity Interfaces in Promiscuous Mode

Although special purpose hardware based on network processors, DAG cards, the combo6 board, and intelligent routers can be used for network monitoring, commodity networks 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 (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 link, independently of whether it is shared or not. This can be done by capturing 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 4.8). 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). To make matters worse, most commodity adaptors interrupt the host processor for each and every packet they receive. Thus, when receiving small packets in high-speed links host processors are overwhelmed with interrupt processing, having left little, if any, capacity to do network monitoring.

4.5.2 The commodity adapter API

Commodity adapters put in promiscuous mode do not have (or at least do not export) any extra processing capabilities that can be capitalized by MAPI implementations. Therefore, a commodity adapter put in promiscuous mode can do nothing more than receive packets and deliver them to the applications above. Therefore MAPI implementations on top of commodity adapters can be implemented on top of the pcap library [16]. The functions exported by pcap that can be used to implement MAPI are:

- **pcap_open_live** is used to obtain a packet capture descriptor and start capturing packets passing through the network.
- **pcap_set_filter** is used to specify a filter program: only packets that match the filter are eventually delivered to the user application.
- **pcap_next** returns a pointer to the next packet.

All the rest of the functionality provided by MAPI, including function application, filtering, etc. will be provided completely in software.
Chapter 5

Software Structure

The overall software architecture of the SCAMPI system is shown in figure 5.1. We see that the central part of the SCAMPI software architecture is the MAPI daemon: mapid. As shown in figure 5.1, the daemon, that runs in user space, communicates with the monitoring applications on top, and with the operating system kernel below. The kernel, in turn communicates with the underlying hardware monitoring system.

5.1 The MAPI daemon

The MAPI daemon (mapid) is responsible for implementing the MAPI functionality. On the one hand, the daemon receives monitoring requests from several different applications, and on the other hand, the daemon receives a stream of network packets from the underlying monitoring system (whether the operating system kernel or the hardware). mapid is responsible for applying the monitoring requests of the applications on the incoming stream of packets, for calculating the requested statistics, and for delivering the appropriate network packets to the relevant monitoring applications. We can picture the mapid as a sophisticated de-multiplexer. It receives a stream of packets from the underlying system, processes the packets, discards some of them, and divides the remaining into several streams: each stream destined to a different application.

5.1.1 mapidcom

mapidcom is the module that handles all the communication with the user applications. mapidcom consists of an infinite loop that receives and processes requests from applications. Each request consists of a structure as follows:

```c
struct mapiipcbuf {
    mapiipcMsg cmd;
    mapiFunction function;
    mapiFunctArg fargs[FUNCTARGS_BUF_SIZE]; // holds function arguments
    char str[128]; // only for dev parameter of create_flow()
};
```

The structure contains all the necessary arguments to implement this request:

- `cmd` contains the request sent by the application to the mapid. Such requests are:
Figure 5.1: The main modules of the MAPI monitoring system.
5.1. THE MAPI DAEMON

CREATE_FLOW
APPLY_FUNCTION
READ_RESULT
CONNECT
CLOSE_FLOW
READ_ERROR

For example the CREATE_FLOW request creates a flow, the APPLY_FUNCTION request applies a function to it, and so on.

- `str` is the name of the monitoring device that will provide the packets to the network flow.
- `function` is the function that must be invoked on each packet of the network flow. The supported functions are listed in appendix B.
- `fargs[FUNCTARGS_BUF_SIZE]` is a buffer that is being used to keep the arguments of the function being invoked.
- `fd` is the network flow descriptor.

5.1.2 Admission Control and Resource Costs

Admission control for MAPI has been designed as an independent module. It is a daemon process, known as `authd`, that runs standalone and uses shared memory IPC to communicate with `mapid`. It is responsible for authenticating a user, as well as checking his/her credentials against the specifications of a flow before connecting the flow to the system. In other words, if the application does not have the appropriate privileges (credentials) to create a flow with these characteristics (filters, functions applied to it, etc.) it will be rejected. At the same time the amount of resources needed by a flow is calculated and returned to `mapid` to determine their availability in the system. If not enough resources are available, the flow may still be rejected.

5.1.2.1 Admission Control Inputs

The information required for the authentication of a user include: his/her public key, a positive integer number, and the same number encrypted with the user’s private key. This information is enough to assure that the user actually holds the public/private key pair of the public key used for identification (and hence, that users are who they say they are). For convenience, we can use the flow id (the file descriptor) for this purpose.

The information required to authorize a flow that a user wants to connect include:

- the user’s public key;
- the user’s credentials (using the Keynote format [3]);
- the device with which the flow is associated;
- a list of the functions that are going to be applied to the flow (e.g. filters, counters, samplers, etc.).
The device name and the list of the applied functions are used to generate assertions about the flow. Such assertions cover for example the type of the functions applied, the number of instances of a specific function type, the position of the instances (in the linear list of applied functions) and the arguments passed to them. For example, one may want to specify that a user is allowed to apply a string search function, but no more than one and only if the string search function is applied after a sampling function that grabs no more than 50% of the packets. The assertions and credentials are processed using Keynote to check whether there are no conflicts between the user’s credentials and the flows specifications.

An example of the sort of assertions that may be expressed in the Keynote credentials with respect to this is shown below. In the credential shown here, an authorizer grants a licensee (both identified by their public keys) the right to apply certain functions to their flow, subject to specific conditions. These conditions specify that the flow request is only valid in application domain MAPI, if the device that the flow is created for is one of eth0..eth9, if the flow has fewer than 2 packet counters, fewer than 2 string search operations and at least one sampler (with a specific first parameter), where the sampler should be applied before the string search (presumably to prevent a computationally intensive function like string search from being applied to every packet). The credential is signed by the authorizer, e.g. the system administrator, or someone to whom the system administrator has delegated part of his/her powers (via another set of credentials).

KeyNote-Version: 2
Comment: Within the application domain MAPI, the authorizer grants the licensee(s) the right to use a set of functions under specific conditions (e.g. regarding the number of occurrences and the order in which these functions are applied). Note: keys and signatures are abbreviated for clarity.
Authorizer: "rsa-base64:MIGJAoGBAMbP4gS2x72ZF1PrhN//VEXfYMtb="
Licensees: "rsa-base64:MIGJAoGBAKmynDiwNtAKd6sGTHulfuyOoAp1="
Conditions: app_domain == "MAPI" && device_name =~ "eth[0-9]" && @$("PKT_COUNTER.num") < 2 && @$("STR_SEARCH.num") < 2 && @$("PKT_SAMPLER.num") > 0 && @$("PKT_SAMPLER.param.1") < 0.5 && @$("PKT_SAMPLER.first") < @$("STR_SEARCH.first") < 2 && -> "true";
Signature: "sig-rsa-sha1-base64:i3zsmSmFmcS7SegUIFgJ6921In+U="

While this is a fairly trivial sort of access condition, it shows the power that such admission control offers. Not only does it guard against unauthorized access, it also helps in preventing misconfiguration. If needed, much more complex conditions can be specified, e.g. concerning the parameters of functions (including maximum and minimum values), their absolute and relative positions in the list of functions to be applied, etc.

Finally for a flow to be allowed to connect, it must be checked whether the system has the necessary resources available. Authd holds the responsibility for calculating the amount of resources needed by a flow. The amount of resources needed by a flow is determined by using the list of applied functions. Each applied function imposes a cost in terms of resources needed by the flow to be connected. The cost of each function is going to be dependent on its type as well as its arguments, and it is measured in terms of processing time, memory consumption, etc. Each function type is associated with an arithmetical formula that can also include the function’s arguments. This formula, along with the function’s arguments, is passed to a cost calculation function to provide the cost of applying the function.
5.1. THE MAPI DAEMON

formula consists of a simple expression in a subset of the language that was defined for FFPF \(^1\) (it includes the common arithmetic operations and refers to a function and its parameters, e.g. memory usage equals \(x \cdot \text{param}_1 + y \cdot (\text{param}_2 + \text{param}_3)\)). The sum of the costs of all applied functions consists of the required resources of a flow. Not specifying the cost formula will be interpreted as zero resource consumption. In other words, a site that does not require such elaborate resource control may simply leave these entries blank.

5.1.2.2 Authd and Mapid communication

Authd and mapid exchange information through a shared memory segment. The functions used to access shared memory are listed in Appendix F. Synchronization of the two processes is possible using a semaphore set. A typical request to authd would include the steps described below.

Steps for mapid:

- lock the shared memory segment;
- store the request data to the shared memory segment;
- signal authd;
- wait to receive a signal from authd;
- get the results from the shared memory segment;
- unlock the shared memory segment.

Steps for authd:

- wait to receive signal from authd;
- process the data in shared memory and place results;
- signal mapid.

The functions used to manage the semaphore set and perform the actions shown above are listed in the appendix F.

All the information that need to be passed to admission control are included in the `scampi_auth` data-type that is described in the appendix F. The results returned by authd include an authorization value and the amount of resources needed by the flow. These are included in the `flow_auth` data-type described also in the appendix. Mapid holds responsibility to check if the amount of available resources is adequate to connect the flow.

5.1.3 The libscampi

The typical MAPI implementor usually interacts with the `libscampi` library instead of interacting directly with the combo6 driver or the card itself. The `libscampi` library exports the following functions:

\(^1\)See appendix F.
CHAPTER 5. SOFTWARE STRUCTURE

```c
int scampiOpen(char *name);
int scampiClose(int fd);
int scampiSetOption(int fd, int option, void *value);
int scampiGetOption(int fd, int option, void *value);
unsigned char * scampiGetNextPacket(int fd, int * size);
/*
* Returns pointer to specified number of new packets and locks these packets
* in the circular buffer - not yet implemented.
*/
unsigned char * scampiGetNextOffset(int fd);
* Unlocks any packets locked on behalf of this application
* Input:
  * fd - file descriptor
*/
void scampiUnlockPackets(int fd);
```

A typical sequence of actions performed in the software that uses libscampi is as follows:

1. A program opens a SCAMPI device (this simple call hides all initialization including memory mapping):
   ```
   int fd=scampiOpen("/dev/scampi/0");
   ```

2. Then (or anytime later), a program can set or get SCAMPI device options:
   ```
   scampiSetOption(fd, option1, &value1);
   scampiSetOption(fd, option2, &value2);
   ```

   The following options are supported (but not all of them are necessarily implemented by early SCAMPI driver and libscampi versions):
   ```
   • SCAMPI_LIB_FILTER - sets packet filter
   • SCAMPI_LIB_SAMPLER - sets packet sampler
   • SCAMPI_LIB_STATMAP - sets bitmask for statistic counters
   • SCAMPI_LIB_STATISTICS - gets statistics
   • SCAMPI_LIB_FUNCTIONALITY - gets functionality supported by this adapter
   ```

3. Next, packets can be read:
   ```
   int size;
   char *packet=scampiGetNextPacket(fd, &size);
   ```

4. Finally, a program should unlock the last packet and close the SCAMPI device:
   ```
   scampiUnlockPackets(fd);
   scampiClose(fd);
   ```

```
scampi@ist-scampi.org 56 November 13th, 2003
5.1.4 The drivers

Each monitoring device is supported by device-specific user-level code which is called “the driver” for this device. Figure 5.1 shows three such drivers: one for the combo6 card (mapicomb06drv), one for the DAG card (mapidagdr), and one for commodity interfaces put in promiscuous mode (mapinicdrv). Each driver implements the following functions:

