Abstract: This document contains the evaluation of the SCAMPI architecture and implementation both by analysis and by assessing the experimental results. We also study how successful the SCAMPI architecture has been in meeting the original objectives.
# Contents

1 Introduction ................................................. 9

2 SCAMPI objectives .......................................... 11

3 Overview of the architecture .............................. 15
   3.1 High-level view ....................................... 15
      3.1.1 The SCAMPI architecture .................... 15
      3.1.2 Limiting the overhead ....................... 17
   3.2 Software development model ......................... 18
      3.2.1 MAPI drivers .................................. 18
      3.2.2 MAPI function libraries .................... 18
      3.2.3 Function interfaces .......................... 19
   3.3 Other components ..................................... 20
      3.3.1 MAPI and management ......................... 20
      3.3.2 Network interfaces .......................... 23
      3.3.3 Middleware ................................. 28

4 Hardware evaluation ...................................... 35
   4.1 SCAMPI adapter ...................................... 35
      4.1.1 Analysis ...................................... 35
      4.1.2 Measurements ................................ 37
   4.2 Timestamp unit ...................................... 37
      4.2.1 Analysis ...................................... 37
      4.2.2 Measurements ................................ 38
   4.3 Summary ............................................. 38

5 Middleware ................................................. 41
   5.1 COMBO6 drivers and libraries ....................... 41
      5.1.1 Analysis ...................................... 41
      5.1.2 Measurements ................................ 41
   5.2 MAPId ................................................. 41
      5.2.1 Control analysis .............................. 41
      5.2.2 Control measurements ....................... 42
      5.2.3 Datapath analysis ............................ 43
CONTENTS

5.2.4 Datapath measurements ................................. 43
5.3 FPL-2 filtering language ................................. 48
   5.3.1 Analysis ........................................... 48
   5.3.2 Measurements ...................................... 55
5.4 MAPI on IXP1200s ....................................... 55
   5.4.1 Analysis ........................................... 55
   5.4.2 Measurements ...................................... 58
   5.4.3 Summary ........................................... 60
5.5 MAPI on generic NICs ..................................... 60
   5.5.1 Analysis ........................................... 60
   5.5.2 Measurements ...................................... 62
5.6 MAPI on DAG cards ....................................... 62
5.7 Authd .................................................... 62
   5.7.1 Analysis ........................................... 62
5.8 Summary .................................................. 64

6 Applications ................................................. 65
6.1 Accounting ................................................. 65
6.2 Stager ..................................................... 69
   6.2.1 Analysis ........................................... 69
   6.2.2 Measurements ...................................... 70
6.3 Intrusion detection and denial of service ............... 71
   6.3.1 Performance ....................................... 72
6.4 Netflow traffic analysis .................................. 72
   6.4.1 Analysis ........................................... 72
6.5 Traffic engineering ...................................... 74
   6.5.1 Measurements ...................................... 76
6.6 End-to-end QoS Monitoring .............................. 76
   6.6.1 Analysis ........................................... 76
   6.6.2 Measurements ...................................... 77
6.7 Summary .................................................. 78

7 Results ....................................................... 79
7.1 Analysis of the system architecture .................... 79
   7.1.1 Context ............................................ 81
   7.1.2 Issues for analysis ............................... 81
7.2 Assessment of the measurements ......................... 85
   7.2.1 The SCAMPI card as regular NIC ................ 86
   7.2.2 The SCAMPI card as SCAMPI card ................ 87
   7.2.3 Overhead of the architecture .................... 87
   7.2.4 The applications .................................... 88
7.3 Summary .................................................. 90
8 Comparison with objectives and competition
8.1 Comparison against objectives 
8.2 Comparison with competitors 
8.2.1 Normal 10Gig network card with PCI express board 
8.2.2 DAG with coprocessor 
8.2.3 IXP28xx 
8.2.4 FFPF

A MAPI functions tutorial
A.1 Introduction 
A.2 Structures 
A.2.1 mapidflib_function_def 
A.2.2 mapidflib_function_instance 
A.2.3 mapidflib_result 
A.2.4 mapi_result 
A.2.5 mapi_hw_info 
A.2.6 mapidflib_flow_mod 
A.2.7 mapid_pkt hdr 
A.3 MAPI function interfaces 
A.3.1 instance 
A.3.2 init 
A.3.3 process 
A.3.4 get_result 
A.3.5 change_args 
A.3.6 reset 
A.3.7 cleanup 
A.3.8 client_init 
A.3.9 client_read_result 
A.3.10 client_cleanup 
A.4 Tutorials 
A.4.1 Simple function: PKTCOUNTER 
A.4.2 Function with internal data: GAP 
A.4.3 Function with multiple process interfaces: PKTINFO 
A.4.4 Function that reads results from other functions: STAT 
A.4.5 Function with function specific get_result: TOBUFFER
List of Figures

3.1 High level overview of SCAMPI monitoring platform . . . . . . . 16
3.2 COMBO6 Mainboard . . . . . . . . . . . . . . . . . . . . . . . 24
3.3 COMBO-2XFP 10GE interface card . . . . . . . . . . . . . . . . 25
3.4 COMBO-PTM Timestamping card . . . . . . . . . . . . . . . . 26
3.5 PF_RING architecture . . . . . . . . . . . . . . . . . . . . . . . 27
3.6 Interrelationships of SCAMPI drivers and libraries . . . . . . . 29
4.1 Parallelized approach for 10 Gb/s processing . . . . . . . . . . . 36
4.2 Internal structure of COMBO-PTM . . . . . . . . . . . . . . . . 38
4.3 Time offset of COMBO-PTM clock . . . . . . . . . . . . . . . . 39
5.1 Processing time of identical rules (in clock cycles) . . . . . . . . 45
5.2 FPL-2 language constructs (\(m\) and \(n\) are arbitrary variables) . 50
5.3 Example of FPL-2 code: count TCP flow activity . . . . . . . . 51
5.4 Monitoring dynamic flows . . . . . . . . . . . . . . . . . . . . . 52
5.5 Packet addressing modes . . . . . . . . . . . . . . . . . . . . . . 53
5.6 IP packet fields (similar macros exist for UDP and TCP headers) . 54
5.7 Example FPL-2 filters . . . . . . . . . . . . . . . . . . . . . . . . 59
5.8 Throughput for different MAPI-X filters . . . . . . . . . . . . . . 60
5.9 Configuration with Juniper router . . . . . . . . . . . . . . . . . 61
5.10 JunoScript . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
6.1 "Virtual" streaming servers . . . . . . . . . . . . . . . . . . . . . 67
6.2 Stager architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 70
6.3 Stager setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
6.4 Netflow analyzer . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
6.5 Overview of Traffic Engineering application . . . . . . . . . . . 75
6.6 Architecture of the QoS monitor . . . . . . . . . . . . . . . . . . 76
6.7 Measured packet delay . . . . . . . . . . . . . . . . . . . . . . . . 77
6.8 Measured packet loss . . . . . . . . . . . . . . . . . . . . . . . . . 77
A.1 MAPI functions structure relations . . . . . . . . . . . . . . . . 100
Chapter 1

Introduction

By the end of the 1990s, it became clear that soon traditional network monitoring tools would not be able to cope with high-speed networks at link rate. The problems that were identified included the speeds of buses and memories that increase at a lower rate than that of network links, the lack of precise time stamping in commodity hardware and software, and the poor flexibility and scalability of existing solutions.

At the same time, it was felt that the importance of monitoring is growing, so that it is now a vital activity to an increasing number of users. In today’s networks, different users have different needs for monitoring. For instance, some may be interested in aggregate statistics for traffic engineering or accounting purposes, others may want to scan the payload of individual packets for intrusion attempts, and others still may use the monitoring platform for performing high-precision measurements and debugging purposes.

The need for monitoring high-speed networks has led to a SCAMPI architecture that is, at all levels of the processing hierarchy, centered around a high-speed datapath. If the overall objective of the project can be summed up as ‘the development and validation of a monitoring platform that is both flexible and scalable’, the means of achieving this goal can be summarised as: ‘make the common case fast’. Where possible, slow path complexity, such as control and management operations, has been kept separate from the fast path. In addition, where possible we have tried to push as much of the processing as possible to the lower levels of the processing hierarchy.

The project covers all layers of the network monitoring problem, from the hardware that receives and pre-processes the packets (the consortium has developed a new programmable network card for packet processing on high-speed links), all the way up to the end-applications (of which several were built during the project’s timeframe). Everything that is neither hardware nor application will be termed ‘middleware’ in this document. This includes typical control-path aspects such as authorisation, as well as data-path aspects, e.g., filters that are executed on the host processor in software. All three layers (hardware, middleware and application) will
CHAPTER 1. INTRODUCTION

be treated separately in this document, and within these layers, we will distinguish explicitly between control and datapath, where relevant.

At first glance, the layers coincide with well-known notions of ‘network card’, ‘low-level host software’, and ‘end application’, respectively. However, this is an over-simplification. For instance, the hardware may well extend beyond the machine. This is the case when the filtering capacities of a Juniper router are used. In addition, SCAMPI has developed functionality that cannot easily be captured uniquely by any one of these categories. Examples include the Monitoring API and the FPL-2 packet language. In principle, these could be implemented directly on hardware (and indeed, we have done exactly that for the FPL-2 language in a prototype implementation on the IXP1200). By using instead the categories of hardware, middleware and application, these entities naturally fall in the middleware section, as they are neither part of the hardware, nor of the software.

It can even be argued that software running on the network card is middleware also. However, while we admit that the boundaries between the classes are vague, we will mostly treat such functions in the chapter about the SCAMPI hardware. The reason is that there are many interdependencies between the functions and the hardware, so that combining the two seems a more logical fit than aiming for 100% consistency.

This document is intended to summarise and assess the SCAMPI architecture and implementation. The summary serves two purposes. First, it gives the SCAMPI consortium a final opportunity to compile the latest results, developments and results in a project deliverable. Second, it helps the reader to appreciate the assessment. Rather than a sales pitch the deliverable will try to evaluate the system as objectively as possible, which means that we will not eschew discussing shortcomings or disappointing results.

We conduct the evaluation of the architecture along several dimensions. Where relevant, we look at functionality to see how much it adds to or detracts from functionality in existing solutions. In addition, the performance of the functionality will be assessed in two ways. Using analytical evaluation, we identify the overhead in terms of the length of the datapath, bottleneck components, comparison with existing systems, etc. Experimental evaluation is then used to validate the results of the analysis, as well as to obtain overall performance results.

This document is organised as follows. Chapter 2 restates the original SCAMPI objectives. In Chapter 3, we explain the final version of the SCAMPI design and implementation. Next, in Chapter 4, the SCAMPI hardware is evaluated, together with the functions that are strongly related to the hardware. In Chapter 5, we turn to middleware, looking both at the control path and (especially) at the datapath. In Chapter 6 applications are assessed, with an eye on performance and functionality. Chapter 7 looks at the overall measurements and tries to use these as the basis for the evaluation. In Chapter 8, the project is summarised and compared against the original SCAMPI objectives. As other projects have also looked at network monitoring and new hardware and software is now available on the market, we also compare the SCAMPI results against the competition.
Chapter 2

SCAMPI objectives

In this chapter, we summarise the main objectives for the SCAMPI as listed in detail in deliverable D0.2.

SCAMPI’s overall goal is to **build a monitoring platform for backbone links that provides flexibility and scales with future link rates.** In addition, a range of requirements was categorised in deliverable D0.2. Without overly repeating what is stated in this deliverable, we briefly high-light some of the main points. As in D0.2., we use the keywords ”must”, ”must not”, ”required”, ”shall”, ”shall not”, ”should”, ”should not”, ”recommended”, ”may”, and ”optional” in this document as described in RFC 2119.

We explicitly list all the features that *must* be supported. In subsequent chapters we will also refer to features that SAMPI *should* or *may* support. While they either serve to back up or detract from the overall assessment of the success of the project, the *must have* requirements are the primary criteria for assessing SCAMPI. The main requirements for SCAMPI are that it must support the following features.