```c
// create a new flow
mapi_drv_create_flow(char * dev, int fd)
// closes an old flow
mapiDrv_close_flow(char *dev, int fd)

// connects to a flow - the flow starts receiving packets
mapiDrv_connect(int fd)

// applies function with fargs to all packets of a flow
mapiDrv_apply_function(int fd, mapiFunction function, mapiFunctArg *fargs)

// this is the main function of the driver:
// it reads and processes one-packet-at-a-time
mapiDrv_proc_loop()

// processes one packet
mapiDrv_process_pkt(char * packet, struct mapi_pkthdr * pkt head)

// reads the results of function fid for network flow fd
// the results are being returned into the buffer void * result
mapiDrv_read_results(int fd, int fid, void * result)

// returns the latest error number
mapiDrv_get_errno()
```

5.1.5 mapidlib

The mapidlib is a library that contains a variety of functions for implementing mapi, and especially the functions that are independent of the monitoring hardware or that can be used by more than one drivers. Such functions include:

- Functions that handle network flows

  ```c
  // adds a new flow and to the list of known flows
  int mapid_add_flow(int fd);
  
  // connects to a flow and enables the reception of packets
  ```

---

2Note that these drivers are distinct entities from the device drivers of the operating system kernel.
int mapid_connect(int fd);

int mapid_close_flow(int fd);

// read results from a flow
void* mapid_read_results(int fd, int fid,
int copy);

// adds a new function to be applied to all packets of a flow
int mapid_apply_function(int fd,
  mapiFunction function,
  void* fptr,
  mapiFunctArg *fargs);

// processes a single packet by applying all functions to it
void mapid_process_pkt(const unsigned char* pkt,
  struct mapid_pkthdr* pkt_head);

- Functions applied to each packet

  // filter
  function_bpf_filter(struct mapid_function *funct,
    const unsigned char* pkt,
    const struct mapid_pkthdr *pkthdr)

  // count the bytes of all packets in a flow
  function_bytecounter(struct mapid_function *funct,
    const unsigned char* pkt,
    const struct mapid_pkthdr *pkthdr)

  // count packets
  function_counter(struct mapid_function *funct,
    const unsigned char* pkt,
    const struct mapid_pkthdr *pkthdr)

  // string search
  function_str_search (struct mapid_function *funct,
    const unsigned char* pkt,
    const struct mapid_pkthdr *pkthdr)

  // dumps packets into a buffer for eventual delivery to the applications
  function_to_buffer(struct mapid_function *funct,
    const unsigned char* pkt,
    const struct mapid_pkthdr *pkthdr)
5.2 Kernel-based Software

A portion of the MAPI software runs inside the operating system kernel as device drivers. Device drivers for commodity NICS and for the DAG cards were provided by their manufacturers. However, the driver for the combo6 monitoring card was written by members of the SCAMPI consortium.

5.2.1 The combo6 driver

The next upper layer sitting just above the Combo6 driver exports to the SCAMPI library a set of ioctl() calls and memory mapped regions.

5.2.1.1 ioctl() calls

The following set of ioctl() calls is available:

- **SCAMPI_IOCTL_VERSION** - read int type, gets driver version
- **SCAMPI_IOCTL_APP_NUMBER** - read int type, gets assigned application number
- **SCAMPI_IOCTL_INTERFACES** - read int type, gets number of physical interfaces on this SCAMPI adapter
- **SCAMPI_IOCTL_GET_INTERFACE** - write/read struct scampi_ioctl_get_interface type, gets description of a given physical interface
- **SCAMPI_IOCTL_SUBSCRIBE_INTERFACE** - write struct scampi_ioctl_subscribe_interface type, subscribes to a given physical interface
- **SCAMPI_IOCTL_UNSUBSCRIBE_INTERFACE** - write struct scampi_ioctl_subscribe_interface type, unsubscribes from a given physical interface
- **SCAMPI_IOCTL_RBUFFER** - read struct scampi_ioctl_rbuffer type, gets size of packet content ring buffer
- **SCAMPI_IOCTL_START** - no argument, starts packet capture
- **SCAMPI_IOCTL_STOP** - no argument, stops packet capture
- **SCAMPI_IOCTL_LOCK_NEXT** - no argument, locks next packet to be read from a memory mapped region
- **SCAMPI_IOCTL_LOCK_MULTI** - write u_int32_t type, locks up to given number of packets to be read from a memory mapped region
- **SCAMPI_IOCTL_UNLOCK** - no argument, unlocks all packets
- **SCAMPI_IOCTL_GET_STATS** - read struct scampi_ioctl_stats type, gets statistics and clears all counters

The relevant section of the header file will be as follows:
CHAPTER 5. SOFTWARE STRUCTURE

#define SCAMPI_MAX_FILTER_HEADERS 256
#define SCAMPI_MAX_FILTER_PAYLOAD 16

/* Phase I adapter only supports headers[] field */
struct scampi_ioctl_set_filter {
    /* tcpdump syntax */
    char headers[SCAMPI_MAX_FILTER_HEADERS];
    /* 0 = none, 1 = deterministic, 2 = probabilistic */
    u_int8_t sampler_type;
    u_int32_t sampler_rate;
    char (*payload)[SCAMPI_MAX_FILTER_PAYLOAD];
};

/* ioctl definition */
#define SCAMPI_IOCTL_SET_FILTER _IOW('X', 0x50, struct scampi_ioctl_set_filter)

5.2.1.2 Memory mapped regions

The following memory regions can be mapped from the Combo6 device to upper layers. We intentionally do not specify numerical values of memory addresses, as they can change with different adapter releases. Programmers should use the symbolic constants instead.

- SCAMPI_MMAP_OFFSET_PKT - mmap offset for packet contents
- SCAMPI_MMAP_OFFSET_PKTDESC - mmap offset for packet descriptors
  (struct scampi_mmap_packet_descriptor)
- SCAMPI_MMAP_OFFSET_LOCK - mmap offset for packet locks
  (struct scampi_mmap_app_lock)
- SCAMPI_MMAP_OFFSET_STATS - mmap offset for packet statistics per interface
  (struct scampi_mmap_packet_lstats)
- SCAMPI_MMAP_OFFSET_STATUS - mmap offset for packet status
  (struct scampi_mmap_packet_status)

5.2.1.3 Using Combo6 driver

A typical sequence of actions performed in the software that uses the Combo6 driver is as follows. These actions are normally performed within libscampi and an application that uses libscampi does not need to care about these low-level details.
1. libscampi opens the Combo6 adapter special device file `/dev/scampi/n`, where n is 0 for the first adapter and obtains a Combo6 driver handle:

   ```c
   int fd=open(SCAMPI_DEVICE, O_RDWR);
   ```

2. The Combo6 driver version, an assigned application number and the number of physical network interfaces present on the Combo6 are determined: adapter:

   ```c
   int version, appNumber, ifNo;
   ioctl(fd, SCAMPI_IOCCTL_VERSION, &version);
   ioctl(fd, SCAMPI_IOCCTL_APP_NUMBER, &appNumber);
   ioctl(fd, SCAMPI_IOCCTL_INTERFACES, &ifNo);
   ```

   Selected network interfaces are subscribed to:

3. /* Subscribe to first interface */
   ```c
   struct ioctl_subscribe_interface ifSub;
   ifSub.if_index[0]=1;
   ifSub.if_index[1]=0xFF;
   ioctl(fd, SCAMPI_IOCCTL_SUBSCRIBE_INTERFACE, &ifSub);
   ```

   The size of packet content ring buffer is determined:

4. ```c
   struct scampi_ioctl_rbuffer rbufferSize;
   ioctl(fd, SCAMPI_IOCCTL_RBUFFER, &rbufferSize);
   ```

5. The necessary memory regions are mapped, that is at least the packet content ring buffer, packet descriptors buffer and packet locks:

   ```c
   /* Map packet contents to user space */
   unsigned char *rbuffer;
   rbuffer=(unsigned char *) mmap(NULL, (size_t) rbufferSize.dma_size,
   PROT_READ, MAP_SHARED, fd, SCAMPI_MMAP_OFFSET_PKT);

   /* Map packet descriptors to userspace */
   tPacketDesc *rbufferDesc;
   rbufferDesc=(tPacketDesc *) mmap(NULL,
   (size_t) (rbufferSize.packets * sizeof(tPacketDesc)),
   PROT_READ, MAP_SHARED, fd, SCAMPI_MMAP_OFFSET_PKTDSC);

   /* Map packet locks */
   struct scampi_mmap_app_lock *locks;
   locks=(struct scampi_mmap_app_lock *) mmap(NULL,
   (size_t) sizeof(struct scampi_mmap_app_lock)),
   PROT_READ, MAP_SHARED, fd, SCAMPI_MMAP_OFFSET_LOCK);
   ```

6. The packet capture is started:
CHAPTER 5. SOFTWARE STRUCTURE

ioctl(fd, SCAMPI_IOCTL_START, &rbuffer);

A corresponding ioctl() call SCAMPI_IOCTL_STOP can be used to suspend packet capture when application cannot catch up with the volume of packets received. Note that other applications might still be running after one application executes this call and thus packets can be still overwritten in the ring buffer even if the application stops the capturing. To get a descriptor of the next packet in fifo for a given application and lock this packet, ioctl() call SCAMPI_IOCTL_LOCK_NEXT is used. This call also automatically unlocks the previous packet.

At the end of communication, the remaining lock should be released with ioctl() call SCAMPI_IOCTL_UNLOCK and the file descriptor should be closed. A typical sequence use after initialisation (opening device, subscribing to interfaces and mapping memory regions): SCAMPI_IOCTL_LOCK_NEXT, SCAMPI_IOCTL_LOCK_NEXT, ..., SCAMPI_IOCTL_UNLOCK. Access to unlocked areas is permitted, but not very useful, because the driver or hardware might overwrite data at any time.

Statistics can be obtained via a mmaped area or using ioctl() call SCAMPI_IOCTL_GET_STATS, which also clears all counters.

For loading packet filters, a daemon which communicates via another character device (/dev/combosix/n) will be available. This daemon will communicate with libscampi using sockets or IPC shared memory. It will merge requests from all applications and write the final filter and specific setup to the hardware.

5.2.1.4 Low-level access

The following low-level ioctl() calls are also available:

- **COMBO6_READ** - reads part of card’s address space into the supplied buffer. Argument should be a pointer to struct combo6_ioc_arg
- **COMBO6_WRITE** - writes part of card’s address space from the supplied buffer. Argument should be a pointer to a struct combo6_ioc_arg

The combo6_ioc_arg structure is defined in combo6io.h file.

5.3 SNMP

5.3.1 Management functions

The SCAMPI platform will be a complex system that will need some sort of management. The MAPI management functions will provide applications information about the internal workings of MAPI as well as usage statistics.