**Protocol analysis**  Related to protocols and analysis, SCAMPI must support the following features:

- Both single-mode and multi-mode fiber. For single mode both short range 1310 nm and long range 1530 nm lasers.
- Both SONET and SDH framing must be supported for 2.5 Gbps (OC-48/STM-16) and 10 Gbps (OC-192/STM-64) links.
- Packet over SONET (RFC2615) for 2.5 Gbps and 10 Gbps links.
- 1 Gbps Ethernet.
- IPv4 and IPv6 basic encapsulation.
- Filtering within any protocol header type.
CHAPTER 2. SCAMPI OBJECTIVES

- Mechanisms to define useful and not useful packet parts, i.e. what parts of useful packets should be stored.
- Sampling based on selection of every \( n^{th} \) packet.
- Sampling based on address and port hashing.

Regarding the protocols, it should be noted that the original deliverable was based on the specifications of the 4Plus card. Since then 4Plus has left the consortium and Cesnet and Masaryk have taken over the development of the card and the driver software, with the understanding that they would realise an Ethernet card and leave other protocol as ‘optional’.

**Monitoring requirements**  It is assumed that a SCAMPI monitoring system will run on a dedicated computer. The computer may be owned and managed by the network operator, or by another party. Running the SCAMPI monitoring system is the only task for this computer.

- Regarding management, the following information must be available:
  - host system memory usage;
  - host system disk usage;
  - network link status of monitoring hardware;
  - congestion warning;
  - number of active MAPI network flows;
  - number of active MAPI filters;
  - number of active MAPI functions;
  - total number of packets received by MAPI.

- Regarding the API, the following must be provided:
  - an API for passive monitoring;
  - an API which supports management of the monitoring system;
  - support for legacy libpcap-based applications.

- Regarding passive monitoring the following must be supported:
  - configuration of filtering and sampling on special purpose hardware when available;
  - possibility to read all data in packet subsets;
  - possible in MAPI to define how to separate network flows from packet subsets;
  - reading flow records generated from packet subsets;
  - both data captured in real time and previously captured data read from storage.
Hardware requirements  Regarding the hardware, the following should be supported:

- various types of monitoring hardware, including special purpose adapters that have on board processing capabilities;
- a special purpose adapter must include an on board precise clock for timestamping;
- possibility to set up and to continuously fine adjust the on-board clock phase and rate by software running at the host computer;
- possibility to synchronize the clock by an external PPS (pulse per second).

Application requirements  Within SCAMPI several applications will be developed. For a number of these applications the following specific requirements were drafted:

- Netflow/IPFIX probe
  - compliance to the IPFIX standard;
  - ability to support several formats for exporting flow records (which must at least include Netflow version 5 and IPFIX);
  - user configurability by means of (from a configuration file) of what format will be exported, the sampling procedure and the IPFIX configuration;

- Flow-based reporting:
  - must accept data from Netflow version, IPFIX and SCAMPI MAPI;
  - must be able to generate a multitude of reports, including average packet size distribution, packets per flow distribution, etc.¹;
  - aggregation of reports over hours, days, weeks, months and years;
  - a web-based interface with easy navigation and sorting options, reporting in both tables and graphs and easy selection of time periods;
  - CLI with easy support for period selection, sorting and output in plain ASCII.

- Threshold-alerting application which aims to create flow alarm notifications based on thresholds violations (for a list of examples of thresholds, refer to D0.2, Section 6.3.1):
  - ability to receive notifications via email.

¹We do not repeat all reports in this deliverable; refer to D0.2, Section 6.2.2 for the full list
QoS Monitoring:

- SCAMPI must provide provisions for correlating packets captured by two different SCAMPI platforms, such as precise timestamps and packet signatures for passive measurements or sequence numbers for active measurements; based on these provisions, the QoS monitoring application MUST be able to correlate packets from two different SCAMPI platforms.

- built-in knowledge of IPv4, IPv6, UDP, TCP and RTP protocols needed to perform monitoring of QoS characteristics related to these network protocols.

- generic way of supporting other protocols (e.g., by using byte offsets, etc.);

- ability of user to configure rules to specify a flow to be monitored, location of measurement points in case of two-point measurement, QoS characteristics to be monitored including parameters such as precision, time granularity, etc.
Chapter 3

Overview of the architecture

In this chapter we present a final overview of the SCAMPI architecture, as well as of the latest developments and additions to the implementation. This serves two purposes. On the one hand, it allows the consortium to present the most recent work in a project deliverable, while on the other hand, it serves to remind readers about the architecture, so that the rest of the deliverable will be easier to understand.

3.1 High-level view

The SCAMPI monitoring platform is designed with a keen eye on scalability, i.e., the ability to cope with future backbone rates. Some of the assumptions that were made at the start of the project were that network speed increases at a higher rate than the speed of buses and memory, so we cannot afford to send all packets to the host’s main memory. With the advent of PCI-Express, one of these assumption is no longer necessarily true. PCI-Express has standards defined for up to 32 Gbps of useful bandwidth (i.e., raw bandwidth minus the channel overhead). The jury is still out on whether the speed of buses will keep up with future link rates, but for the near future at least, bus bandwidth is probably no longer a bottleneck.

Unfortunately, this is not true for memory. The ‘memory gap’, both in bandwidth and in latency is growing at an exponential rate, and so, even if one of the bottlenecks (the slow bus) has been removed, the architectural decision not to send packets up to userspace proved to be the right one.

3.1.1 The SCAMPI architecture

The need for monitoring high-speed networks has led to a SCAMPI architecture that is, at all levels of the processing hierarchy, centered around a high-speed datapath [CdBB+03]. The project covers all layers of the network monitoring problem, from the hardware that receives and pre-processes the packets (the consortium has developed a new programmable network card for packet processing on high-speed links), all the way up to the end-applications (of which several were built during the
Everything that is neither hardware nor application will be termed 'middleware' in this document. This includes typical control-path aspects such as authorisation, as well as data-path aspects, e.g., filters that are executed on the host processor in software. A high-level overview of the main SCAMPI datapath components, as well as some of the control channels, is shown in Figure 3.1.

As shown in the figure, the central component in the SCAMPI architecture is Mapid, the Monitoring API daemon that connects to the hardware and device drivers at the bottom, and to the monitoring applications on top of it. Mapid receives requests from different monitoring applications, and implements them in the most efficient way. For instance, an application may request to receive a stream of all TCP packets that are sent to port 80 and contain the string ‘foo’. Assuming that the hardware supports simple filtering (as the SCAMPI adapter does), Mapid may implement this request by instantiating a ‘port 80’ filter in the network card and linking it to a string search algorithm in userspace (running in the address space of Mapid). The complexity is hidden from the application, which simply receives all relevant packets.

The software provides monitoring applications with a powerful Monitoring Application Programming Interface (MAPI), which enable them to express their monitoring needs in a device-independent way [PaAO04]. 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. Applications use the MAPI mainly for communication with Mapid (1). Using control primitives they specify in what flows or flow statistics they are interested, while datapath operations allow them to obtain the results.
3.1. Limiting the overhead

Mapid is implemented in user-space, avoiding costly operating system overheads associated with kernel implementations such as those provided by Linux Socket Filters. Mapid in cooperation 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, Mapid is able to read all the captured packets directly from user-space without invoking the kernel and without copying the packets from kernel space to userspace. For low-end NICs that are not capable of writing packets in a memory mapped area in main memory, there also exists an implementation on top of normal device drivers. However, the latter approach is less efficient as it requires kernel involvement.

Moreover, once Mapid has set up the flow (control path), applications may be handed direct access to the network card itself. For instance, an application may implement an application’s request for counting the number of TCP SYN packets that are sent to a specific website by configuring the network card to keep such a counter in hardware. It may then provide the application with a handle which allows it to read this counter directly, without intervention by either Mapid or the kernel.

In summary, the application may obtain results either from the network card directly, or from Mapid. Similarly, Mapid also may obtain the results directly from the hardware, or from an intermediate device driver. It should also be mentioned that the hardware need not reside as a plug-in board in the PC. While this is probably the most common configuration, the SCAMPI architecture also explicitly supports programmable hardware attached to the PC via a network link. For instance, the architecture is able to exploit the filtering capabilities of programmable routers (such as those marketed by Juniper).

An incomplete list of what the consortium has produced within the SCAMPI architecture includes:

- a high-speed network card with support for high-precision timestamps, filters, statistics, and programmability (versions for 1 Gbps and 10 Gbps were produced);
- an advanced monitoring API (MAPI) that offers many new features that could not easily be supported by existing solutions, e.g., pcap [PaAO04];
- the middleware as sketched above with support for the SCAMPI network card, DAG cards, normal NICs and JUNIPER routers - the number of times packets are copied is limited to one (or even zero) and the context switching is minimal;
- an implementation of the MAPI on the Intel IXP1200 network processor;
- a host of applications, including intrusion and denial-of-service-detection,
traffic engineering, accounting and several generic tools for storage, aggregation and presentation of network statistics;

- a stand-alone authorisation daemon, known as Authd [PB04];
- new packet processing languages (FPL-1 and FPL-2) with compilers for Linux on PCs and IXP1200 network processors [BP04, CB04, NdBCB04, BdBC+04].
- Linux kernel work (for normal NICs) on minimising packet copies by way of a ringbuffer that underlies the pcap library.

### 3.2 Software development model

One of the main design criteria when implementing MAPId was that it should be easy to extend. This has resulted in a design where support for new hardware adapters can easily be added by writing new MAPI drivers and libraries of functions that can be applied to network flows.

#### 3.2.1 MAPI drivers

Applications that use the SCAMPI architecture communicate with MAPId to perform measurements. By adding new MAPI drivers for hardware adapters, it is possible to run the same applications on top of multiple hardware adapters without having to make any modifications. The defined interface of MAPI drivers has many similarities to the MAPI interface used by applications and is the only restrictions placed on MAPI drivers. The drivers are therefore free to implement them as they see fit. Most drivers will however use a module called mapidlib to do most of the work.

Mapidlib is a generic library that can be used by all drivers to keep track of open flows, loaded function libraries and functions applied to flows. Many of the interfaces defined for a driver will simply forward the call to mapidlib which will do all or most of the processing.

The main job of a driver is therefore to initialize a device and retrieve new packets from it. A driver is also responsible for retrieving packets from files for offline processing. Each file format supported by MAPI is assigned to a MAPI driver and that driver is then responsible for decoding this specific file format.

#### 3.2.2 MAPI function libraries

MAPI function libraries contains functions that can be applied to network flows, like BPF filter, string search, byte counter etc. It is possible to load new function libraries to a running MAPId without interrupting already running monitoring jobs.

---

1See luca.ntop.org/Ring.pdf.
3.2. SOFTWARE DEVELOPMENT MODEL

and as long as no functions in a library is in use, libraries can also be removed dynamically.

3.2.3 Function interfaces

MAPId accesses MAPI functions through a series of well-defined interfaces. Most of the interfaces are optional so that when implementing new functions, only the interfaces that are needed by the function are implemented. The following interfaces are defined for a MAPI function:

- **Instance**: called when an application first applies the function to a flow. This interface may perform a simple syntax check of any arguments passed to the function and return an error message if there are problems.

- **Init**: called when an application connects to a flow to start the measurements. This interface allocates and initializes all resources that are needed by the function.

- **Process**: this interface does the actual processing of packets and is called for each new captured packet.

- **Get_result**: returns the results of this function to other functions or the application. There is built-in support for returning results to the client application through IPC or shared memory, but it is also possible to implement other methods like reading results directly from the hardware adapter.

- **Change_args**: used for changing the arguments of an already running function.

- **Reset**: resets the results of the function.

- **Cleanup**: called when the flow closes and should release all resources used by the function.

- **Client_init**: this interface is used for initializing code that runs on the client side.

- **Client_read_result**: used for implementing function-specific methods for returning results.

- **Client_cleanup**: called when the flow closes and releases any resources used by the function on the client side.

When a function has been implemented it can be added to a function library which can then be used by MAPId. For a detailed description of how to implement new MAPI functions see the tutorial in Appendix A.
CHAPTER 3. OVERVIEW OF THE ARCHITECTURE

3.3 Other components

3.3.1 MAPI and management

As an application programming interface, MAPI should provide a flexible but powerful abstraction to express complex user requirements in an easy and understandable way. The abstraction behind MAPI is the network flow. A network flow is defined to be a sequence of packets that satisfy a given condition. As an example, a simple flow can be defined as all network packets. MAPI offers low-level operations allowing the user to perform high-level operations based on the primitives that it offers. A successful example of the same approach in a related domain is found in the file abstraction of the file system interface, where the user can create, close, open, read and write flows. The file abstraction provides users with raw data allowing them to decouple data processing. MAPI is an analogous interface for network data where the user can create, close and read results from a network flow. It offers a basic set of primitive operations along with a wide range of predefined functions, providing a flexible way to combine them and build any monitoring application.

3.3.1.1 Creating, enabling and terminating network flows

Each flow is identified by a unique id: the flow descriptor. The user can create a flow by calling

flow_descriptor fd=mapi_create_flow(char *dev)

where dev is the name of the network interface. Packets can be also read from a trace file through:

flow_descriptor fd=mapi_create_offline_flow(char *filename,int type,int speed)

where filename is the name of the trace file and speed the pace that the packets will be replayed. The type parameter defines the format of the trace file and can be MFF_TCPDUMP (tcpdump traces), MFF_DAG_ERF (traces captured by DAG cards) or MFF_RAW (raw packets saved to a file).

After the successful creation of the network flow, it can be controlled by using its flow descriptor. When the flow is no longer useful it can be closed by calling

mapi_close_flow(flow_descriptor fd)

All internal structures of the flow and functions applied to it will be released. By default, a network flow is disabled after its creation. The user can call

mapi_connect(flow_descriptor fd)
3.3. OTHER COMPONENTS

to enable the flow so it can start processing packets. Network flows allow users
to organize the packets they are interested in monitoring into separate streams, and
thus to treat them differently. For example, often users are interested in monitoring
several sources of packets, and for each source of packet they may be interested
in monitoring different properties. Assume for the moment that a network admin-
istrator is interested in several flows at-a-time: he may be interested in observing
the bandwidth consumed by peer-to-peer file sharing systems that may be running,
while also being interested in detecting Denial of Service attacks on their web
server. On top of that, the site may participate in a trajectory sampling experiment
that samples and records a small percentage of packets. Organizing these three dif-
derent monitoring activities as separate flows, allows users (i.e., the administrators)
to identify them, to isolate them, and to treat them differently.

3.3.1.2 Applying functions

The condition that a network flow satisfies is described through the functions that
are applied to it. MAPI comes along with a wide range of predefined functions
such as packet and byte counters, BPF filters, pattern matching, sampling, packet
reasembly, hashing and logging to file. The user can apply as many functions as
needed to a flow, without any restrictions on the order of application.

Packets coming to the flows are passed through its functions in the order in
which the functions were applied. A function can decide whether the packet will
be examined by the next function or not. For example, if a BPF filter is applied to
a flow and a packet does not match this filter, then it does not proceed to the next
functions of the flow. As another example, packet and byte counters always return
a positive value indicating that the packet may proceed to the preceeding functions.

The type and order of functions form the condition of the network flow. For ex-
ample, if a network flow is described as the web packets that contain the “/bin/perl.exe”
pattern in their payload, it can be viewed as a flow with two functions applied: a
BPF filter “tcp and port 80” that matches all web packets followed by a pattern
matching function searching for “/bin/perl.exe”. The user can call

```
function_descriptor fid;
 fid=mapi_apply_function(flow_descriptor fd, char *function_name,...)
```

where `function_name` is the name of the function (e.g. “BYTE_COUNTER”
for the byte counter function). Depending on the function applied the arguments af-
eter `function_name` may differ. For example, packet and byte counter functions
take no arguments while BPF filter take the filter expression as an argument.

Some functions produce results and place them into their data structures, which
are part of a large shared memory segment. The use of shared memory avoids the
need of copying the results. The user can read the results of these functions with
the help of

```
void *mapi_read_results(flow_descriptor fd,function_descriptor fid, int copy)
```

scampi@ist-scampi.org 21 January 12, 2005
CHAPTER 3. OVERVIEW OF THE ARCHITECTURE

where copy takes boolean values (0 or 1) and indicates whether the results themselves will be copied or a pointer to the shared memory where results are located.

3.3.1.3 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
mapi_packet *p;
p = mapi_get_next_packet(flow_descriptor fd, function_descriptor fid);
```

where fid must be a function descriptor of a TO BUFFER function, a function that stores all packets arriving to it. If the user does not want to read one packet at-a-time and possibly block, he may register a callback function that will be called when a packet for the specific flow is available. The following call 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, function_descriptor fid, 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. Like in the case of mapi_get_next_packet, the fid must belong to a TO BUFFER function.

3.3.1.4 Management functions

To retrieve information and usage statistics of running flows the following two functions can be used:

```c
int mapi_get_flow_info(int fd, mapi_flow_info_t *info);
int mapi_get_next_flow_info(int fd, mapi_flow_info_t *info);
```

where info is a pointer to the structure that will hold the information about the flow. The following information about a flow is available:

- UID of user running the flow
- Flow descriptor
- Name of the device the flow is running on
- Number of functions that are applied to the flow
- Start time of the flow
3.3. OTHER COMPONENTS

- End time of the flow
- Status of the flow. Can be initializing, active, finished or closed.

The `mapi_get_flow_info` function returns information about the flow with flow descriptor `fd`. The `mapi_get_next_flow_info` function returns information about the next flow that has a flow descriptor that is higher than `fd`. This can be used for listing all available flows.

To get information about functions applied to flows, two similar functions exist:

```c
int mapi_get_function_info(int fd, int fid, mapi_function_info_t *info);
int mapi_get_next_function_info(int fd, int fid, mapi_function_info_t *info);
```

The information available about functions is:

- Function ID
- Name of function
- Name of library to which the function belongs
- The type of device with which the function is compatible
- The number of packets that have been sent to the function
- The number of packets that have been processed and passed through the function.

An SNMP MIB for NETSNMP has been implemented so that it is possible to retrieve statistics about MAPI flow and functions through SNMP.

3.3.2 Network interfaces

3.3.2.1 Combo6

The main SCAMPI hardware platform is the adapter based on Combo6 card. The adapter consists of three cards:

- Main card Combo6 (32/33MHz PCI bus) or its upgraded cousin the Combo6X (64/66MHz PCI bus). Main cards do not have any link interfaces, instead of it they have a connector for a daughter link card. The main board is shown in Figure 3.2.
- Link interface card Combo-4MTX (with copper GE interface), Combo-4SFP (with cages for GE SFP link modules) or Combo-2XFP (with cages for 10 GE link modules). The link interface card is shown in Figure 3.3.
- Precise clock card Combo-PTM which provides highly accurate timestamps. The timestamp card is shown in Figure 3.4.
CHAPTER 3. OVERVIEW OF THE ARCHITECTURE

The most advanced features of the adapter are functional blocks for monitoring applications support. This way, part of monitoring functionality is moved from software to hardware. The driver, libraries and MAPI implementation are complex as they have to provide access to these hardware blocks.

![COMBO6 Mainboard](image)

Figure 3.2: COMBO6 Mainboard

3.3.2.2 DAG

The current MAPI DAG driver only supports simple DAG cards with no processing capabilities. The driver is therefore relatively simple. It initializes the card, reads captured packets and passes them on to mapidlib which performs all the processing in software. The driver also has offline processing capabilities so that MAPI can process trace files that are stored in the DAG ERF file format.

3.3.2.3 Normal NICS

MAPI supports generic NICs (i.e., the network adapters that can be normally found on the market and that do not sport specific features for packet capture) by means of the libpcap library. Libpcap is a platform and NIC independent packet capture library that guarantees code portability across operating systems and network cards. This means that regardless of the network card being used (e.g., a generic NIC or even a specialized NIC such as a DAG card) the library captures packets in the same way, exploiting the operating system capabilities. Considering Linux as target platform, libpcap on Linux exploits a special type of socket named PF_PACKET that allows raw packets to be captured from a generic network adapter.

Linux packet capture speed is quite good on 100 Mbit networks. Unfortunately, on faster networks, even the fastest available PC equipped with Linux and a generic
3.3. OTHER COMPONENTS

Figure 3.3: COMBO-2XFP 10GE interface card

NIC cannot capture packets at wire speed. Hence, a new packet capture model has been designed, based on the following assumptions and requirements:

- Design a solution for improving packet capture performance that is general and not locked to a specific driver or operating system architecture.

- Given that network adapters are rather cheap, it is not too costly to allocate a network adapter only for passive packet capture, as the goal is to maximize packet capture performance and not reduce the overall costs.

- Device polling proved to be very effective, hence (if available) it should be exploited to improve the overall performance.

- For performance reasons, it is necessary to avoid passing incoming packets to the kernel that will pass them to userspace. Instead a straight path from the adapter to the user space needs to be identified in order to avoid the kernel overhead.

- Facilities such as packet sampling should be implemented efficiently. In fact with the current libpcap, in case of sampling all the packets are moved to userspace and then the sampled packets are discarded with a large waste of CPU cycles.

The idea behind this work is the following (see also Figure 3.3.2.3):

- Create a new type of socket (PF_RING) optimized for packet capture that is based on a circular buffer where incoming packets are copied.
The buffer is allocated when the socket is created, and deallocated when the
socket is deactivated. Different sockets will have a private ring buffer.

If a PF_RING socket is bound to an adapter (via the bind() syscall), such
adapter will be used in read-only mode until the socket is destroyed.

Whenever a packet is received from the adapter (usually via DMA, direct
memory access), the driver passes the packet to upper layers (on Linux this
is implemented by the netif_receive_skb and netif_rx functions depending
whether polling is enabled or not). In the case of the PF_RING socket, every
incoming packet is copied into the socket ring or discarded if necessary (e.g.,
in case of sampling when the specified sample rate has not been satisfied). If
the buffer is full, the packet is discarded.

Received packets for adapters with bounded PF_RING sockets, by default
are not forwarded to upper layers. Instead, they are discarded after they have
been copied into the rings. This practice increases the overall performance,
as packets do not need to be handled by upper layers but only by the ring.

The socket ring buffer is exported to userspace applications via mmap (i.e.,
the PF_RING socket supports mmap).

Userspace applications that want to access the buffer need to open the file,
then call mmap() on it in order to obtain a pointer to the circular buffer.

The kernel copies packets into the ring and moves the write pointer forward.
Userspace applications do the same with the read pointer.
3.3. OTHER COMPONENTS

- New incoming packets overwrite packets that have been read by userspace applications. Memory is not allocated/deallocated by packets read/written to the buffer, but it is simply overwritten.

- The buffer length and bucket size is fully user configurable and it is the same for all sockets.

![PF_RING architecture](image)

Figure 3.5: PF_RING architecture

The advantages of a ring buffer located into the socket are manifold, including:

- Packets are not queued into kernel network data structures.

- The mmap primitive allows userspace applications to access the circular buffer with no overhead due to system calls as in the case of socket calls.

- Even with a kernel that does not support device polling, under strong traffic conditions the system is usable. This is because the time necessary to handle the interrupt is very limited compared to normal packet handling.

- Implementing packet sampling is very simple and effective, as sampled packets do not need to be passed to upper layers then discarded as happens with conventional libpcap-based applications.

scampi@ist-scampi.org  27  January 12, 2005
Multiple applications can open several PF_RING socket simultaneously without cross interference (e.g., the slowest application does not slow down the fastest application).

The main difference with respect to normal packet capture is that applications that rely on libpcap need to be recompiled against a modified PF_RING-aware version of the library as incoming packets are stored into the buffer and no longer in the kernel data structures. In order to evaluate the proposed architecture, a different test-bed has been used. A fast PC packet sender is used to transmit packets towards a receiver PC equipped with a Pentium 4 running at 1.7 GHz with a 32 bit Intel GE ethernet card. The table below shows the test outcome.

<table>
<thead>
<tr>
<th>Pkt Size (bytes)</th>
<th>Linux 2.4.23, NAPI+PF_RING (Pkt Capture)</th>
<th>Linux 2.4.23, NAPI+PF_RING (NetFlow Generation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>550789 pps ≈ 202 Mbit</td>
<td>376453 pps ≈ 144 Mbit</td>
</tr>
<tr>
<td>512</td>
<td>213548 pps ≈ 850 Mbit</td>
<td>213548 pps ≈ 850 Mbit</td>
</tr>
<tr>
<td>1500</td>
<td>81616 pps ≈ 970 Mbit</td>
<td>81616 pps ≈ 970 Mbit</td>
</tr>
</tbody>
</table>

As the table shows, the NetFlow analysis is limited by the available CPU cycles. Moreover, there is a moderate packet loss only with tiny packets, whereas with medium and large packets there is basically no loss. The errors on the receiving interface have significantly decreased with respect to the existing setups. Considering that Gbit networks usually have jumbo (≥ 9000 bytes) MTUs, the PF_RING solution can very well be used for analyzing traffic at wire speed. Furthermore, it is worth to remark that the figures shown in the above table achieved using a low-end PC, are far better than many high-end routers available on the market. Additional tests performed using a traffic generator and a fast, Dual Xeon 2.8 with 64-bit GE cards, shown that PF_RING can capture over 1.2 Mpps, i.e., close to wire-speed.

In summary, packet capture has been greatly improved on PCs based on generic (commodity) NICs using PF_RING. The library allows to both give new life to existing pcap-based applications, as well as new monitoring libraries such as MAPI.

### 3.3.3 Middleware

The middleware consists of many components, not all of which will be described in detail in this section. In particular, we will reserve the discussion about such components as new languages for Chapter 5.

#### 3.3.3.1 Drivers and libraries

There are several drivers and libraries that form the lower part of SCAMPI middleware between the hardware monitoring adapter and the MAPI implementation. The interrelationships of these drivers and libraries is illustrated in Fig. 3.6. The MAPI drivers were already described in section 3.2.1.
Network adapter drivers Each supported network adapter has its own device driver that is loaded into the kernel and runs in kernel space. Standard Linux kernel distribution includes device drivers for common NICs. DAG cards use a driver provided with them and Combo6 cards use a driver available in Liberouter CVS (see www.liberouter.org). The Combo6 device driver is formed by several kernel modules and can be loaded by a `modprobe` command. For example, to load a driver for scampi-ph1 design (firmware code), you can use `modprobe scampi-ph1` command. Combo6 device driver provides a set of `ioctl()` calls to get information about the adapter, subscribe to its physical interfaces, read packets and read interface statistics.

DAG library and SCAMPI library The libraries (e.g., `libdag` and `libscampi`) provide a higher level of abstraction for applications that want to communicate with DAG or Combo6 device driver. The libraries provide a set of functions to open a new flow and start reading packets, read packets and read interface statistics. The libraries allow programmers to use one simple function call instead of manually mapping several pieces of memory and using several `ioctl()` calls to achieve the same results.

scampi@ist-scampi.org 29 January 12, 2005
libcombo  This library is a programming interface to units in the Combo6 adapter design (firmware code). The library provides functions to attach to the current design, to retrieve its structure, to configure LUP (the Lookup Processor with is CAM and SRAM memories), SAU (the Sampling Unit) and STU (the Statistics Unit), to read STU statistics and a few helper functions. Future version will also provide access to PCK (Payload Checker).

scampidump  The 'libscampid' library provides a higher level of abstraction to libscombo calls for setting up header filtering, sampling and statistics. We plan to enhance functions in these library such that it will be more extensible and to integrate it with libscampi.

3.3.3.2 Mapid

A central part of the SCAMPI software architecture is the MAPI daemon: Mapid. Since SCAMPI is a multi-user platform where users can execute different monitoring applications at the same time, there can often be situations where more than one application needs to perform the same type of processing on a packet. Mapid is a complex software component which implements the MAPI functionality on behalf of each user application. By centralizing the processing of arriving packets, Mapid can perform optimizations on a global scale that takes into consideration all applications from all users.

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 monitoring 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.

The daemon, which runs in userspace, communicates with the monitoring applications using IPC. Since this IPC communication is only used when new network flows are being set up, it does not introduce any substantial overhead. In cases where the results of a function have to be communicated back to the user, the user application can read the results directly through shared memory, avoiding any IPC communication.

Key characteristic of Mapid is modularity and extensibility. Support for loadable drivers makes possible to run the same application on top of different types of monitoring hardware, without having to make any changes to it. There is one MAPI driver for each type of hardware monitoring adapter supported by MAPI, and new drivers can be developed for future monitoring hardware. Furthermore, besides the standard MAPI function library which provides a set of generic monitoring functions, it is possible to create new function libraries which can be loaded dynamically while Mapid is still running. This enables the extension of MAPI with new functionality without interrupting existing monitoring jobs.
3.3. OTHER COMPONENTS

3.3.3. Authd

At the start of the SCAMPI project, it was thought that most of the middleware would run in the kernel of the OS, not unlike the drivers of network cards. For this purpose, the Open Kernel Environment (OKE) was adopted and developed in the direction that was most appropriate for SCAMPI [BS02]. The OKE was implemented both for the kernel and for network processor based network cards and a release is now available in the public domain. In retrospect, running the middleware in the kernel provides little advantage over running code in userspace, if the hardware is able to store the data directly in the address space of userspace applications.

When the applications run in userspace, however, SCAMPI-specific resource control at runtime becomes less critical. For example, memory protection in userspace is guaranteed by the memory management unit which is configured appropriately by the kernel. This means that there is no need to provide additional functionality in this aspect. Similarly, CPU-time is explicitly scheduled by the scheduler in the kernel and there are no ‘privileged’ APIs or security-sensitive structures that need protection. Unlike code running in the kernel, user code cannot access any of the privileged instructions that allow it to increase access rights to any of the above resources. Moreover, even a hard crash of the user code, will not jeopardise the stability of the entire system.

For this reason, the resource control in SCAMPI is modified to benefit from the features that are already present in the OS. In particular, the resource control has evolved to something that is more akin to admission control as found in, say, ATM networks. Whenever a user wants to create a SCAMPI ‘flow’ and apply various functions to it (e.g., samplers, string searches, etc.), the SCAMPI authorisation daemon that implements the admission control checks whether the user has the appropriate privileges to create this flow with this specific combination of functions and options applied to it [PB04]. For example, while certain users may not have the right to create a flow that receives all traffic with a string search algorithm applied to every packet, he/she may be granted the right to create such a flow provided the string search algorithm is preceded by a sampler that samples just 10% of the packets. The example shows that admission control, when taking a decision, should take the entire flow specification into account. This way, it is possible to specify that a certain combination of functions is allowed, while each of the constituent functions in isolation is not.

In addition, we have started implementing accounting of resources to avoid exhausting the resource capacity of the platform. For example, if a function search takes up a certain amount of buffer space and CPU time, the total number of this function that can be applied needs to be less than the total resource capacity of the system.

Admission control is implemented as a daemon process known as Authd. The information required to authorize a flow that a user wants to have connected include:
CHAPTER 3. OVERVIEW OF THE ARCHITECTURE

- the user’s public key;
- the user’s credentials (using the Keynote format [BFIK99]);
- a nonce-based authentication;
- 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 application domain (which is supplied by Mapid and in SCAMPI always equals "MAPI").

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. An example of the types of assertion that may be expressed in the credentials is detailed in D1.3. We now explain the various steps involved in authorising a user request.

The first thing a user needs to do is tell the system what his/her credentials are, and what public key can be used to identify this user. This only needs to be done once per application. Whenever the user wants to instantiate a flow, he/she also has to provide authentication to show that he/she really is the user that corresponds to the public key. The way this is done is by supplying a nonce value (an integer number), together with an encryption of this integer with the user’s private key. If the encrypted nonce decrypted with the public key equals the original value of the nonce, the user’s request is authenticated. By using the flow descriptor as the nonce value, the authentication is also tied to this particular request.

The user creates a flow and if successful a flow descriptor corresponding to the specified device is returned. The flow descriptor is used in all subsequent calls, e.g., to apply functions to the flow. The functions to be applied to the flow form a linked list of functions (each possibly having a linked list of parameters). The order in the list indicates the order in which the functions are to be applied.

Only when all the functions that a user wants to specify for this flow are applied, does the user call `connect_flow()`. This results in the following actions:

1. A flow specification structure is created that is a serialisation of the flow state that was created in the previous few steps (plus some extra parameters, such as the application domain).
2. The serialised flow+trust specification is sent by Mapid to the authorisation component (`Authd`).
3. `Authd` checks the authentication.
4. Authd parses the serialised flow+trust specification and extracts information from it regarding for instance:
   - the occurrence and order in which functions are applied;
   - the value of parameters that are passed to the functions;
   - the resource consumption.

5. This information, together with the credentials is then evaluated for compliance with Authd’s security policy.

6. If they did not comply, the connect_flow() request is rejected. If not it is accepted.

7. If the request was accepted by Authd Mapid will instantiate the flow.

Communication between Mapid and Authd is performed using shared memory. However, we have also implemented a front-end to Authd to allow secure, remote access (using SSL encrypted communication over sockets). In the current implementation, Authd runs as a daemon which, after reading the policy file, tries to setup the shared memory segment with Mapid at start-up. The configuration of Authd is currently still hardcoded. All the variables are located in a single file (config.h).

An experimental version of resource control and accounting is also included in the distribution. It allows Authd to check whether (regardless of privileges) there is enough capacity left to accommodate the new request. 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. Corresponding to each function to be applied to a flow, a ‘cost formula’ may be specified that calculates for each of the resources in the system (e.g., CPU, memory, FPGA/nanoengine space, etc) what amount will be consumed with the specified parameters. If the total amount of resources consumed exceeds the capacity, the request will be rejected.
Chapter 4

Hardware evaluation

In this chapter, we discuss and assess the hardware that has been developed within the project. Where possible, we repeatedly use the same template, where we first present an analysis of the system (e.g., concerning datapath length) and next discuss the measured results.

4.1 SCAMPI adapter

The SCAMPI adapter consists of the Combo6 mainboard and one of several interface cards. Currently three types of interface cards are available: Combo-4MTX (4-port twisted-pair Gigabit Ethernet), Combo-4SFP (4-port SFP transceivers Gigabit Ethernet) and Combo-2XFP (2-port XFP transceivers 10 Gigabit Ethernet).

A new version of the Combo6X mainboard with PCI-X interface operating at 64-bit and 133 MHz is in manufacturing for circuit population. We have also started work on a Combo6E mainboard that will support PCI-Express bus technology. The next generation of the interface card Combo-4SFPRO (supporting Gigabit Ethernet with Virtex II PRO and OC-48 with Virtex II PRO-X) and Combo-2XFP (supporting both 10 Gigabit Ethernet and OC-192) is in manufacturing to build a printed circuit board.

4.1.1 Analysis

The SCAMPI adapter performs pipelined processing in cascaded blocks. Therefore, 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 through filters and their throughput can be lower if necessary for technical reasons.
CHAPTER 4. HARDWARE EVALUATION

Figure 4.1: Parallelized approach for 10 Gb/s processing

HFE and LUP blocks

Simulations show that the HFE and LUP blocks can process more than two millions of packets per second. This is enough for a 1 Gb/s link as the time distance between two packets is not less than 500 ns. For a 10 Gb/s adapter, filtering is parallelized. Four filtering units 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 of 10 ns. It implies that one SRAM can serve four LUP units and together three SRAMs have to be installed on the adapter. The 500 ns time limit is enough long for one CAM access and for subsequent processing of nine instructions of LUP nanoprocessor. The parallelized processing is illustrated in Fig. 4.1.

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.

PCK block

The PCK block is based on a CAM organized as 272 bits in 8000 rows (the same type of CAM of used for LUP). Each pattern is up to 16 bytes long. The pattern is stored in the CAM in the form of all shifted pictures within a 32-byte grid. It implies that each pattern occupies 16 (for 16 byte long pattern) to 32 (for 1 byte long pattern) rows. A full CAM can contain at least 256 different patterns. The algorithm requires one CAM access (40 ns) for each 32 bytes of payload, therefore PCK processing speed is about 5 Gb/s. The PCK block is equipped with an 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
4.2. TIMESTAMP UNIT

PCI bus

The adapter uses standard 32 bit / 33 MHz PCI interface. Its maximum theoretical throughput is 132 MB/s, the practically achievable throughput is about 100 MB/s. Because packets can be filtered, sampled and searched directly on the card we do not consider the bus throughput as critical point. However, as we mentioned before, a new version of the mainboard with PCI-X 64-bit/133 MHz bus is in manufacturing and a PCI-Express version is in development.

4.1.2 Measurements

Measurement results are described in more detail in deliverable D3.4 - Description of experiment results. Here we summarize some of the observations.

When reading packets without any processing on the adapter, the maximum rate that could be processed without any loss by the was 37500 packets per second for both 1500-byte and 64-byte packets. This is a higher rate than could be achieved with Intel Gigabit Ethernet adapter that suffered from occasional losses at lower rates. However, the Intel adapter was able to pass a higher data rate because it has a faster PCI bus and it is optimized for fast packet transfer.

With unwanted packets filtered out on the adapter and not passed over the PCI bus to the host computer we can process a much higher number of packets per second than without filtering. For instance, with 64-byte packets when 10% of packets pass through the filter, we can process up to 500000 packets per second with very low packet loss (0.004%), which is about 10 times more than without filtering on the adapter.

We will test performance of payload searching when the scampi-ph2 design and corresponding driver is completed.

4.2 Timestamp unit

The COMBO-PTM is a PCI(32/33) card equipped with a XILINX Spartan3 FPGA, an embedded processor, a precise oscillator (TCX) and several connectors. The card was developed to provide precise and accurate timestamps, but it can be also used as a general time-base card. The card is illustrated in Figure 4.2.

4.2.1 Analysis

The format of the timestamps is derived from time representation in NTP. A timestamp is a 64-bits fixed point number, where the high 32-bits represent the number of seconds since time 0 (1/1/1970) and the low 32-bits represent the fraction of a second. The precision (granularity of time representation) is $2^{-32}$ sec, i.e., about

scampi@ist-scampi.org 37 January 12, 2005
CHAPTER 4. HARDWARE EVALUATION

Figure 4.2: Internal structure of COMBO-PTM

230 ps. Time representation in described format will be valid until the year 2106, when the 32-bits seconds counter will overflow.

We need to assign a unique timestamp to each packet. There can be up to 20 million packets each second on 10 Gb/s line. Therefore, the timestamp resolution has to be better than 50 nanosecond. The optimal clock frequency is 100 MHz.

The internal clock of the COMBO-PTM can be synchronized either by PPS (Pulse per second) signal or by NTP via the network. The preferred and most accurate method is using the PPS signal from a GPS receiver, in which case the accuracy of timestamp is about 1 microsecond. When an NTP server is used for synchronization, the accuracy can be about 50 nanoseconds depending on local setup (e.g., whether the NTP server is on the same LAN).

4.2.2 Measurements

The graph in Figure 4.3 shows the accuracy of the COMBO-PTM clock, as measured during a one day testing period. We can see worse accuracy in the last third of the graph, when an alternative and noisy PPS signal was used.

4.3 Summary

In summary, the hardware developed in the SCAMPI project is on the cutting edge of technology. For this reason, components are not always available and the ordering of components frequently incurs delays. Nevertheless, we have been able to build a complete set of hardware: a main card, a link card and a high-precision
timestamp card. While we are still in a phase of development, testing and refining the design of the card, we believe the results are promising. On the other hand, we admit that the current version of the hardware still falls a little short of the target speed. We expect to solve this in the new version of the card that should be available shortly.
Chapter 5

Middleware

In this chapter, we discuss and assess the middleware that has been developed within the project. Where possible, we use the same template where we first present an analysis of the system (e.g., concerning datapath length) and next discuss the measured results.

5.1 COMBO6 drivers and libraries

5.1.1 Analysis

The Combo6 device driver provides packets in batches, i.e., libscampi can request several packets using a single ioctl() call. These packets are then passed to the upper layers without copying. Nevertheless, data throughput is limited by the PCI bus, which currently operates at 32-bits and 33 MHz.

5.1.2 Measurements

We used the PAPI (Performance API) to find the number of instructions and cycles required to process one packet in various scenarios. Various MAPI functions were applied to a packet stream one at a time or several of them together. Results can be found in deliverable “D3.4 - Description of experiment results”.

5.2 MAPIId

5.2.1 Control analysis

In this section we analyze how control is performed inside MAPI.

When a user calls a MAPI function, mapilib creates a request message, in order to communicate with the MAPI daemon. This message is of the following form, and contains all the necessary arguments to implement the request