The following management functions have been defined:

- `mapi_get_mngt_var` returns simple management variables like the number of active users, number of active flows etc.
- `mapi_set_mngt_var` sets the value of a management variable
- `mapi_get_device_info_list` returns a linked list of information about devices. The following structure is used:
5.3. SNMP

typedef struct mapiDeviceInfoList {
  unsigned index; //index of device
  char *name; //Name of device
  char *description; //Description of device
  char *alias; //User defined alias for the device
  unsigned numInterfaces; //Number of interfaces on device
  int gpsSync; //Positive value if device is synchronized with GPS receiver
  struct mapiDeviceInfoList *next; //Pointer to next record
} mapiDeviceInfoList;

mapi_get_interface_info_list returns a linked list of information about interfaces. The following structure is used:

typedef struct mapiInterfaceInfoList {
  unsigned index; //Interface index
  unsigned devIndex; //Index of device this interface belongs to
  unsigned ifType; //Type of interface
  unsigned ifSpeed; //Speed of interface
  char *alias; //User defined alias of the interface
  enum mapiIfStatus ifStatus; //Status of interface
  unsigned long pkts; //Total captured packets
  unsigned long octets; //Total captured octets
  unsigned long droppedPkts; //Total dropped packets
  unsigned long ifSpeed; //Speed of interface
  char *alias; //User defined alias of the interface
  struct mapiInterfaceInfoList *next; //Pointer to next record
} mapiInterfaceInfoList;

mapi_get_flow_info_list returns a linked list of information about active and recently closed flows. The following structure is used:

typedef struct mapiFlowInfoList {
  unsigned flowIndex; //Flow descriptor of the flow
  unsigned UID; //UID of the process that owns the flow
  unsigned dev; //Device this flow capture packets from
  unsigned interface; //Interface on the device this flow captures packets from
  unsigned long pkts; //Total captured packets
  unsigned long octets; //Total captured octets
  unsigned long droppedPkts; //Total dropped packets
  unsigned long long start; //Timestamp of when the flow was created
  unsigned long long stop; //Timestamp of when the flow was closed
  int gpsSync; //A positive value indicates proper GPS synchronization during the lifetime of the flow
  struct mapiFlowInfoList *next; //Pointer to next record
} mapiFlowInfoList;

An important aspect of the implementation of the management functions is that they should not have a negative impact on the performance of the system. Management functions are therefore not considered high priority.

scampi@ist-scampi.org
It is the MAPI drivers that keep track of the management information and increments the necessary counters as it receives packets from the hardware adapter.

When a user requests some management information through the management functions, an IPC message is sent from the application to the MAPId. The mapidcom module receives the message and collects the necessary information from the MAPI drivers before returning the answer to the application.

### 5.3.2 SNMP Access

A SCAMPI MIB will be developed so that the MAPI management functions can be accessed through SNMP. The SCAMPI MIB will be implemented using NET-SNMP[20]. NET-SNMP is a collection of various tools relating to SNMP 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.

The managed objects available in the SCAMPI MIB will be organized into four main groups as shown in figure 5.2. A full definition of the MIB is included in Appendix G.1.

#### 5.3.2.1 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 interfaces on devices.

##### 5.3.2.1.1 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.

scampiDeviceIfNum An integer representing the number of interfaces on the device.

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

scampiMIB

scampiMIBObjects(1)

scampiDeviceIfTable(1) scampiDeviceIfEntry(1)

scampiDevIfIndex(1)
scampiDevIfAlias(5)
scampiDevIfStatus(6)
scampiDevIfPkts(7)
scampiDevIfOctets(8)
scampiDevIfDroppedPkts(9)
scampiDevIfCounterDiscontinuityTime(10)

scampiFlows(3)

scampiFlowCfgMaxHistLength(1)
scampiFlowCfgMaxTime(2)
scampiFlowTable(3) scampiFlowEntry(1)

scampiFlowUID(2)
scampiFlowIndex(1)
scampiFlowIfIndex(3)
scampiFlowPkts(5)
scampiFlowOctets(5)
scampiFlowDroppedPkts(6)
scampiFlowStart(8)
scampiFlowStop(9)

scampiMapi(2)

scampiMapiUsers(1) scampiMapiFlows(2)

scampiMapiFunctions(3)
scampiMapiUserFunctions(4)

scampiMeasurements(4)

scampiMesCfgTable(1) scampiMesCfgEntry(1)

scampiMesCfgUID(1)
scampiMesCfgIndex(2)
scampiMesCfgIfIndex(3)
scampiMesCfgIntervalSec(4)
scampiMesCfgMaxLength(6)
scampiMesCfgActive(7)
scampiMesCfgStorageType(8)
scampiMesCfgStatus(9)

scampiMesTable(2) scampiMesEntry(2)

scampiMesUID(1)
scampiMesIndex(2)
scampiMesIntervalId(3)
scampiMesStartSec(4)
scampiMesStartFrac(5)
scampiMesPkts(6)
scampiMesOctets(7)

scampiDevices(1)

scampiDeviceTable(1) scampiDeviceEntry(1)

scampiDeviceIndex(1)
scampiDeviceName(2)
scampiDeviceDescr(3)
scampiDeviceAlias(4)
scampiDeviceIfNum(5)
scampiDeviceGPSSync(6)
scampiDevIfDeviceIndex(2)
scampiDevIfType(3)
scampiDevIfSpeed(4)

scampiFlowCondition(4)
scampiFlowGPSSync(7)
scampiMesCfgIntervalFrac(5)

Figure 5.2: SCAMPI MIB
### 5.3.2.1.2 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.

### 5.3.2.2 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.

### 5.3.2.3 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.
5.3.2.3.1 **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.
- **scampiFlowUID**: The UID of the process that initiated this flow.
- **scampiFlowIfIndex**: The scampiIfIndex of the interface that the flow uses to capture packets.
- **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.
- **scampiFlowGPSSync**: 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.

5.3.2.4 **Measurement group**

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

5.3.2.4.1 **scampiMesCfgTable** This table is used for configuring measurement jobs.

- **scampiMesCfgUID**: The UID of the user who created and controls this measurement.
- **scampiMesCfgIndex**: The index of the measurement. Should be unique for the corresponding scampiMesCfgUID value.
- **scampiMesCfgIfIndex**: Reference to scampiDevIfIndex and specifies which interface that should be used for measurements.
- **scampiMesCfgIntervalSec** and **scampiMesCfgIntervalFrac**: Together form the time interval for measurements. scampiMesCfgIntervalSec specifies the number of whole seconds of the interval
- **scampiMesCfgIntervalFrac**: 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.
- **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.

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

scampiMesUID The UID of the user who created and controls this measurement.

scampiMesIndex The index of the measurement.

scampiMesIntervalId A unique ID for the time interval.

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

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.

5.3.3 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.

scampi@ist-scampi.org 68 November 13th, 2003
5.4 Monitoring in IPv6 networks

MAPI will as far as possible have full IPv6 support. This means that no MAPI functions or internal data structures or variables will put any restriction on supporting IPv6. The only problem is that many of the functions that can be applied to flows are based on external libraries and for these functions to support IPv6 the external libraries must also support it. An example of this is the BPF filter function. This function depends on the libpcap library so for it to have IPv6 support it is a requirement that the libpcap library that is being used has been compiled with IPv6 support. Normally when an application creates a new flow, this flow will contain both IPv4 and IPv6 traffic. To restrict the flow to only one IP version, a filter will have to be applied.

5.4.1 Monitoring DWDM links and other link layer technologies

The physical interface module (transceiver) and the link layer implementation in the firmware of the COMBO6 card will be both replaceable. The physical interface will use the XFP standard for interchangeable transceivers. This will enable to connect the monitoring system to any physical layer link for which transceivers are available. For coarse wave division multiplexing (CWDM) networks, such transceivers are currently available. XFP transceivers for dense wave division multiplexing (DWDM) networks only very recently have started to become available. However, as the monitoring adapter is only receiving packets and optical receivers are normally sensitive in a broad part of spectrum, we can use a CWDM transceiver and filter out other wavelengths with an add-on optical filter. Indeed, there exist stand-alone optical filters that can be used to filter a particular wavelength; the filtered wavelength can subsequently be sent to a COMBO6 card that performs the higher layer traffic monitoring.

Monitoring of multiple wavelengths can be achieved with multiple monitoring systems, each tuned to monitor a specific wavelength. The monitoring systems can be placed in the same or in different physical observation points. The combination of the measurements from multiple monitors depends on the particular application. For some applications such as accounting, typically there will be no need for combining measurements when the traffic to/from a particular users flows over different wavelengths (as identified below, this is not the case when inverse multiplexing is used). On the other hand, applications such as QoS monitoring can potentially gain from considering measurements from different wavelengths, since e.g. a degradation of the quality of more than one wavelengths may provide an indication of a physical layer problem (optical tranceivers, optical fiber, repeaters, etc).

In the case of inverse multiplexing, where two or more wavelengths are combined to form a higher rate pipe, monitoring each wavelength independently is no longer sufficient since traffic from the same flow may traverse different wavelengths. Monitoring in this case will require additional hardware for combining the traffic from multiple wavelengths.

Monitoring networks with a non-Ethernet link layer, such as SDH networks, will require firmware modifications. Firmware can be replaced any time without hardware modification. Within the SCAMPI project, the development of two monitoring adapters is planned: the Phase I adapter for Gigabit Ethernet (1 Gb/s) and the Phase II adapter for 10 Gigabit Ethernet (10 Gb/s). This decision is based on the demands of European NRENs and expected development towards Ethernet-based networks. Firmware modifications for SDH or other link layer technologies can be done after the project based on the monitoring demands.
Chapter 6

MAPI Definition

6.1 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 [25]. 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 200-Gbyte disks which are expected to grow significantly larger in the near future. 1

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.

6.1.1 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:

---

1To 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”.

71
flow_descriptor fd = 
mapi_create_flow(char *device_d)

The flow, defined by the flow descriptor \textit{fd} consists of all packets that arrive in the device \textit{device_d} (which can be either a monitoring device or a trace file)

Once a flow is created, the user may decide to manipulate it by setting some options related to this flow:

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

Network flows come in three modes as defined by option \textit{mode} which can take three values: \textit{raw}, \textit{cooked}, and \textit{hierarchical}. In the raw mode, the flow consists of all network packets that satisfy the condition \textit{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 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 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: \textsuperscript{2}

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

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:

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 \cite{8}. 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.

\textsuperscript{2}More information about network flow options can be found in appendix D.2.
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.

### 6.1.2 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:

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

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\(^3\) invokes the callback handler for each packet that arrives in the network flow `fd`, and for the next `cnt` packets.

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

The callback handler takes two arguments: the first is the network flow that invokes the handler, and the second is the packet itself. It is important to understand that the first argument to the handler must be the network flow. Otherwise, a handler that receives a packet has no way of knowing which flow this packet belongs to.

### 6.1.3 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 `f` to all packets of network flow `fd`.

```c
int function_id = mapi_apply_function(flow_descriptor fd, char * f, ...)
```

---

\(^3\)Although `mapi_loop` was inspired by the `pcap_loop` call from the `libpcap` library, contrary to the `pcap_loop`, `mapi_loop` is non-blocking.

scampi@ist-scampi.org

November 13th, 2003
If the call is successful, it returns a handler to the applies function. This handler can be used to identify the function, to read its results, to remove it, etc.

The SCAMPI monitoring system will provide several predefined functions that that will probably cover most of the network monitoring needs of ordinary users. ¹ For example, there will be a function (PACKET_COUNT) that counts all packets in a flow. Another function (SAMPLE_PACKETS with two arguments sample_period, mode) will sample one out of every sample_period 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 packet_count 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. ² This can be achieved by allocating a small amount of memory, or a data structure to each network flow. The functions that will be applied to 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:

```
void * mapi_read_results(flow fd, int function_id, int copy)
```

The call will read the results of function function_id of network flow fd and will return a pointer (void *) to them. If copy is FALSE, then the pointer will point to a memory region shared between the MAPI daemon and the user application. This will be the exact region where the daemon writes the results of the function. The application may access future results through the same pointer without having to invoke mapi_read_results again.

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_id 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:

```
struct packet_count_results * pr ;
pr = (struct packet_count_results *)
    mapi_read_results(fd, packet_count_function_id, FALSE) ;
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:

```
struct byte_count_results * pr ;
pr = mapi_read_results(fd, byte_count_function_id, FALSE) ;
printf("Number of packets counted for flow %d is %llu \n",
    fd, pr.bytes)
```

If necessary, the user may also remove a function:

```
mapi_remove_function(flow_descriptor fd, int function_id)
```

This call can be used when the function is no longer needed in this network flow.

¹For a current list of these functions have a look at appendix B.
²Functions can be applied in cooked mode as well, after the re-assembly (i.e. cooking) of packets is done.
6.1.4 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, the data type of a packet is 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.

6.1.5 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 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.

6.1.5.1 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 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. Where 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, 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.

6.1.5.1.1 Flow key specification

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

---


7 It is likely that this section will have to be revised when the IPFIX standard is finalized.

scampi@ist-scampi.org  November 13th, 2003
CHAPTER 6. MAPI DEFINITION

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

6.1.5.1.2 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)
6.1. PASSIVE MONITORING MAPI

- 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
- 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.

6.1.5.1.3 Implementation issues

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
CHAPTER 6. MAPI DEFINITION

- 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

**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.

**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:

```plaintext
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.
6.1.5.2 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)  
callback(flow_descriptor fd, sub_flow * p)
```

6.1.6 Loadable Functions

MAPI supports dynamic loadable functions, or more generally, libraries of functions, in the following way: MAPI applications will inform the MAPI daemon `mapid` for a new library with a new function:

```c
mapi_load_library("newlib.so")
```

```verbatim
{tt newlib.so} can be searched out in the same directory, 
or in a default path 
denoted by MAPI variable {tt MAPI\_DLIB\_PATH}. 
The daemon {tt mapid} will then {tt dlopen()} this library. 
Users can from now and on apply functions from the new library as they do 
for predefined functions. 
%, while a slight change is necessary in 
%mapi_apply_function():
%
% mapi_apply_function(fd, "newlib\_funct1", args...)
%
% The second argument must be changed to a string, because mapid should know 
% the names of the functions included in the library to be able to call 
% them. Another workaround could be to have only one function per file, and 
% a function like 
%
% funct_id = mapi_load_funct("funct.so", "functname")
%
% which returns an integer identifying this function. funct_id will then be 
% passed as the second argument of create_flow(), preserving its current 
% definition.
```

\subsection{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.\footnote{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.}

In our first example we are interested in printing the headers of all 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 ;

// create a flow that consists of all packets
// destined to port 80
fd = mapi_create_flow(dev) ;
mapi_set_option(fd, mapi_filter, "dst port 80") ;
mapi_connect(fd) ;
for (; ;) {
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 ;
int fid ;

fd = mapi_create_flow(dev) ;
mapi_set_option(fd, mapi_filter, "dst port 80") ;
fid = mapi_apply_function(fd, PACKET_COUNT) ;
mapi_connect(fd) ;
sleep(10) ; // sleep for 10 seconds
r = mapi_read_results(fd, fid, FALSE) ;
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):

```c
```
char * dev = "/dev/scampi";

flow_descriptor fd;
int function_id;
byte_count_results * r;
int previous_byte_count = 0;

fd = mapi_create_flow(dev);
mapi_set_option(fd, mapi_filter, "ip 139.91.191.150");
function_id = mapi_apply_function(fd, BYTE_COUNT);
mapi_connect(fd);

r = mapi_read_results(fd, function_id, TRUE);
while(1) { //forever
    sleep(1) ; // sleep for one second
    printf("%d bytes/sec
",
          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;
int fin, fout;

in_fd = mapi_create_flow(dev);
out_fd = mapi_create_flow(dev);
mapi_set_option(in_fd, mapi_filter, "dst port 80 and dst ip 139.91.191.150");
mapi_set_option(out_fd, mapi_filter, "src port 80 and src ip 139.91.191.150");
fin = mapi_apply_function(in_fd, BYTE_COUNT);
fout = mapi_apply_function(out_fd, BYTE_COUNT);
mapi_connect(in_fd);
mapi_connect(out_fd);

in_r = mapi_read_results(in_fd, fin, TRUE);
out_r = mapi_read_results(out_fd, fout, TRUE);
while(1) { //forever
    sleep(1) ; // sleep for one second
    printf("%d incoming bytes/sec
",
          in_r.bytes-in_previous_byte_count);
    in_previous_byte_count = in_r.bytes;
}
CHAPTER 6. MAPI DEFINITION

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 [8]. Trajectory sampling, 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 deterministically 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) ;
//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 flow 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.

6.1.7 Some More Complex Monitoring Examples

6.1.7.1 Monitoring FTP traffic

In this Section we present an example of using MAPI to monitor all FTP traffic in a system. The main difficulty with monitoring FTP traffic, as compared to applications like email or web traffic, is that FTP transfers may be performed over dynamically allocated ports, which are not known in advance. FTP uses a well-known port (i.e. 21) only as a control channel. When a file transfer is initiated, the FTP server informs the client about the dynamic port number to be used for the transfer. Therefore, in order to accurately account for all FTP traffic, a monitoring application needs to monitor port 21.
to find new clients as well as the dynamic ports these new clients will use in order to transfer their data. Traditional monitoring systems, such as NetFlow, find it difficult to monitor traffic of applications that use dynamically generated ports. For example, although NetFlow and similar approaches, can report the amount of observed traffic per port, they do not know which applications these (dynamically generated) ports correspond to, and thus it is difficult to attribute network traffic to specific applications. On the contrary, MAPI is able to analyze packet payloads to find the dynamically generated ports and to associate those ports with the application that generated them.

The following code can be used to monitor all FTP traffic using MAPI:

```c
packet *p;
flow_descriptor fd, xfers[1024];
int fid, fid_xfers[1024];
struct byte_count_results br;
int ports[2], count;
unsigned long total_ftp_traffic=0;
char new_flow[64];

/* Create a flow to monitor the control port of FTP: port 21 */
1a: fd = mapi_create_flow("/dev/scampi");
1b: mapi_set_option(fd, mapi_filter, "tcp port 21");

/* Find packets that indicate the beginning of a new transfer */
/* such packets contain the string "227 Entering Passive Mode" */
2: fid = mapi_apply_function(fd, SUBSTRING_SEARCH, "227 Entering Passive Mode");

/* Track the next 100 transfers */
3: for(count=0; count<100; count++){
4: p = mapi_get_next_packet(fd);
5: extract_ports(p, ports);
6: sprintf(new_flow, "tcp src port %d and dest port %d", port[0], port[1]);
7a: xfers[count] = mapi_create_flow("/dev/scampi", new_flow); 7b: mapi_set_option(xfers[count], mapi_filter, new_flow);
8: fid_xfers[count] = mapi_apply_function(xfers[count], BYTE_COUNT);
9: print_ftp_traffic();
}

print_ftp_traffic() {
 /* summary */
10: for(count=0; count<100; count++){
11: br = mapi_read_results(xfers[count], fid_xfers[count], FALSE);
12: total_ftp_traffic += br.bytes;
}   
print("FTP traffic = %d\n", total_ftp_traffic);
}
```

In order to monitor all FTP traffic, we initially define a network flow for capturing all FTP control packets that go through port 21 (line 1a). We are interested only for packets indicating a file transfer initiation, thus substring search is applied to distinguish them among the rest (line 2a). An example payload of such packet is the following:

227 Entering Passive Mode (147,52,17,51,146,226)

scampi@ist-scampi.org 83 November 13th, 2003
CHAPTER 6. MAPI DEFINITION

This packet is sent by the server and contains the IP address (147.52.17.51) and the port number (37602) of the forthcoming transfer. Therefore, all necessary information for the transfer session is known so a new flow can be defined for its accounting.

Peer-to-peer, multimedia conferencing and messaging applications usually operate in the same fashion, negotiating transfer ports through a control channel. It is straightforward to adapt the above code to monitor the traffic of any of these applications.

Although the above example demonstrates that MAPI can provide traffic information that traditional flow-level traffic summaries, such as Netflow, cannot provide, one could have used a packet dumping facility, such as tcpdump or other libpcap-based tools, in order to find similar information. However, implementing the above application using libpcap would have resulted in longer code and higher overheads. For example, libpcap does not provide any string searching facility, and thus the programmer would have to provide a significant chunk of code to substitute line 2: above. In addition, libpcap does not provide any facility to apply functions to packets, and thus the programmer would have to provide a the code to read packets and count their bytes. Instead of forcing the programmer to provide all this mundane code, MAPI already provides this frequently used functionality.

6.1.7.2 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.

6.1.7.3 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 this paragraph we will describe another way of doing it. 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.

6.1.8 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 [17] or Linux Socket Filters [11]. 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 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.

### 6.1.8.1 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.

### 6.1.8.2 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 */
}
```

#### 6.1.8.2.1 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)
6.1.8.3 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.

6.1.9 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.

6.2 MAPI vs. Other Approaches

Current network monitoring tools use a wide variety of languages and environments, including libpcap [16], Berkeley Packet Filters [17], CoralReef [14], Linux Socket Filters [11], IPFIX [23], and NETFLOW [28]. 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)” [24]. The notion of a flow has also been widely used by CISCO in its various network monitoring products including NetFlow [28]. 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 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 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 [17] can filter monitored traffic much like the
conditions of the network flows do. As another example, Linux Socket Filters [11] 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 [26]. 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, let’s 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. network flows a first-class status, MAPI enables the efficient implementation of monitoring applications on top of a wide variety of platforms.

6.3 Applications that use multiple observation points

So far, we only addressed applications that require monitoring information from a single observation point. In these cases the applications run locally on the observation point. The observation point both gathers the required information and does the processing of the captured data.

Advanced applications require not only information from a single observation point, they need monitoring data from the entire network. Therefore, the application needs to obtain information from multiple distributed points in the network. Consider a Quality of Service (QoS) application. This application needs to determine network characteristics such as throughput, loss, jitter, goodput, delay,... These characteristics can only be computed if we have information from at least both the ingress and egress nodes in the network.