```c
struct mapiipcbuf {
  long mtype;
```
MAPI uses spin locks to protect critical regions from simultaneous access.

If the request stands for creating a new flow, a new UNIX socket is created in order for the user program to communicate with the MAPI daemon (Mapid). Then the daemon appends the new flow to the clients list of flows maintained by the daemon. Finally, the daemon sends a message to the client to report success or failure of the new flow creation.

If the request is a `mapi_apply` function, mapilib parses the function's arguments, and sends a message of the above format to the daemon requesting that the function be applied to the flow. The driver retrieves the function from the appropriate driver function list and returns the function id to the client.

On receiving the function id mapilib appends the function instance to the clients function list. When the processing of the packets begins the functions are to be called as entered on the function list.

When closing a flow, mapilib removes the information about the flow from the client side and communicates, through the socket, with the daemon, the flow is removed from the flow list.

When a function issues results, the MAPI daemon copies the results in a shared memory region. From this buffer, mapilib reads the results, when the clients issues a `mapi_read_results` function, and returns a pointer to them back to the client.

### 5.2.2 Control measurements

In this section we discuss measurements of the time needed by MAPI function calls. More specifically, we measured the time for a flow to get instantiated, the time for a function to get applied to the flow and the overhead of the `connect` function. As for the result function we measured the time for a `mapi_read_results` call and for a `mapi_get_next_pkt` call to return the results.

The measurements were taken in an Intel Pentium 4 CPU 3.00GHz. Our system has 512MB main memory, 512KB L2 cache. We used an Intel 82540EM Gigabit Ethernet Controller for live traffic experiments.

We actually measured clock cycles per function. The measurements were taken using RDTSC, which is an assembly function that returns the CPU difference between successive calls. A thousand (1000) measurements were taken for each function and the results shown are the average of those measurements.

In the table below we show the clock cycles measured and the corresponding times calculated by the CPU frequency. The clock cycles and the times below are
the ones that are seen by the user when he is calling functions.

In the case of \texttt{mapi\_get\_next\_pkt}, the function is implemented as a blocking call, so it blocks when there is no available packet in the interface and unblocks as soon as a packet is received. This blocking nature of the function explains the large number of cycles measured. Actually, the cycles do not represent only the pure overhead of the call, but also include the cycles spent in a 'idle' state due to the blocking behavior.

<table>
<thead>
<tr>
<th>Function applied</th>
<th>Cycles</th>
<th>Time (microsecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{mapi_create_flow}</td>
<td>239656</td>
<td>10.42</td>
</tr>
<tr>
<td>\texttt{mapi_apply_function}</td>
<td>295549</td>
<td>12.85</td>
</tr>
<tr>
<td>\texttt{mapi_connect}</td>
<td>2321584</td>
<td>100.96</td>
</tr>
<tr>
<td>\texttt{mapi_close}</td>
<td>221909</td>
<td>9.65</td>
</tr>
<tr>
<td>\texttt{mapi_read_results}</td>
<td>1387</td>
<td>0.06</td>
</tr>
<tr>
<td>\texttt{mapi_get_next_pkt}</td>
<td>44020184</td>
<td>1900.00</td>
</tr>
</tbody>
</table>

\section*{5.2.3 Datapath analysis}

For the analysis of the datapath, we take a closer look at the processing time of each individual MAPI function. We will try to determine the computational intensitivity of each applied function and the additional overhead of the SCAMPI framework with respect to the datapath. Because SCAMPI aims at scaling up to multi-gigabit networks, the performance of the datapath will be critical. Therefore, the overhead of the framework is kept as minimal as possible. In this section we will describe the results obtained by profiling the SCAMPI code.

When a packet enters the SCAMPI framework, the packet will be received by one of the SCAMPI drivers, \texttt{mapinicdrv}, \texttt{mapidagdrv} or \texttt{mapicombo6drv}. The driver will handle the reception of the packet and pass it to the MAPI daemon by calling the \texttt{mapid\_process\_pkt(pkt, pkt\_head)} function. This process will run through the list of all applied functions from all configured flows and call each function that is present in the functionlist (e.g., \texttt{function\_bpf\_filter(funct, pkt, pkthdr)} and \texttt{function\_counter(funct, pkt, pkthdr)}). Once all functions are executed, \texttt{mapid\_process\_pkt} will return control to the driver until a new packet is received.

\section*{5.2.4 Datapath measurements}

All measurements were done on an Athlon XP 1800+ @ 1533 MHz, 128KB L1 and 256KB L2 cache, 512MB main memory. On this machine the standard MAPI software is installed. To measure the number of instructions, used clockcycles or real time, Level 1 and 2 cache misses, the Performance Application Programming Interface or PAPI is used [pap] in addition to the Linux performance counters.
CHAPTER 5. MIDDLEWARE

user- and kernel-level timings are included in all measurements. The numbers in these tables include the overhead of PAPI itself. In order to establish the overhead of PAPI itself we used two consecutive calls to individual PAPI functions, with no intervening program code between them. One set of calls to PAPI required 300 instructions. The PAPI overhead should be subtracted from the packet and MAPI function overhead tables. All obtained results are the averages of $2 \times 14$ identical test-runs. This is done 2 times because it is not possible to measure Level 1 and 2 cache misses simultaneously.

5.2.4.1 SCAMPI framework overhead

Receiving a packet from a SCAMPI driver and passing it to the right processing functions takes time as well. In this section we measure the overhead of the SCAMPI implementation.

Table 5.1: Overhead of applying functions

<table>
<thead>
<tr>
<th>Empty functions</th>
<th>Instructions</th>
<th>Time ($\mu$s)</th>
<th>L1 misses</th>
<th>L2 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 empty flow</td>
<td>600</td>
<td>6</td>
<td>42</td>
<td>16</td>
</tr>
<tr>
<td>1 function in 1 flow</td>
<td>642</td>
<td>6</td>
<td>44</td>
<td>19</td>
</tr>
<tr>
<td>2 functions in 1 flow</td>
<td>685</td>
<td>7</td>
<td>48</td>
<td>19</td>
</tr>
<tr>
<td>2 functions in 2 flows</td>
<td>704</td>
<td>7</td>
<td>55</td>
<td>18</td>
</tr>
<tr>
<td>3 functions in 1 flow</td>
<td>726</td>
<td>7</td>
<td>52</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 5.1 shows the overhead of applying multiple functions. All configured functions don’t do anything but passing the packet through to the next function. This way, it is possible to measure only the overhead imposed by the SCAMPI framework. We measured the processing time it takes to execute the `mapid` function, i.e., the total time it takes to process a packet in the MAPI daemon (including the PAPI overhead).

Without any applied functions, it takes 600 (minus 300 PAPI overhead) instructions to read a packet from the driver and pass it through the `mapid` function. Applying an empty function to a single flow adds about 42 instructions. This includes the overhead from running through the function list, calling the function and returning to the `mapid` function. As illustrated in Table 5.1, adding a flow also takes some processing time (20 instructions), because now multiple flows need to be considered for finding the different applied functions.

5.2.4.2 BPF filter optimisation performance and overhead

Figure 5.1 shows the effect on the packet processing time of identical rules when using optimised BPF filters (by means of the elimination technique). Only the
processing time of the evaluation of the BPF filter is measured, the overhead of the SCAMPI framework is ignored. In this performance measurement, between 0 and 100 rules consisting of a single atom (src 10.10.10.173) were configured. With or without BPF optimisation, the packet processing time increases linearly with the number of configured rules. In case the BPF optimisation technique is used, the rule will only be evaluated once. The evaluation of all other rules will reuse the already obtained result. By consequence, the processing time increases much slower.

Table 5.2: Overhead of the BPF specific optimisation

<table>
<thead>
<tr>
<th>Non-optimised BPF</th>
<th>Instructions</th>
<th>Time (μs)</th>
<th>L1 misses</th>
<th>L2 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 filter</td>
<td>885</td>
<td>7</td>
<td>75</td>
<td>30</td>
</tr>
<tr>
<td>2 filters (no ident atoms)</td>
<td>1067</td>
<td>13</td>
<td>73</td>
<td>52</td>
</tr>
<tr>
<td>2 filters (1 identical atom)</td>
<td>1022</td>
<td>11</td>
<td>92</td>
<td>62</td>
</tr>
<tr>
<td>Optimised BPF</td>
<td>Instructions</td>
<td>Time (μs)</td>
<td>L1 misses</td>
<td>L2 misses</td>
</tr>
<tr>
<td>1 filter</td>
<td>897</td>
<td>8</td>
<td>90</td>
<td>26</td>
</tr>
<tr>
<td>2 filters (no ident atoms)</td>
<td>1074</td>
<td>14</td>
<td>108</td>
<td>62</td>
</tr>
<tr>
<td>2 filters (1 identical atom)</td>
<td>1002</td>
<td>11</td>
<td>109</td>
<td>67</td>
</tr>
</tbody>
</table>

The performance of the BPF optimisation techniques has been proven in previous documents such as [CSdB+04] and SCAMPI deliverable D2.3. However, in this section we take a closer look at the overhead of the combination of the “elimination” and “short-circuit” optimisation techniques caused by the SCAMPI framework. Table 5.2 shows the processing time of different optimised and non-optimised BPF filters, including the overhead imposed by the SCAMPI framework.

If we only consider a single BPF filter, no optimisation is possible. In Table 5.2 we see that the processing time of the optimised version is slightly higher than the
non-optimised version (897 compared to 885 instructions) when evaluating a simple expression (“port = 80”). This is because of the additional overhead imposed by the optimised implementation, such as breaking up a filter expression in its individual atoms and storing the result of the evaluation for future elimination. The same behaviour can be seen when configuring two filters without identical atoms, i.e. no result of an evaluation can be reused. However, when 2 BPF filters are configured with only one identical atom ((1) “icmp OR (tcp AND port 80) OR (udp AND port 8080)”, (2) “icmp OR igmp”), the SCAMPI framework overhead is reduced and we can already measure a small performance increase. If we consider only the processing time from both BPF filters, the first is slightly higher in case of the optimised version, while the processing time of the second BPF expression is significantly lower due to the elimination of the evaluation of the icmp atom. If no optimisation of the BPF rules is possible or there is only a single rule without any identical atoms, it might be better to turn off the BPF optimisation in the SCAMPI daemon. In all other cases, it is encouraged to enable the BPF optimisation.

5.2.4.3 Processing time of various MAPI functions

In this section we consider the implemented MAPI functions from the standard function library. Because each function implements its own functionality, the measured processing time will vary from function to function, depending on its complexity. Again, the overhead of the SCAMPI framework and the PAPI are included in the measurements.

<table>
<thead>
<tr>
<th>Function applied</th>
<th>Instructions</th>
<th>Time (μs)</th>
<th>L1 misses</th>
<th>L2 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bucket (a)</td>
<td>1289</td>
<td>14</td>
<td>121</td>
<td>87</td>
</tr>
<tr>
<td>Bucket (b)</td>
<td>1422</td>
<td>16</td>
<td>135</td>
<td>91</td>
</tr>
<tr>
<td>Byte counter</td>
<td>648</td>
<td>7</td>
<td>49</td>
<td>24</td>
</tr>
<tr>
<td>Ethereal</td>
<td>105995</td>
<td>261</td>
<td>2266</td>
<td>1587</td>
</tr>
<tr>
<td>Hash sampling</td>
<td>688</td>
<td>8</td>
<td>61</td>
<td>40</td>
</tr>
<tr>
<td>Stats</td>
<td>727</td>
<td>10</td>
<td>71</td>
<td>54</td>
</tr>
<tr>
<td>Threshold</td>
<td>753</td>
<td>9</td>
<td>79</td>
<td>42</td>
</tr>
<tr>
<td>To_tcpdump</td>
<td>1199</td>
<td>16</td>
<td>128</td>
<td>88</td>
</tr>
<tr>
<td>To_buffer</td>
<td>825</td>
<td>10</td>
<td>75</td>
<td>51</td>
</tr>
<tr>
<td>To_buffer_all</td>
<td>1198</td>
<td>13</td>
<td>118</td>
<td>87</td>
</tr>
</tbody>
</table>

Table 5.3 lists the processing time of various standard MAPI functions. For the measurement of the Bucket (a) function, a bytecounter was applied. The processing time was measured in case the bucket doesn’t overflow, i.e., the packet was added to an existing bucket. In Bucket (b) the same bytecounter was applied to the bucket function, but the bucket has timed-out and a new bucket must be created. Clearly,
5.2. MAPID

It takes some time \((1422 - 1289 = 133 \text{ instruction})\) to create a new bucket.

A bytecounter takes 648 instructions, including SCAMPI en PAPI overhead. Without the SCAMPI and PAPI overhead (measured in table 5.1), this results in 48 instructions to execute a bytecounter function. For the measurement of the Ethereal function, we used the same filtering expression as one of the BPF measurements (“port = 80”). Because Ethereal is very powerful and can parse a huge number of different protocols, its performance is much lower than an equivalent BPF filter (105995 instructions for ethereal compared to 885 for BPF). If possible, it is strongly suggested to use a BPF function. Ethereal should only be used when high-level protocols need to be parsed. Analogous to the bucket function, the measurements of the “stat” and “threshold” functions are in combination with a bytecounter. For the “stat” function statistics are computed for an applied byte-counter, while the “threshold” function triggers on a certain number of counted bytes by the bytecounter. For the “ToTCPdump”, “ToBuffer” and “ToBuffer_all” functions, 98 bytes packets were used. Table 5.3 shows that the performance of the “ToBuffer” function is significantly higher than the “ToBuffer_all” function. However, because of initialisation time of the “ToBuffer” function, only the first packet needs a higher processing time of 3120 instructions.