The current MAPI supports these kinds of applications. To do so, a distributed application needs to create multiple network flows, at least one for each observation point, by using the “remote” MAPI.

6.3.1 The Remote MAPI

The interfaces of the “remote” MAPI and the “local” MAPI are very similar, if not the same. Based on the use of the interface, the SCAMPI platform will internally behave different. The only remarkable difference is the location of the invocation of certain function-calls. In the case of the “local” MAPI, every function is executed on the local monitoring agent. There is no need to specify the location of the observation point. When the MAPI is used to initiate a monitoring job remotely, the location

---

8From a programmer’s point-of-view, the remote MAPI and the local MAPI are functionally the same.
and identification of the monitoring agent needs to be specified. The application has to specify which observation point should run the monitoring job. Obviously, each participating observation point has to run the MAPI daemon.

An application that needs to monitor a remote observation point, needs to use a MAPI with remote functionality. Therefore we extended the MAPI library to allow remote executions. In this case, the `mapi_create_flow` function has to be extended with a location/identity parameter to locate or identify the observation point. From an implementation point-of-view, instead of inter-process communication (IPC) to communicate with the MAPI daemon, the remote MAPI uses remote procedure calls (RPC).

When the observation point is located on a different machine than the client application, the application can specify the location in the `mapi_create_flow` function call:

```c
mapi_create_flow(device);
```

In the case of a remote MAPI, the device will specify both the interface the flow should be captured from, and the IP address of the host the observation point is running on. The format of “device” is:

```
Device = ‘’Interface@IPaddress’’ e.g. ‘’eth1@10.10.10.1’’
```

When no IP address is specified, i.e. device = “interface”, the MAPI assumes the observation point is running on the local host, i.e. we use the local MAPI. When on the other hand interface = “DEFAULT”, the application captures the flow from its default interface (in case the observation point has for example only one interface).

### 6.3.2 Creating multiple network flows

A second step in writing a distributed monitoring application is to open multiple network flows, at least one on each observation point. After retrieving monitoring results from these flows, the application can process and correlate them. The processing and correlation of the data obtained from the distributed sites is the responsibility of the applications. The MAPI only offers the programmer the opportunity to easily obtain distributed data.

The following example illustrates a basic application that requires 2 observation points.

```c
int fd1, fd2;
int counter1, counter2;
unsigned long long result1, result2;
fd1 = mapi_create_flow("eth1@10.0.0.1");
mapi_apply_function(fd1, BPF_FILTER, "SRC 10.0.0.1 and DST 10.0.0.2");
counter1 = mapi_apply_function(fd1, PKT_COUNTER);
fd2 = mapi_create_flow("/dev/dag2@10.0.0.2");
mapi_apply_function(fd2, BPF_FILTER, "SRC 10.0.0.1 and DST 10.0.0.2");
counter2 = mapi_apply_function(fd2, PKT_COUNTER);
mapi_connect(fd1);
mapi_connect(fd2);
sleep(60);
result1 = mapi_read_results(fd1, counter1,FALSE);
result2 = mapi_read_results(fd2, counter2,FALSE);
printf("There were %lld packets dropped\n., counter1 . counter2);
mapi_close_flow(fd1);
mapi_close_flow(fd2);
```

scampi@ist-scampi.org 88 November 13th, 2003
In this application 2 flows are configured, one on the observation point with IP address 10.0.0.1 (the ingress) and one on the observation point with IP address 10.0.0.2 (the egress). Both the Flows count all the packets with source address the first observation point and destinations address the second observation point. After one minute, the application prints the number of dropped packets sent from observation point 1 to 2. This very basic example illustrates how an application can use both the remote MAPI and the creation of multiple flows in order to monitor distributes sites. After the configuration of both flows, the application only needs to correlate and process the retrieved data.
Chapter 7

Summary

This document presented a detailed description of the SCAMPI architecture - both software and hardware. We believe that we have introduced a novel architecture that can be efficiently implemented in high-speed networks. Capitalizing on powerful abstractions, such as the first-class network flows, and on expressive ability, such as that provided by function application to flows, the SCAMPI architecture is in a unique position to provide a solution for high-speed network monitoring.

The impact of SCAMPI on efficiency and ease of programming is expected to increase as networks get faster, monitoring applications become more complex, and hardware-supported monitoring becomes more prevalent. We expect that SCAMPI will provide an effective platform which enables applications to express their needs and allows the underlying monitoring system to optimize the implementation in the best possible way.
Appendix A

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 [17]. It is interesting to note that the same language is being used in tcpdump [30], in Linux Socket Filters [11], and in libpcap [16].

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:

```
net 128.32
```
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 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 [26]), or they may want to perform sophisticated sampling, like in trajectory sampling [8]. 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 B

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:

**PKT_COUNTER**

structure of results:

```
struct packet_count_results {
    uint64_t packets ;
};
```

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

**STR_SEARCH(unsigned char *s, int len, int offset, int depth)**

structure of results: none

It searches the body of the network packet to see if it contains string `s` of length `len` \(^1\). The search starts from byte `offset` and lasts for `depth` bytes. If substring `s` is not found, the packet is discarded.

**BYTES_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 octets observed in a time interval. The length of the interval in microseconds is the input parameter `us`. The start of the interval is return in the results structure, along with the number of the packets observed.

\(^1\)Note that the string `s` may contain the character `0` as one of its characters.
APPENDIX B. PREDEFINED FUNCTIONS FOR NETWORK FLOWS

DISCARD

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

    It discards the current packet. It is being used when users are not really interested in receiving packets, but are only interested in calculating 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 [22].

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.

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 [8], 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 [8].

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 \texttt{endbyte} is 0, the function saves up to the end of the packet.
Appendix C

Functions and attributes for admission control

C.1 Attributes for assertion and assertion examples

Function type assertions:
- function is defined: (function_name) == "defined"
- function instances: (function_name).num == (number_of_instances)
- first position of an instance:
  (function_name).first == (first_position_of_function)
- last position of an instance:
  (function_name).last == (last_position_of_function)
- maximum value of a parameter (if a string, the length):
  (function_name).param.(parameter_no).max == (maximum_value)
- minimum value of a parameter (if a string, the length):
  (function_name).param.(parameter_no).min == (minimum value)

Function instances assertions
- instance position:
  (function_name).(instance_no).pos == (function_position)
  func.(function_position)_name == "(function_name)"
  e.g. PKT_COUNTER.0.pos == 0 , func.0.name == "PKT_COUNTER"
- instance parameters:
  (function_name).(instance_no).param.(parameter_no) == (parameter_value)
  func.(function_position).param.(parameter_no) == (parameter_value)
  e.g. BPF_FILTER.0.param.0 == "port 80" , func.0.param.0 == "port 80"

C.1.1 The scampi_auth datatype

All the information needed for admission control(authd).

unsigned char pubkey [MAX_PUBKEY_SIZE]
Public key.
char credentials [MAX_CREDENTIALS_SIZE]
Credentials.
unsigned int nonce
Number provided by mapid as a challenge to authenticate user (flow id)
char encrypted_nonce [MAX_ENC_NONCE_SIZE]
Encrypted nonce with private key.
APPENDIX C. FUNCTIONS AND ATTRIBUTES FOR ADMISSION CONTROL

unsigned int encrypted_nonce_len
The length of the encrypted nonce.

char device_name [MAX_DEVICE_NAME_SIZE]
The name of the device associated with this flow.

char app_domain [MAX_APP_DOMAIN_SIZE]
The application domain of this request. Should be "MAPI".

unsigned int functions_num
The number of the serialized functions in the buffer.

unsigned char function_list [MAX_FUNCTION_LIST_SIZE]
The buffer with the serialized function list.

C.1.2 The flow_auth datatype
Results returned by admission control(authd).

int PCV
Asserted if the flow is authorized

mapi_resources required_resources;
Amount of required resources.

C.1.3 The scampi_resources datatype
Resources can be of various types. Currently the following two are defined

unsigned int processing;
Processing time.

unsigned int memory;
Memory consumption.

C.1.4 mapid writing and reading the shared memory

int mapid_shm_store(void *seg,int flow_id)
Parameters:
seg the shared memory segment
flow_id the flow id to get data from
Returns: 0 on success, or -1 on failure

int mapid_shm_retrieve(void *seg,flow_auth *f)
Parameters:
seg the shared memory segment
f pointer to variable to copy data into
Returns: 0 on success, or -1 on failure
Appendix D

Man pages for MAPI functions

D.1 MAPI functions

flow_descriptor fd =
    mapi_create_flow(char *device_d)

create_flow is used for creating a new network flow of packets. The flow of packets
will be read from device device_d. If unsuccessful, it returns a negative number.

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.

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.

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.

void * mapi_get_flow_option(flow_descriptor fd, int option)

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.
APPENDIX D. MAN PAGES FOR MAPI FUNCTIONS

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

If users do not want to block in the process of receiving network packets from a flow using the `tt 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`.

```c
int function_id = mapi_apply_function(flow_descriptor fd, char* f,...)
```

`mapi_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.

```c
int mapi_load_library(char * libname)
```

`mapi_load_library` informs MAPI that the user is interested in using functions defined in library `libname`. The library is searched in the current directory as well as in the path defined in variable `MAPI_DLIB_PATH`. If everything is successful, the function returns a non-negative integer. Otherwise, it returns a negative one.

```c
int mapi_remove_function(flow_descriptor fd, int function_id,...)
```

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

```c
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.

```c
void * mapi_read_results(flow_descriptor fd, int function_id, int copy)
```

`mapi_read_results` will receive statistics or any kind of results that have been computed by the application of function `function_id` in the packets of flow `fd`. The results will be returned in a structure pointed to by the return value of the function. If `copy` is FALSE, then the pointer points to a memory region shared between the `mapid` and the user application. This region is exactly the place where the MAPI daemon writes the results of the function. The application may continually access the updated results through the pointer to this shared region without re-invoking the `mapi_read_results` function. This shared region can also be used to efficiently transfer large amount of results from the daemon to the application. If `copy` is TRUE the results are copied in a separate memory region that will not be continually updated by the MAPI daemon.

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 `{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:
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.

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.

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.

int mapi_settime (struct timespec *ts)

mapi_settime sets the adapter clock to the given value.

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.

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)

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)

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.
APPENDIX D. MAN PAGES FOR MAPI FUNCTIONS

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.

mapi_mode

Mode mode defines the type of the flow: RAW, COOKED, and HIERARCHICAL. If successful, the function returns a non-negative integer flow descriptor fd. The default type of a newly-created flow is RAW.

(char *) mapi_filter

All packets of the flow that do not match filter filter are discarded. filter is expressed in the language used by tcpdump. For example ‘host arion’ matches all packets arriving to or departing from host arion. Similarly ‘dst port 80’ matches all packets which are destined to port 80.

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 ith bit is set, the flow receives packets from the ith 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:

1Recall that these options may be modified with the set_flow_option(flow descriptor, fd, int option, int value) function.
int if1 = 1 ; /* 0...0001 */
int if2 = 2 ; /* 0...0010 */

packet * p1, p2 ;

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.

filter

It returns a string containing the filter 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 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
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 a 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 cannot 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;
  }
  ```
D.2. MAPI VARIABLES

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:

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

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 [10]. A “packet” in this format is actually a sequence if packet records headed by a flow record 2 as described below:

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 ;
}

---

2 We should make clear that a flow record as defined in [10] is different from a flow record returned by function read_flow_record. The first contains some packet header fields as defined in [10], while the latter contains statistics (like packet count, byte count, etc.) about a flow.
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.

MAPI_DLIB_PATH

MAPI_DLIB_PATH contains the path of the directories where libraries of functions can be loaded using the mapi_load_library call.

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.

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.
D.3. VARIABLES FOR HIERARCHICAL FLOWS

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

Header Files

E.1 mapi.h

 ifndef _MAPI_H
 define _MAPI_H 1

 include <sys/ipc.h>

 define PAPI 10

 include "mapi_errors.h"

 ifndef TRUE
 define TRUE 1
 endif

 ifndef FALSE
 define FALSE 0
 endif

 /* NOTE: When more functions are added, also update the array in
 adm_ctrl/scampi_auth.c with the name of the function */
typedef enum {PKT_COUNTER, /* Count the packet */
 STR_SEARCH, /* Search the specified string */
 BPF_FILTER, /* Filter the packet */
 TO_BUFFER, /* Copies packets to a buffer that can be
 read by user applications */
 ETHEREAL, /* ethereal filter, uses the ethereal display filter to filter pkts */
 TO_TCPDUMP, /* Create tcpdump file format from captured packages */
 BYTE_COUNTER /* Count the bytes passing through the filter */
} mapiFunction;

typedef unsigned char mapiFunctArg;

 /* Replaced by the following enumeration
 * Also used by admission control
 * Handy since mapi_parameter_type can now be used to type check at compile time
 #define INT 1
 #define STRING 2
 #define UNSIGNED_LONG_LONG 3
 */
APPENDIX E. HEADER FILES

typedef enum { INT = 1, STRING, UNSIGNED_LONG_LONG } mapi_parameter_type;

/*Structure that contains device independant information about packets*/
struct mapipkt {
    unsigned long long ts; /* NTP 64-bit timestamp of packet as defined in
RFC 1305*/
    unsigned caplen; /* Number of bytes from the packet that were captured*/
    unsigned wlen; /* Wire length. Real length of packet as seen on network*/
    unsigned char pkt; /* Pointer to the IP packet */
};

enum trace_formats {
    TCPDUMP_TRACE
};

//Create new mapi flow
int mapi_create_flow(char *dev);

//Create new mapi flow based on a trace file
int mapi_create_offline_flow(char *path, short format);

//Apply function to a flow
int mapi_apply_function(int fd, mapiFunction funct, ...);

//Connect to a mapi flow
int mapi_connect(int fd);

//Get the next packet from a to_buffer function
struct mapipkt *mapi_get_next_pkt(int fd, int fid);

//Read result from a function
void* mapi_read_results(int fd, int fid, int copy);

//Close a mapi flow
int mapi_close_flow(int fd);

//Read the last error-code set by mapid or mapi-api
//err_no and errorstr should be allocated, the function won’t allocate memory
//errorstr is always < 512 bytes
int mapi_read_error(int* err_no, char* errorstr);

#endif

E.2 mapidlib.h

#ifndef _MAPIDLIB_H
#define _MAPIDLIB_H 1

#include "mapid.h"
#include "mapidrv.h"
#include <pcap.h>
#include <net/bpf.h>
//structure for internal use by the shared library
struct ethereal_ses
{
    int link_type;
    void* dfilter;
};

//Structure used by the ethereal filter to store internal data
struct ethereal_data
{
    struct ethereal_ses session;
    unsigned cap_length;
    void* dll;
};

struct to_tcpdump_data
{
    char* run;
    char* buffer;
    int buflen;
    unsigned bufpos;
    int filed;
    char* filename;
    char* tmpfilename;
    unsigned long long cnt;
    unsigned long long mpkts;
};

//Structure used by the to_buffer function
struct mapid_to_buffer
{
    key_t buf_key; //Shared memory key for the buffer
    unsigned long read_ptr; //Pointer to the next packet that can be read
    unsigned long write_ptr; //Pointer to where the next packet can be written
    char* buf; //Pointer to buffer
    int cap_length; //Maximum size of a captured packet
    unsigned bufsize; //Size of buffer
    key_t sem_key; //key of the semaphore used for blocking the clients app.
    int semaphore; //the semaphore ID
};

struct mapid_pkthdr
{
    unsigned long long ts; /* NTP 64-bit timestamp of packet as defined in
RFC 1305*/
    unsigned caplen; /* length of portion present */
    unsigned wlen; /* length this packet (off wire) */
};

//General information about the hardware adapter that is being used
//and that various functions might find useful
struct mapid_hw_info
{
    unsigned int link_type; // Data-link level type as defined in bpf.h
    unsigned int cap_length; // Maximum packet capture length
};
APPENDIX E. HEADER FILES

//Stores information about functions applied to flows
struct mapid_function {
    mapiFunction function; //Type of function
    char (*functionptr)(struct mapid_function* funct,
                        const unsigned char* pkt,
                        const struct mapid_pkthdr* pkthdr); //Pointer to function
    int fid; //Function ID
    key_t key; //Shared memory key
    int size; //Size of shared memory
    void* data; //Pointer to shared memory
    unsigned char *serialized_arguments; //Serialized arguments. Used by admission control
    struct mapid_function *next;
};

struct mapid_bpf_filter {
    char *expression; // BPF filter expression
    struct bpf_program compiled;
    int currentIteration; // Optimization for packet in this iteration
    char result; // result of evaluation
};

struct mapid_strsearch {
    unsigned char *str; /* string to search for in payload (without \0)*/
    int slen; /* length of string */
    int offset; /* Starting search position from the beginning of packet */
    int depth; /* Maximum search depth from the beginning of search position */
    int *shift; /* Boyer-Moore Shift table */
    int *skip; /* Boyer-Moore Skip table */
    int currentIteration; // Optimization for packet in this iteration
    char result; // result of evaluation
};

int mapid_connect(int fd);
int mapid_add_flow(int fd);
int mapid_close_flow(int fd);
int mapid_read_results(int fd,
                        int fid,
                        struct mapid_result *result);
int mapid_apply_function(int fd,
                          mapiFunction function,
                          void* fptr,
                          mapiFunctArg *fargs);
void mapid_process_pkt(const unsigned char* pkt,
                       struct mapid_pkthdr* pkthdr);
void mapid_set_hw_info(struct mapid_hw_info* info);
int mapid_get_errno(int fid);
int mapid_serialize_functions(int flow_id,long max,unsigned char *buf);
#endif
E.3  libscampi.h

/*
 * libscampi.h: MAPI <--> SCAMPI library interface
 * Author: Sven Ubik <ubik@cesnet.cz>
 * Applications share a common circular packet buffer and have their own FIFOs
 * of packet pointers in library/driver (phase I) or in adapter (phase II).
 * Filters, samplers and statistics counters are implemented in library/driver
 * (phase I) or in adapter (phase II).
 */

#ifndef _LIB_SCAMPI_H
#define _LIB_SCAMPI_H

/* #include "types.h" */
#include "scampi_kernel.h"

#define SCAMPI_DEVICE "/dev/scampi/0"
#define SCAMPI_SYSLOG_FACILITY LOG_LOCAL0

/* Type of packet descriptor */
typedef struct scampi_mmap_packet_descriptor tPacketDesc;

/*
 * Opens SCAMPI device
 *
 * Input:
 *   name - device filename, such as "/dev/scampi/0"
 *
 * Return value:
 *   >=0 - file descriptor
 *   -1 - error occurred (see syslog)
 */
int scampiOpen(char *name);

/*
 * Closes SCAMPI device
 *
 * Input:
 *   fd - file descriptor
 *
 * Return value:
 *   0 - ok
 *   -1 - error occurred (see syslog)
 */
int scampiClose(int fd);

/*
 * Options
 */
#define SCAMPI_LIB_FILTER 0
#define SCAMPI_LIB_SAMPLER 1
#define SCAMPI_LIB_STATMAP 2
#define SCAMPI_LIB_STATISTICS 3
APPENDIX E. HEADER FILES

#define SCAMPI_LIB_FUNCTIONALITY 4

/*
 * Sets option (Note: options are not yet implemented)
 *
 * Input:
 * fd     - file descriptor
 * option - name of option to be set
 * value  - value of option to be set
 *
 * Return value:
 * 0 = ok
 * -1 = error occurred (see syslog)
 */
int scampiSetOption(int fd, int option, void * value);

/*
 * Gets option
 *
 * Input:
 * fd     - file descriptor
 * option - name of option to be get
 *
 * Output:
 * value  - value of requested option
 *
 * Return value:
 * 0 = ok
 * -1 = error occurred (see syslog)
 */
int scampiGetOption(int fd, int option, void * value);

/*
 * Returns pointer to one new packet and locks this packet in the ring buffer
 *
 * Input:
 * fd - file descriptor
 *
 * Output:
 * size - size of new packet in bytes including Ethernet header
 *
 * Return value:
 * !=NULL - pointer at the first byte of Ethernet header of new packet
 * NULL  = error occurred (see syslog)
 */
unsigned char * scampiGetNextPacket(int fd, int * size);

/*
 * Returns pointer to specified number of new packets and locks these packets
 * in the circular buffer - not yet implemented.
 */
unsigned char * scampiGetNextOffset(int fd);

/*
 * Unlocks any packets locked on behalf of this application
 *
 * Input:
* fd - file descriptor
*/
void scampiUnlockPackets(int fd);
#endif /* _LIB_SCAMPI_H */
Appendix F

FFPF

The design for MAPI for IXP1200 is based on the design of FFPF (fairly fast packet filter) developed at the University Leiden. The original FFPF design consists of a Linux kernel module that is responsible for receiving packets and passing them to userspace without copying. To this end, FFPF employs a single, large, shared packet buffer in which all packets are stored that are of interest to at least one of the applications in a group. A group is simply a collection of applications with the same access right to packets and while there may be more than one group for security reasons, we will ignore this aspect of FFPF for now, and assume that all applications have the same rights to access the packets and, hence, that there is only a single group. FFPF maps the packet buffer to the address space of all the applications in the group, so that (in principle) each application can access all the packets in this buffer.

Of course, not all applications will be interested in all the packets, so for every MAPI-like flow, FFPF maps another circular buffer to the address space of each application (using FFPF kernel code that implements the `mmap` operation) and this buffer contains pointers into the shared packet buffer. Unlike the packet buffer, this buffer, known as the index buffer is not shared. Instead, each application and indeed each flow has its own index buffer that contains the pointer (indices) that are of interest to this flow only.

The FFPF kernel module fills the shared packet buffer with packets and enters the appropriate indices in the index buffers of flows that have expressed interest in this packet. For this purpose, FFPF allows applications to store flow-specific expressions in the kernel that describe which packets are of interest to the flow and even what should be done with the packet.