Table 5.4: Processing time of the stringsearch function

<table>
<thead>
<tr>
<th>FIND “GET”</th>
<th>Instructions</th>
<th>Time ((\mu s))</th>
<th>L1 misses</th>
<th>L2 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>98 bytes packets</td>
<td>2103</td>
<td>11</td>
<td>82</td>
<td>54</td>
</tr>
<tr>
<td>1500 bytes packets</td>
<td>21410</td>
<td>24</td>
<td>86</td>
<td>57</td>
</tr>
<tr>
<td>FIND “GET <a href="http://www.google.be/index.html%E2%80%9D">http://www.google.be/index.html”</a></td>
<td>Instructions</td>
<td>Time ((\mu s))</td>
<td>L1 misses</td>
<td>L2 misses</td>
</tr>
<tr>
<td>98 bytes packets</td>
<td>831</td>
<td>9</td>
<td>70</td>
<td>49</td>
</tr>
<tr>
<td>1500 bytes packets</td>
<td>2545</td>
<td>12</td>
<td>87</td>
<td>57</td>
</tr>
</tbody>
</table>

The processing time of finding a pattern in a packet is shown in table 5.4. Depending on the pattern and the size of the packets, more/less time is needed to do the searching.

Summary We think the implementation of Mapid and MAPI is close to being as lean and efficient as possible. The overhead of the framework itself is minimal, while the overheads of the functions are inherent to the tasks they perform. We went considerably further than just a straightforward implementation of existing filtering techniques (e.g., implementing a fairly extensive set of functions that are useful for many monitoring applications). In addition, we have shown that we can effectively optimise BPF filters that share a common prefix. In later sections we will show that we have also performed research in optimising different flows that

scampi@ist-scampi.org 47 January 12, 2005
share a common prefix in functions that are applied to the traffic. Presently, however, we will discuss our work on developing a better alternative to well-established languages like BPF.

5.3 FPL-2 filtering language

In this section, we assess the performance, usability and inherent efficiency of a new packet processing language, known as FPL-2 (FFPF packet language). It is designed as an alternative to BPF that is both faster and more flexible. Currently, an FPL-2 compiler exists for IXP1200 network processors, Linux kernel, and Linux userspace [CB04, NdBCB04, BdBC+04]. As explained later in this document, FPL-2 was originally developed in the context of the FFPF monitoring framework, but is now also ported to the SCAMPI Mapid architecture.

5.3.1 Analysis

5.3.1.1 Speed

Unlike BPF, FPL-2 is fully compiled and optimised to object code. By first compiling FPL-2 to C and then using a native compiler to generate object code, we use the strong optimisation techniques present in modern compilers. This is an important advantage. Instead of having to define a useful packet processing language and develop a high-grade optimiser for it, we can just focus on the former and let the C compiler worry about the latter. The result of compilation to native code is that FPL-2 is significantly faster than interpreted solutions such as BPF.

5.3.1.2 Flexibility

At least as important as speed is the flexibility that is added by the more expressive language. FPL-2 has three key features that are currently lacking in BPF: (1) a higher-level language design, (2) persistent memory, and (3) loops.

Starting with the first, unlike BPF, FPL-2 is not quite a language at assembly level. Rather, the language is structured much like a high-level language such as C. This makes it easier to write code in FPL-2 directly. In contrast, almost all BPF filters are compiled pcap expressions.

Persistent memory is useful when filters gather information about packets relative to other packets. A trivial example is a counter, which needs information about previous packets (i.e., the number of packets seen so far). The counter can be kept in persistent memory.

A more complex example is found in the handling of dynamic ports. The port number used for data transfer in many p2p and multimedia protocols is not known a priori. Instead, communication across a control channel (with a well-known port) is used to negotiate a port number for the data transfer. As a result, the data transfer is hard to monitor with solutions based on BPF. It can only send all
control packets to userspace, where an application must analyse them to extract the data port numbers and use these to install a new filter for these data packets specifically. In contrast, FPL-2 is able to intercept the control packets, extract the port numbers used for data transfer itself and store them in a filter-specific persistent memory area, known as MBuf. All incoming packets are then matched against the portnumbers in MBuf, so classification and filtering for protocols with dynamic ports is simple.

FOR loops introduce backward jumps to the filtering language. This is extremely useful, e.g., because it allows administrators to iterate over the length of the packet.

5.3.1.3 Architecture

FPL-2 assumes that there are two types of memory it can access: the network packet and a filter-specific memory area known as MBuf, which is used as persistent state. The assumption is that MBuf can be read both by the filter and by the userspace application. In existing implementations this is accomplished by memory mapping.

FPL-2 accesses MBuf and the packet much like a simple memory array. In addition, it uses registers for fast storage. Such a load-store register architecture can be mapped efficiently on modern architectures. As observed by McCanne and Van Jacobson: for modern processor architectures, stack-based languages are less efficient than register-based approaches [MJ93]. For this reason, we developed FPL-2 as a language that (a) compiles to fully optimised object code, and and (b) is based on registers and memory.

5.3.1.4 FPL-2 overview

As described above, FPL-2 achieves the goal of fast packet processing by using a custom front-end for FPL-2 filter expressions, and as back-end a native compiler (e.g., on Linux we use the well-known gcc compiler). As a result, the object code will be heavily optimised even though we did not write an optimizer ourselves. This object code will be invoked by Mapid (or another module that hosts filters, such as FFPF, as described later in this chapter) for each incoming packet. During packet handling, the filter object code updates the processing results in a local memory area shared with userspace, known as MBuf. Moreover, by using a safe language for the packet filters, the resulting system provides safety, while packets are processed at the speed of native code, fully optimised for the latest hardware. A trusted FPL-2 compiler and a custom code loader guarantee that only programs written in FPL-2 can be loaded in the framework.

5.3.1.5 The language

The FPL-2 language is summarised in Figure 5.2. It supports all common integer types (signed and unsigned bits, nibbles, octets, words and double words) and
allows expressions to get hold of any field in the packet header or payload in a
friendly manner. Moreover, offsets in packets can be variable, i.e., determined by
an expression. For convenience, an extensible set of macros allows use of short-
hand for packet fields, e.g., instead of asking for bytes nine and ten to obtain the IP
header’s protocol field, a user may abbreviate to ‘IP_PROTO’. We briefly explain
constructs that are not intuitively clear.

<table>
<thead>
<tr>
<th>operator-type</th>
<th>operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+, −, /, *, %, −−, ++</td>
</tr>
<tr>
<td>Assignment</td>
<td>=, %=, /=, +=, −=</td>
</tr>
</tbody>
</table>
| Logical/Relational | ==, !=, >, <, >=, <=, &
|                 | |, | |
| Bitwise         | &amp; | |

<table>
<thead>
<tr>
<th>statement-type</th>
<th>operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>if/then/else</td>
<td>IF (expr) THEN stmt1 ELSE stmt2 FI</td>
</tr>
<tr>
<td>for()</td>
<td>FOR (initialise; test; update)</td>
</tr>
<tr>
<td>external function</td>
<td>EXTERN(filter, input, output)</td>
</tr>
<tr>
<td>hash()</td>
<td>INT HASH(start byte, len, tablesize)</td>
</tr>
<tr>
<td>return a value</td>
<td>RETURN (val)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data type</th>
<th>syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register n</td>
<td>R[n]</td>
</tr>
<tr>
<td>Memory location n</td>
<td>MEM[n]</td>
</tr>
<tr>
<td>Packets access:</td>
<td>PKT.B[f(n)]</td>
</tr>
<tr>
<td>- byte f(n)</td>
<td>PKT.WI[f(n)]</td>
</tr>
<tr>
<td>- word f(n)</td>
<td>PKT.B[n].U1[m]</td>
</tr>
<tr>
<td>- bit m in byte n</td>
<td>PKT.B[n].U8[m]</td>
</tr>
<tr>
<td>- byte m in word n</td>
<td>(many options, including macros)</td>
</tr>
</tbody>
</table>

Figure 5.2: FPL-2 language constructs (m and n are arbitrary variables)

FOR. The FOR loop construct is limited to loops with a pre-determined number of
iterations. The break instruction, allows one to exit the loop 'early'. In this
case (and also when the loop finishes), execution continues at the instruction
following the ROF construct.

Registers and memory. FPL-2 is able to access the filter’s MBuf by means of the
assignment operator. For instance, one may assign the content of a memory
location to a register, perform a set of calculations, and then assign the value
of the register back to memory. All accesses to MBuf are checked for bounds
violations. An example of MBuf usage in FPL-2 is shown in Figure 5.3.

The code implements a filter that keeps track of how many packets were
received on each TCP connection (assuming for simplicity that the hash is
unique for each live TCP flow).
5.3. FPL-2 FILTERING LANGUAGE

// count number of packets in every flow,
// by keeping counters in hash table
// (assume hash is unique for each flow)
IF (PKT.IP_PROTO == PROTO_TCP) THEN
  // assign hash over TCP flow fields to register
  R[0] = Hash(14,12,256);
  // increment the pkt counter at this position
  MEM[R[0]]++;
FI

Figure 5.3: Example of FPL-2 code: count TCP flow activity

External functions. An important feature of FPL-2 is extensibility and the concept of an ‘external function’ is key to extensibility, flexibility and speed. External functions are an explicit mechanism to introduce extended functionality to FPL-2. While they look like filters themselves, the functions may implement anything that is considered useful (e.g., checksum calculation, pattern matching). They can be written in any of the supported languages, but it is anticipated that they will often be used to call optimised native code performing computationally expensive operations.

In FPL-2, an external function is called using the EXTERN construct, where the parameters indicate the filter to call, the position in MBuf where the filter can find its input data (if any), and the position where it should write its output, respectively. For instance, EXTERN(foo,x,y) will call external function foo, which will read its input from memory position x, and produce output, if any, at position y.

A small library of external filter functions has been implemented (including implementation of popular pattern matching algorithms, such as Aho-Corasick and Boyer-Moore). The implementation will be evaluated in Section 7. External functions in FPL-2 can also be used to ‘script together’ filters from different approaches (e.g., BPF+ [BMG99], DPF [EK96], PathFinder [BGP+94], etc.), much like a shell script in UNIX.

5.3.1.6 Monitoring application with dynamic ports

Many existing packet filters are not well suited for handling applications with dynamic ports. Such applications use control channels with well-known port numbers, while data transfer takes place over ports that are negotiated dynamically. Examples are found in peer-to-peer networks and multimedia streams that employ control protocols like RTSP, SIP and H.323 [SRL98, ITU96] to negotiate port numbers for data transfer protocols such as RTP [SCFJ98].

These flows are complex to monitor and the problem was considered important enough to develop special-purpose tools such as mmdump to tackle it [vdMCCS00]. Like xPF [IAIK02], mmdump adds statefulness to the pcap/BPF architecture and

scampi@ist-scampi.org
in addition allows filters to be self-modifying. A filter may capture and inspect all control packets and if they contain the port number to be used for data, modify itself to also capture these packets.

There is a simpler way to monitor dynamic ports. For example, given that RTSP packets are sent on port 554, the filter in Figure 5.4 filters out all such packets and passes them to an external function `GetDynTCPDPortFromRTSP`. When called, the function scans all RTSP session packets for the occurrence of ‘Transport’, ‘client_port’ and ‘server_port’ to find the port numbers that will be used for data transfer (e.g., audio and video). These ports are stored in an array `MBuf` (lines 4-5). If the packet is not RTSP, we check if the destination port of the packets is in the array of port numbers and if so, return the value `TRUE` (lines 7-9), so that the packet is sent to userspace. In other words, only data packets of streams that are set up using RTSP are sent to userland. Note that the example is for illustration purposes only. It is a simplified version of what real applications would use. For instance, we only deal with transfers that use TCP (also for the data) and extract just a single destination port (while the traffic is likely to be bi-directional).

```
1. // R[0] initially 0 stores number of dynports found
2. IF (PKT.IP_PROTO==PROTO_TCP) THEN
3.   IF (PKT.TCP_DPORT==554) THEN
4.     MEM [R[0]] = EXTERN("GetDynTCPDPortFromRTSP",0,0);
5.   R[0]++;
6. ELSE
7.    FOR (R[1]=0; R[1] < R[0]; R[1]++)
8.      IF (PKT.TCP_DPORT == M[ R[1] ] ) THEN
9.        RETURN TRUE;
10. FI
11. ROF
12. FI
13. RETURN FALSE;
```

Figure 5.4: Monitoring dynamic flows

5.3.1.7 Compile-time checks

The FPL-2 compiler is able to generate ‘resource safe’ code, i.e., it is possible to check at compile time how many resources can be consumed by an expression, how many loop iterations may be incurred, etc. Neither FPL-2 language supports pointers and interaction with the rest of the kernel is limited to the explicitly registered external functions. Also, while it is not possible to determine the resource consumption of external functions statically, we are able to check (and control) which functions may be called from a filter. As a result, a simple authorisation check rejects filter expressions that do not agree with the local safety policy and no runtime checks for resource consumption are necessary. At runtime the code only checks for array bound violations, divide by zero, etc.
In an approach modelled after the OKE [BS02], the FPL-2 compiler takes the filter expression, checks whether it is safe and if so, compiles it to a Linux kernel module which is subsequently compiled by gcc. It also generates a compilation record, which proves that this module was generated by the local (trusted) FPL-2 compiler. The proof contains the MD5 of the object code, the list of external functions called in the FPL-2 code and is signed by the compiler. We can use the compilation record to check whether or not this function can be safely loaded in Mapi.

5.3.1.8 Addressing modes

The addressing modes of packet data are important for ease of use. Therefore, we support several ways of addressing in order to provide an intuitive way of handling the data. We have already shown some examples in Figure 5.2, but we now discuss these in more detail.

5.3.1.8.1 Packet data addressing

Index addressing mode combined with variable offset index addressing mode can give any bit of data within the packet data, as shown in Figure 5.5. For improving the readability of programs, we used the lexical conventions according to the industrial standard IEC 1131-3.

<table>
<thead>
<tr>
<th>types</th>
<th>Example</th>
<th>point to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>PKT.B[num]</td>
<td>the whole byte 'num'</td>
</tr>
<tr>
<td></td>
<td>PKT.B[num].U4[0-1]</td>
<td>the lowest/highest 4 bits of byte 'num' cast to byte</td>
</tr>
<tr>
<td></td>
<td>PKT.B[num].LO or PKT.B[num].HI</td>
<td>the lowest/highest 4 bits of byte 'num' cast to byte</td>
</tr>
<tr>
<td></td>
<td>PKT.B[num].U1[0-7]</td>
<td>the bit of byte 'num' cast to byte</td>
</tr>
<tr>
<td>Word</td>
<td>PKT.W[num]</td>
<td>the word 'num'</td>
</tr>
<tr>
<td></td>
<td>PKT.W[num].U8[0-1]</td>
<td>the lowest/highest part of word 'num' cast to byte</td>
</tr>
<tr>
<td></td>
<td>PKT.W[num].HI or PKT.W[num].LO</td>
<td>the lowest/highest part of word 'num' cast to byte</td>
</tr>
<tr>
<td></td>
<td>PKT.W[num].U4[0-3]</td>
<td>a byte of word 'num'</td>
</tr>
<tr>
<td></td>
<td>PKT.W[num].U1[0-15]</td>
<td>a bit of word 'num'</td>
</tr>
<tr>
<td>DWord</td>
<td>PKT.DW[num]</td>
<td>the double-word 'num'</td>
</tr>
<tr>
<td></td>
<td>PKT.DW[num].U16[0-1]</td>
<td>the lowest/highest part of dword 'num' cast to word</td>
</tr>
<tr>
<td></td>
<td>PKT.DW[num].HI or PKT.DW[num].LO</td>
<td>the lowest/highest part of dword 'num' cast to word</td>
</tr>
<tr>
<td></td>
<td>PKT.DW[num].U8[0-3]</td>
<td>a byte of dword 'num'</td>
</tr>
<tr>
<td></td>
<td>PKT.DW[num].U4[0-15]</td>
<td>a half-byte of dword 'num' cast to byte</td>
</tr>
<tr>
<td></td>
<td>PKT.DW[num].U1[0-31]</td>
<td>a bit of dword 'num' cast to byte</td>
</tr>
</tbody>
</table>

Figure 5.5: Packet addressing modes
CHAPTER 5. MIDDLEWARE

Some examples of packet addressing are drawn in Figure 5.6. These examples show that it is fairly intuitive to access a specific IP field within the packet. The most used fields may also be defined as macros, allowing users to customise the way they express themselves. Moreover, using a register or memory variable as index for packet reference the language increases considerably the applications area. In the (trivial) example shown below, the sum of the first 20 bytes in the packet is computed.