For each flow, there is a third (non-circular) buffer that is shared between kernel and application and this is used as a generic information buffer which can be used by the application and the expression (really a simple program) operating in the kernel on its behalf, as they see fit. For example, the expression in the kernel can calculate a hash of each `<ip_src, ip_dest, tcp_srcport, tcp_destport>` tuple (identifying a TCP/IP flow) and use the hash as index in an array. Whenever a packet with that hash value arrives, the FFPF expression may then increment a counter in this array, effectively counting all packets received for all flows.

In fact, in FFPF there are ways to express very complex filters, keep counters, calculate hashes, find the most active TCP flows, search for strings, perform classification, etc., but the details of the FFPF expression language are beyond the scope of this document. Flow expressions can be loaded and changed at runtime.

Other features offered by FFPF that may be of interest to a MAPI implementations include the ability for an application to specify that it should be blocked until `n` packets have been received. It is also possible to specify that it should not be blocked, but that, instead, a callback function should be
called whenever \( n \) packets have been received.

The circular buffers in FFPF are controlled by read and write (R and W) pointers. Each circular buffer has an R an W pair that indicate the current read and write positions. Whenever W catches up with R (the buffer is full), new packets are dropped and a droppedPkt counter is incremented until R is updated again (i.e., new slots become available). For the shared packet buffer, this means that the rate at which packets can be processed is determined by the slowest reader. In other words, the R of this buffer is the slowest R of all flows. For all circular buffers holds that W can only be written by the kernel module. For all index buffers holds that the application may update R directly (but only by incrementing it). So, given a current value of R, an application may read as many packets as it can, e.g. 1000, and then perform a single update of R (increasing it by a 10'000). This way, an application reads 1000 packets in a single access, without having to context switch for every packet. On an update of some index buffer R, the kernel module will determine whether or not the value of R for the shared packet buffer should also be updated and if so, it will do so.

Furthermore, FFPF follows the MAPI model of creating and closing flows. First a create_flow request is done. This is followed by a number of functions that should be applied to the flow (e.g. filter, sample, search string, etc.). Next, a request equivalent to mapi_connect_flow is made by the application (it is called instantiate_flow in FFPF). At this point, the flow specification (i.e. the flow in combination with all functions that should be applied to it) is checked to see whether it complies with the security policy, and if so the flow is instantiated. Simply calling close on a flow will destroy all the state that was created for it.

The admission and resource control scheme that was just alluded to and that is currently in use in SCAMPI was adapted from a similar design in FFPF. Very few changes are necessary at the architecture/interface level. It is discussed in more detail in Section XXX. Most of the work involved in porting it to MAPI is due to the fact that the modules no longer have a trusted kernel through which they communicate. Instead, the communication between the various SCAMPI components in the final design is direct, via IPC mechanisms. However, the important thing to note is that FFPF offers an equivalent resource/admission control mechanism and hence is well-suited to serve a platform for implementing the MAPI.

In summary, FFPF offers functionality that is a little richer than what MAPI offers, and a simple implementation of MAPI on top of FFPF has been realised. On the other hand, FFPF operates in the kernel of the operating system and not in user space, like mapid. Also, its language is not like any existing filtering or classification language. However, because of its capability to call (previously registered) external functions and services from within the FFPF language, FFPF is a very convenient platform on which to implement the MAPI. And since there is an implementation of MAPI on FFPF already, it suffices to implement FFPF on IXPs to demonstrate that MAPI can be supported by such network processors. In the next section, we describe the design of FFPF on the IXP1200.

### F.1 FFPF on the IXP1200

The design of FFPF on IXPs, is also based on a similar software framework used in previous work in Leiden on providing software isolation in network processors [4].

#### F.1.1 Basics

A single microengine is responsible for receiving packets from the network ports and storing them in a large circular buffer. Should a single microengine be insufficient, more than one can be dedicated
to this task. All remaining microengines execute application code. In the case of FFPF/MAPI, these microengines correspond to high-speed MAPI flows at a higher level.

On each of the microengines a main loop is provided by the application framework. Given the appropriate privileges (determined by admission control, described elsewhere in this document), programmers may ‘plug in’ application code such as compiled FFPF expressions in this loop and load the complete program on the microengine. This means that the FFPF expression on microengines differ from those in the kernel (at least in the current implementation of FFPF in the kernel) in that they consist of fully compiled native code, rather than interpreted FFPF code. The reason why this is fairly straightforward on the IXP is that the granularity of an FFPF expression corresponding to a flow is a microengine, which means that a microengine is fully dedicated to a single flow. In other words, we can stop, reload and restart a microengine without interfering with any other flow. The steps involved in setting up a flow with a certain FFPF expression is now as follows:

1. Write the expression (program) in FFPF language;
2. compile the program with the FFPF compiler;
3. statically link the code with the main-event loop of the application framework;
4. stop the current program running on the relevant microengine;
5. load the new code;
6. restart the microengine.

Currently, the application framework is capable of offering explicit support for dealing with with tardy application. If this is used, the packet fetch microengine dedicates a single thread (known as ‘mop-up’) to make sure slow applications do not slow down fast applications. Whenever a microengine is falling behind too much (it has not processed enough packets in a given amount of time), the mop-up terminates it. We consider this to be the network processor equivalent of timeouts on a shared processor. The microengine only posts a kill request, the actual termination is done by the StrongARM. The packet fetch that we implemented places packets in a circular buffer spread over SRAM and SDRAM. The actual packets are stored in SDRAM, while buffer control structures are kept in SRAM. For example, a bit field per microengine per buffered packet is used to indicate whether a microengine is ‘done’ with this packet. Applications may choose to process every packet or only a percentage of the packets. However, for all packets in the buffer they have to set their ‘done’ flag in time. In the same structure we also keep bit fields to implement readers/writers locks (with readers’ preference) and fields indicating whether the packet has been fully received.

Without the mop-up, the application framework turns into a more conventional system, where the write threads write data in the circular buffer until they reach a slot that is not empty and which is still in use by certain flows. In this case, the new packet that should have been written in this slot is dropped and a counter called droppedPkts is incremented. Both types of behaviour have advantages and disadvantages. By default, FFPF on IXPs will use the latter one, as this corresponds to the behaviour of FFPF and MAPI in the kernel.

Packets are written in 64 byte chunks and applications do not have to wait for the packets to be received in their entirety: as soon as the first chunk of bytes has arrived, they may start processing. The point is that in the presence of a mop-up thread, they need to process the packets within the cycle budget.
APPENDIX F. FFPF

The mop-up thread, if present, at some distance from the writing position, explicitly removes packets from the circular buffer. If the mop-up finds that, for any microengine, the ‘done’ bit is not set, it means the corresponding application has not completed the processing of this packet. In other words, the application is too slow and will be terminated. By varying the distance between mop-up and receive, we may limit or extend the cycle budget for sets of packets available to applications.

We extended the default Intel device driver to provide memory mapping of all the IXP’s SDRAM all the way to the host applications running in userspace. Microengines may pass a reference to a packet to a queue destined for host applications, which prompts these applications to process the packet further on the host (accessing the required data from across the PCI bus).

### F.1.2 Design details

The receiver threads enqueue the packet they receive in a single large packet buffer in the SDRAM on the board that is shared by all flows. The application code running on the other microengines processes these packets one by one and if the packet is classified as interesting, places a descriptor for this packet in a flow-specific buffer in SRAM (similar to the index buffers in the kernel). The main difference at this level is that all packets are placed in the shared buffer, rather than just the packets that are of interest to at least some application. In essence, this is the way true zero-copy behaviour is implemented.

For simplicity and to reduce PCI communication as much as possible, the applications are run on the StrangARM processor, present on the network processor. In other words, the entire FFPF framework is ported to the StrongARM running Linux on the IXP1200 itself. In the port, the packet reception and buffer handling is replaced by the one that was described in the previous paragraph. In case the capacity is not enough, it is relatively straightforward to do exactly the same in the kernel (except that the latency in communication will be longer because of the PCI bus). The memory mapping all the way to the address space of applications on the host processor has already been realised, so we see few difficulties in this way of operating.

The StrongARM is also responsible for typical control tasks. For example, it performs the initialisation of various components of the IXP, e.g. the FBI (the bus interface responsible for the actual network ports). Moreover, it handles the starting and stopping of microengines, and the loading of new images on a specific microengine. The most important task for the StrongARM is captured in a kernel module `ffpf.o` that initialises the flows and connects flows on microengines to user applications. This involves for instance, managing the buffer space for the index buffers, so that the microengines write in these buffers and applications are given a pointer to these buffers, so they may access them. It also provides the application with a handle on the shared packet buffer.

### F.1.3 FFPF API and MAPI details

Given the implementation of FFPF on the IXP1200, the MAPI is a straightforward port of the code that already exists for translating MAPI calls to their FFPF counterparts. As there are no important dependencies regarding external libraries, this may amount to as little as recompilation of the existing code for the StrongARM. As in the section about DAG cards, we will now describe the API provided by FFPF. In FFPF, a ‘virtual’ flow is first created, before it an ‘actual’ flow is instantiated. In other words, a flow is described in terms of the FFPF expression to associate with it, the callback, etc., and after all of this is constructed, a user may try to instantiate it. It is at this instance that the flow specifics will be checked against security policies.

**Flow specification.** The following type is used to specify in detail the flow that will be created.
typedef struct flow_spec {
  int fd; // flow descriptor
  char devname[256]; // device to which this flow should attach
  ffpf_expression_t expr; // filters, classifiers, functions, etc.
  int waitfor; // if > 0 -> block until this flow received n pkts
  int signal; // signal to be used on callback
  int signalafter; // if > 0 -> send signal after n pkts
  int signalrepeat; // how long should the callback remain active
  void (*callback)(ffpf_callback_thrd_rec_t *trec); // actual callback function
} flow_spec_t;

Several (trivial) functions are provided for filling this structure. After the structure is filled according to the desires of the user, it is submitted for instantiation. For this to proceed, it is first checked to see whether it complies with the security policy (i.e. whether the application has the appropriate privileges).

In essence, the ffpf expression consists of a program string that is compiled to a simple object code. In this expression, users may program the functions they wish to apply to the packets in the flow (including ‘external’ functions). The program is compiled to binary form using the function:

```
int ffpf_filter_encode (char *inexpr, char *outexpr); // returns size of binary
```

**Instantiate a flow.** The following operation will open the FFPF/IXP device file and create the three FFPF buffers (memory mapped to the address space in the application). The flow is created according to the flow specification (e.g. concerning callbacks, filters, functions applied, etc.). If the flow already existed (i.e. `flow->fd != 0`), an existing flow is updated. The operation returns a flow descriptor that is used in all subsequent calls referring to this flow.

```
int instantiate_flow (flow_spec_t *flow, cbuf_t **pktbuf, cbuf_t **indexbuf,
                      unsigned long **memarray)
```

If the flow instantiation was successful, i.e. the request was accepted by the admission control, and all state for it was created in both kernel and user space, the flow exists, but is not active yet. Inactive flows do not receive any packets. A flow has to be activated explicitly. Likewise, it can be paused explicitly.

**Activating/pausing a flow.** The following operation closes the flow. Returns 1 on success, 0 on failure.

```
enum ffpf_flow_status { FFPF_PAUSED_FLOW, FFPF_ACTIVE_FLOW};
int ffpf_set_flow_status (int fd, enum ffpf_flow_status status);
```

**Close a flow.** The following operation closes the flow. Returns 1 on success, 0 on failure.

```
int ffpf_close_flow (int fd);
```

**Advancing the read position.** The following operation advances the read position for a specific flow. The packet buffer in FFPF is a circular buffer with fixed-size slots (big enough to hold the largest packets). So R can be advanced by incrementing by a ‘number of slots’.

```
int ffpf_advance_read (int fd, int num_buf_slots);
```
Get next packet. There is no direct equivalent for the `get_next_packet()` operation in MAPI. In FFPF, a flow is given its (circular) buffers and pointers to the packets that correspond to the flow. Instead of `get_next_packet()`, a function is provided that returns the number of ‘readable slots’ in the packet buffer, e.g. the difference between W and R for this flow. These values are also directly accessible to the application code. Reading a set of packets can thus be implemented, e.g. as follows (assuming `MyIndexBuf` contains the index buffer, `PktBuf` contains the packet buffer and there exists a function `print_packet`):

```c
int num = cbuf_readspace (MyIndexBuf); // returns amount of space
for (i = 0; i < num; i++)
    print_packet (PktBuf->buf[(PktBuf->R+i) % PktBuf->num_elem]);
ffpf_advance_read (fd, num);
```

Callbacks. The callback works by means of a signal from the kernel. Assume that a user application wants to register a callback. It first defines a callback function, e.g.:

```c
/* just a trivial flow-specific callback function */
void example_callback (ffpf_callback_thrd_rec_t *trec)
{
    printf ("call back called for flow %d!\n", trec->rec.fd);
    for (i = 0; i < trec->rec.num; i++)
        print_packet (trec->pktbuf->buf[(trec->pktbuf->R+i) % trec->pktbuf->num_elem]);
    ioctl_advance_read (trec->rec.fd, trec->rec.num);
}
```

The parameter `trec` contains all the information necessary for identifying the flow and all its state in userspace. The callback is installed by setting the appropriate fields in the flow specification and then instantiating the flow, e.g.:

```c
chuf_t *cb, *fifo;
unsigned long *mem;
flow_spec_t *flow;
flow->signal = SIGUSR1; // currently only SIGUSR1 and SIGUSR2 allowed
flow->signalafter = 100;
flow->signalrepeat= 10;
flow->callback = example_callback;
flow.fd = instantiate_flow (&flow, &cb, &fifo, &mem);
```

If admission control accepts the instantiation a new flow has been created (or an existing flow updated, if `flow.fd` already existed) with a callback that is called after 100 packets have been received. The callback remains active for 10 callbacks (i.e. until a 1000 packets have been received in total). On a callback, the callback function is called, which prints the packets and advances the read position by 100.
Appendix G

SNMP access

G.1 SCAMPI MIB

SCAMPI-MIB DEFINITIONS ::= BEGIN

IMPORTS
    MODULE-IDENTITY, OBJECT-TYPE, Counter32, Counter64, Gauge32, enterprises
    FROM SNMPv2-SMI

    DisplayString, TimeStamp
    FROM SNMPv2-TC;

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 }

scampiIfTable OBJECT-TYPE
    SYNTAX  SEQUENCE OF ScampiIfEntry
    MAX-ACCESS not-accessible
APPENDIX G. SNMP ACCESS

STATUS current
DESCRIPTION "Information about each installed SCAMPI adapter."
::= { scampiMIB 1 }

scampiIfEntry OBJECT-TYPE
SYNTAX ScampiIfEntry
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION "An entry in this table provides information about a SCAMPI adapter"
INDEX { scampiIfIndex }
::={ scampiIfTable 1 }

ScampiIfEntry ::= SEQUENCE
{
  scampiIfIndex Integer32,
  scampiIfDescr DisplayString,
  scampiIfDevice DisplayString,
  scampiIfAlias DisplayString,
  scampiIfGPSSync TruthValue,
  scampiIfBusy TruthValue,
  scampiIfPkts Counter64,
  scampiIfOctets Counter64,
  scampiIfDroppedPkts Counter64,
  scampiIfCounterDiscontinuityTime TimeStamp
}

scampiIfIndex 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."
::={ scampiIfEntry 1 }

scampiIfDescr OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..255))
MAX-ACCESS read-only
STATUS current
DESCRIPTION "A textual string containing information about the interface. The string should include the name of the manufacturer, the product name and the version of the interface hardware/software."
::={ scampiIfEntry 2 }

scampiIfDevice OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..63))
MAX-ACCESS read-only
STATUS current
DESCRIPTION

scampi@ist-scampi.org 126 November 13th, 2003
"A textual string containing information about which device this interface is associated with. An example of this can be '/dev/eth1'."

::={ scampiIfEntry 3 }

scampiIfAlias 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 scampiIfAlias associated with that interface is the zero-length string. As and when a value is written into an instance of scampiIfAlias through a network management set operation, then the agent must retain the supplied value in the scampiIfAlias 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 scampiIfIndex value."

::={ scampiIfEntry 4 }

scampiIfGPSSync OBJECT-TYPE
SYNTAX TruthValue
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"A boolean value used for signalling time synchronization problems. Many SCAMPI adapters will support GPS time synchronization. A true value signals that this synchronization with GPS is working. A false value signals some type of problems with the synchronization. Interfaces that does not support proper time synchronization should set this value to false."

::={ scampiIfEntry 5 }

scampiIfBusy OBJECT-TYPE
SYNTAX TruthValue
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"A boolean value used for signalling if the interface is busy capturing packets. A true value is used if the interface is busy and false if it not in use."

::={ scampiIfEntry 6 }

scampiIfPkts OBJECT-TYPE
SYNTAX Counter64
MAX-ACCESS read-only
APPENDIX G. SNMP ACCESS

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."
::=( scampiIfEntry 7 )

scampiIfOctets OBJECT-TYPE
SYNTAX Counter64
MAX-ACCESS read-only
STATUS current
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 scampiIfCounterDiscontinuityTime."
::=( scampiIfEntry 8 )

scampiIfDroppedPkts 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."
::=( scampiIfEntry 9 )

scampiIfCounterDiscontinuityTime 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."
::= { scampiIfEntry 10 }

-- Information about MAPI flows *******************************************

scampiFlows OBJECT IDENTIFIER ::= { scampiMIB 2 }

scampiFlowCfgMaxHistLength OBJECT-TYPE
SYNTAX  Integer32
G.1. SCAMPI MIB

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

scampiFlowCfgMaxDays OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Specifies the maximum age in days of entries in scampiFlowTable. This is the number of days since scampiFlowStop.
A negative number 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 { scampiFlowUid scampiFlowIndex }
 ::= { scampiFlowTable 1 }

ScampiFlowEntry ::= SEQUENCE
{
  scampiFlowUid Integer32,
  scampiFlowIndex Integer32,
  scampiFlowIfIndex Integer32,
  scampiFlowPkts Counter64,
  scampiFlowOctets Counter64,
  scampiFlowDroppedPkts Counter64,
  scampiFlowStart TimeStamp,
  scampiFlowStop TimeStamp
}

scampiFlowUid OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS not-accessible
STATUS current
APPENDIX G. SNMP ACCESS

DESCRIPTION
"The UID of the process that initiated this flow."
::= { scampiFlowEntry 1 }

scampiFlowIndex OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION
"The index number of the flow. The index number is a sequential
number starting at 1 for each unique scampiFlowUid."
::= { 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 }

scampiFlowPkts OBJECT-TYPE
SYNTAX Counter64
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The total number of packets captured by the flow."
::={ scampiFlowEntry 4 }

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

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 6 }

scampiFlowStart OBJECT-TYPE
SYNTAX TimeStamp
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The sysUpTime of when the flow started."
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 8 }

-- General information about MAPI ************************************
scampiMapi OBJECT IDENTIFIER ::= { scampiMIB 3 }

scampiMapiUsers OBJECT-TYPE
SYNTAX Counter32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The total number of unique users that are currently using MAPI"
::={ scampiMapi 1 }

scampiMapiFlows OBJECT-TYPE
SYNTAX Counter32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The total number of active flows in MAPI"
::={ scampiMapi 2 }

scampiMapiFunctions OBJECT-TYPE
SYNTAX Counter32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The total number of active predefined functions in MAPI"
::={ scampiMapi 3 }

scampiMapiFunctions OBJECT-TYPE
SYNTAX Counter32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The total number of active predefined functions in MAPI"
::={ scampiMapi 3 }

scampiMapiUserFunctions OBJECT-TYPE
SYNTAX Counter32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The total number of active user defined functions in MAPI"
APPENDIX G. SNMP ACCESS

::={ scampiMapi 3 }

-- Measurement group *****************************************************

scampiMeasurements OBJECT IDENTIFIER ::= { scampiMIB 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"
::={ scampiMeasurements 1 }

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

ScampiMesCfgEntry ::= SEQUENCE
{
scampiMesCfgUID Integer32,
scampiMesCfgIndex Integer32,
scampiMesCfgIf Integer32,
scampiMesCfgInterval Integer32,
scampiMesCfgMaxLength Integer32,
scampiMesCfgActive TruthValue,
scampiMesCfgStorageType StorageType,
scampiMesCfgStatus RowStatus
}

scampiMesCfgUID OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION
"The UID of 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 the
for the corresponding scampimesCfgUID value.

::={ scampiMesCfgEntry 2 }

scampiMesCfgInterval OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS read-create
STATUS current
DESCRIPTION
"The interval for measurements in microsecond"
::={ scampiMesCfgEntry 3 }

scampiMesCfgMaxLength OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS read-create
STATUS current
DESCRIPTION
"The maximum number of entries for this measurement in the scampiMesTable."
::={ scampiMesCfgEntry 4 }

scampiMesCfgMaxActive 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 5 }

scampiMesCfgMaxStorageType OBJECT-TYPE
SYNTAX StorageType
MAX-ACCESS read-create
STATUS current
DESCRIPTION
"The storage type of this conceptual row."
::={ scampiMesCfgEntry 6 }

scampiMesCfgMaxStatus 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."
::={ scampiMesCfgEntry 7 }
APPENDIX G. SNMP ACCESS

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 in a certain time interval"
INDEX { scampiMesUID scampiMesIndex scampiMesInterval }
::={ scampiMesTable 1 }

ScampiMesEntry ::= SEQUENCE
{
  scampiMesUID Integer32,
  scampiMesIndex Counter32,
  scampiMesInterval Counter32,
  scampiMesStartSec Integer32,
  scampiMesStartNano Integer32,
  scampiMesPkts Counter64,
  scampiMesBytes 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 }

scampiMesInterval OBJECT-TYPE
SYNTAX Counter32
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION
"Interval counter for a certain measurement. When wrapped existing intervals are overwritten."
::={ scampiMesEntry 3 }

scampiMesStartSec OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"scampiMesStartSec and scampiMesStartNano together form a timestamp for when the interval started. scampiMesStartSec contains number of second since midnight January 1 1970."
 ::= { scampiMesEntry 4 }

scampiMesStartNano OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"scampiMesStartSec and scampiMesStartNano together form a timestamp for when the interval started. scampiMesStartNano 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 }

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

END
Bibliography