```
FOR (R[0]=0;R[0]<20;R[0]++)
M[0]+= PKT.B[R[0]];
```

<table>
<thead>
<tr>
<th>IP field Macro</th>
<th>Possible full FPL-2 expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_VERSIONS</td>
<td>PKT.B[0].HI</td>
<td>U8 (the value of high four bits of the first byte)</td>
</tr>
<tr>
<td>IP_HDR_LEN</td>
<td>PKT.B[0].LO</td>
<td>U8 (the value of low four bits of the first byte)</td>
</tr>
<tr>
<td>IP_PRECEDENCE_TOS</td>
<td>PKT.B[1]</td>
<td>U8 (the value of whole byte)</td>
</tr>
<tr>
<td>IP_TOT_LEN</td>
<td>PKT.W[1]</td>
<td>U16 (the value of second word)</td>
</tr>
<tr>
<td>IP_DATAGRAM_ID</td>
<td>PKT.W[2]</td>
<td>U16 (the value of third word)</td>
</tr>
<tr>
<td>IP_FRAGM_AREA</td>
<td>PKT.W[3]</td>
<td>U16 (the value of fourth word)</td>
</tr>
<tr>
<td>IP_TTL</td>
<td>PKT.B[8] or PKT.W[4].HI</td>
<td>U8</td>
</tr>
<tr>
<td>IP_PROTO</td>
<td>PKT.B[9] or PKT.W[4].LO</td>
<td>U8</td>
</tr>
<tr>
<td>IP_CHKSUM</td>
<td>PKT.W[5]</td>
<td>U16</td>
</tr>
<tr>
<td>IP_SRC</td>
<td>PKT.DW[3]</td>
<td>U32</td>
</tr>
<tr>
<td>IP_DEST</td>
<td>PKT.DW[4]</td>
<td>U32</td>
</tr>
</tbody>
</table>

Figure 5.6: IP packet fields (similar macros exist for UDP and TCP headers)

5.3.1.8.2 Memory data addressing Another important addressing mode is used for accessing the memory locations. We support two types of memory: a shared memory array, MBuf, and fast local registers. MBuf should be provided by the hosting framework (e.g., Mapid, FFPF). Therefore, the filter module does not perform any dynamic memory allocation/deallocation.

Generally, using data stored in registers increases the processing speed in case of very often used variables. The maximum number of local registers are defined in the resource restrictions configuration file and depends on the hardware.

scampi@ist-scampi.org 54 January 12, 2005
5.3.2 Measurements

The result of compilation to native code is that FPL-2 is significantly faster than interpreted solutions such as BPF. For example, a simple tcpdump filter\(^1\) executed in BPF takes 740 cycles, while FPL-2 needs only 185 cycles for the equivalent filter on one of our test machines, an improvement by a factor of four. As filters get more complex (i.e., contain more instructions that must be evaluated for each packet), the performance gain increases. Unfortunately, BPF is too limited to perform anything that is much more complicated than variations of the above filters. We therefore conclude that the biggest advantage of FPL-2 may be the increased flexibility that allows one to perform complex processing in Mapid or even the network card.\(^a\)

5.4 MAPI on IXP1200s

The implementation of MAPI on a network interface equipped with the Intel IXP1200 network processor has come to be known as MAPI-X. It should be noted that MAPI-X is intended mainly to demonstrate that the SCAMPI approach is also amenable to platforms other than COMBO6 (or DAG) and to implementations different from SCAMPI’s Mapid approach. The implementation is embedded in the FFPF framework and concerns only MAPI [BdBC’04]. Although one can run any MAPI application on top of MAPI-X, it does not comprise the Mapid. Phrased differently, FFPF provides an alternative implementation of MAPI (both on the network card, in the kernel and in userspace).

5.4.1 Analysis

5.4.1.1 FFPF

On the one hand, FFPF is a research project that targets what may be described as a possible ‘next generation SCAMPI platform’. For instance, it allows for functions to be applied in a complex processing graph, rather than SCAMPI’s current linear list of functions. It is language neutral and today supports different types of packet languages and explicitly allows administrators to leverage hardware assists. Moreover, it has advanced support for mapping the functions to be applied at the lowest possible level of the processing hierarchy, while ensuring that if flows share a prefix of common functions, this prefix is executed only once\(^2\). On the other hand, it is more research-oriented than the current SCAMPI project and as such is not involved in developing high-level applications or supporting features needed for large-scale deployment, such as SNMP MIBs, etc. In addition, it has been implemented on only few platforms and does not yet support the same rich library of functions that is found in the Mapid implementation.

\(^1\)“ip src 192.168.1.3 and ip proto udp and dst port 54321”

\(^2\)Note that this operates at a different level than the BPF optimisation discussed in Section 5.2.4.2.
FFPF employs a single, shared packet buffer known as \texttt{PBuf} in which all packets are stored that are of interest to at least one of the applications in a group. A group is a collection of applications with the same access right to packets. As there is no notion of groups in SCAMPi, we will ignore this aspect and assume all applications are in the same group. FFPF maps the packet buffer to the address space of the applications in the group, so that (in principle) each application can access all the packets in this buffer. In addition, for every MAPI-like flow, FFPF maps another circular buffer to the address space of each application which contains pointers into the shared packet buffer. This buffer, known as ‘index buffer’, or \texttt{IBuf}, is not shared. If the packet matches a flow-specific expression (e.g., a filter), FFPF pushes the packet in \texttt{PBuf} and enters the index in the flow’s \texttt{IBuf}. If another flow is also interested in this packet, instead of copying the packet, FFPF just adds the packet index to the other flow’s \texttt{IBuf}. There exists a third flow-specific buffer, a byte array known as \texttt{MBuf}, that is shared between kernel and application. It is used as a generic information buffer which can be employed by the application and the filter/function operating in the kernel or on the card on its behalf, as they see fit (e.g. to report results). In this section, we discuss primarily the implementation of MAPI on the IXP1200 network card.

5.4.1.2 MAPI on the IXP1200

The IXP1200 contains a single StrongARM processor (running Linux) and 6 microengines (MEs) running no operating system whatsoever. 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. In our testbed, one or more IXP1200 boards are plugged into the PCI backplane of a host.

**Efficiency of processing** A single microengine is responsible for receiving packets from the network ports and storing them in \texttt{PBuf}. 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 MAPI-X FFPF, these microengines correspond to high-speed MAPI flows at a higher level.

Within the context of FFPF we developed an entirely new packet processing language, knowns as FPL-2 (the FFPF packet language version 2, see previous section). Compared to BPF, probably the most well-known filtering language, FPL-2 is:

- more expressive (e.g., it supports persistent, bounded loops, and explicit capabilities to call external functions, such as hardware assists);
- more efficient (as it compiles to fully optimised native code, rather than interpreted byte code, it is roughly 4 times as fast even for very simple filters).
The FPL-2 language was ported to the IXP1200, so the microengines that process network packets for MAPI flows are programmed using FPL-2.\(^3\)

As the packet buffer (PBuf), the index buffer (IBuf), and the memory array (MBuf) are all memory mapped, it is possible to store packets once (on the network card) and have all levels of the processing hierarchy access the network data directly. However, since such ‘true zero copy’ is only a good solution if the packet is not touched frequently from code running on the host processor (because accesses across the PCI bus are slow), we also implemented two other strategies in packet copying: copy-once, and copy-as-needed. In the former, there will always be one copy from the network card to host memory, while in the latter, the packet will be copied if and only if it is queued for the application. Which copying policy is used, can be configured at startup time and the ‘right’ choice depends entirely on the nature of the applications.

Whichever copying strategy is used, the packets are received in the circular buffers known as PBufs. As MAPI-X implements the MAPI, it fully supports MAPI’s way of handling packets in batches, i.e., readers advance their read pointers in a circular buffer by more than one (instead of handling reading each packet individually, handle maybe hundreds of packets at a time). Because PBufs are shared by multiple functions, and indeed multiple flows, we also need policies for dealing with buffer overflow. Specifically, as there may be multiple readers in a PBuf, we need a policy for handling the scenario that the packet producer catches up with the ‘slowest reader’. Again, we have implemented different buffer management strategies (BMSs) in MAPI-X and it is up to the system administrator to decide which one to use. Moreover, MAPI-X can easily be extended with a new BMS, if necessary.

The first strategy, known as Slow Reader Preference (SRP) is the traditional way of dealing with overflow: whenever the producer catches up with the slowest reader, all new packets are dropped. While this is semantically simple (and expected by some legacy applications), it means that the pace is set by the slowest reader. As an alternative, we developed Fast Reader Preference (FRP), where the producer just keeps writing, regardless of the status of the readers. In some cases, it will overwrite packet that are still unread by some tardy application. The key idea is that the application is able to check after it has processed a set of packets, whether these packets had been overwritten in the meantime and, hence, whether they should be considered lost after all. In this way, we may obtain quite different drop statistics for different applications that use the same buffer, i.e., slow readers drop more packets, which is a desirable property.

\(^3\)To illustrate the way FFPF serves as an environment for testing out ideas that are subsequently used in SCAMPI, the FPL-2 language, is now being ported to the SCAMPI Mapid environment.
5.4.2 Measurements

While MAPI-X was designed to scale with link rates, we do not have a 10Gbps IXP card yet. In the meantime, we measured its performance with a 232 MHz IXP1200 card, with two 1Gbps ports. We connected the card to a common Linux PC that was deliberately underengineered, with an eye on creating an environment in which host bus and memory speeds would become bottlenecks. We call this setup the NIC-FIX testbed. It resembles in many aspects the situation one would have in networks faster than 10Gbps.

An important constraint for monitors is the cycle budget. At 1 Gbps and 100 byte packets, the budget for four threads processing four different packets is almost 4000 cycles. As an indication of what this means, Table 5.5 shows the overhead of some operations. Note that these results include all boilerplate (e.g., transfers from memory into read registers and masking).

Without MAPI-X the maximum rate at which we can monitor the network is 602 Mbps for maximum-size packets, not nearly line-rate. To evaluate MAPI-X, we execute the three filters shown in Figure 5.7 on various packet sizes and measure throughput. Only \( A \) is a ‘traditional’ filter. The other two gather information about traffic, either about the activity in every flow (assuming the hash is unique), or about the occurrence of a specific byte. Note that the hash function used in \( B \) utilizes dedicated hardware support. The results are shown in Figure 5.8. We implemented three variations of filter \( C \). In \( C_1 \) the loop does not iterate over the full packet, just over 35 bytes (creating constant overhead). In \( C_2 \), we iterate over the full size, but each iteration reads a new quadword (8B) rather than a byte. \( C_3 \) is Figure 5.7 without modifications.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R[0] = \text{HASH}(26,12,256) )</td>
<td>200 cycles</td>
</tr>
<tr>
<td>( R[0] = \text{PKT.B}[0] )</td>
<td>110 cycles</td>
</tr>
<tr>
<td>( R[0] = \text{PKT.W}[0] )</td>
<td>120 cycles</td>
</tr>
</tbody>
</table>

Table 5.5: Approximate overhead of some operators

Above a packet size of 500 bytes, MAPI-X can process packets at line rate for \( A \), \( B \) and \( C_1 \). This means that if 10% of the traffic consisted of packets that match filter \( A \), the prefiltering in MAPI-X ensures applications like tcpdump would also handle link rate.

For smaller packets, filters \( C_1 \) – 3 are not able to process the packets within the cycle budget. Up to roughly 165,000 pps \( C_1 \) still achieves throughputs of well above 900 Mbps. Beyond that, the constant overhead cannot be sustained. \( C_2 \) and \( C_3 \) require more cycles for large packets and, hence, level off sooner. This suggests that simple prefilters that do not access every byte in the payload are to be preferred. This is fine, as the system was intended precisely for that purpose.

Just as for the \( C \) filters, throughput also drops for the simple filters \( A \) and

\(^4\) Although one of the consortium members has ordered such a card, it has not yet been received.
(A) filter packets:

\[
\text{IF (PKT.IP_PROTO == PROTO_UDP }
\quad \text{AND PKT.IP_DEST == X AND PKT.UDP_DPORT == Y)}
\quad \text{THEN RETURN 1;}
\quad \text{ELSE RETURN 0;}
\text{FI}
\]

(B) count TCP flow activity:

\[
\text{// count number of packets in every flow,}
\text{// by keeping counters in a hash table of size 1024}
\text{// (we assume for simplicity that the hash is unique}
\text{// and that there are at most 1024 flows)}
\text{IF (PKT.IP_PROTO == PROTO_TCP) THEN}
\quad \text{R[0] = Hash(26,12,1024); // hash over TCP flow fields)}
\quad \text{// increment the pkt counter at this position}
\quad \text{M[R[0]]++;}
\text{FI}
\]

(C) count all occurrences of a character in a UDP packet:

\[
\text{IF (PKT.IP_PROTO == PROTO_UDP } \text{THEN}
\quad \text{R[0] = PKT.IP_TOT_SIZE; // saved pkt size in register}
\quad \text{FOR (R[1] = 0; R[1] < R[0]; R[1]++)}
\quad \quad \text{IF (PKT.B[R[1]] == 65) THEN // look for char 'A'}
\quad \quad \quad \text{R[2]++; // increment counter in register}
\quad \quad \text{FI}
\quad \text{ROF}
\quad \text{M[0] = R[2]; // save to shared memory}
\text{FI}
\]

Figure 5.7: Example FPL-2 filters
when processing smaller packets. However, these drops occur for a different reason, namely because the receiving microengine simply cannot keep up.

5.4.3 Summary

The MAPI-X implementation shows that the MAPI is quite amenable to implementation on different hardware and different software. As such it proves the general applicability of the MAPI approach. Some of the design decisions of the Mapid implementation of SCAMPI were motivated by the need to have a fairly mature monitoring system that could readily be applied by the time the project ended. For this reason, some of the more adventurous and risky ideas were eschewed in favour of more certain and practical solutions. However, such potentially useful ideas were not abandoned. Instead, they were tried in systems like FFPF, which has fewer dependencies. Several successful ideas were subsequently incorporated in the SCAMPI Mapid implementation.

5.5 MAPI on generic NICs

5.5.1 Analysis

MAPI on generic NICs is based on libpcap. The file mapincicdrv.c implements the pcap driver that shields the implementation from the interaction with libpcap. MAPI-based applications will interact with pcap through the mapidstdflib library. From the application point of view there is no direct interaction with pcap but just with MAPI. Please note that in addition to live packet capture, pcap enables

Figure 5.8: Throughput for different MAPI-X filters

B when processing smaller packets. However, these drops occur for a different reason, namely because the receiving microengine simply cannot keep up.
MAPI to play with files stored on disk that contain packets and headers previously captured.

Generic NICs provide very poor filtering capabilities (e.g., VLAN and MAC address based) that are useless for network monitoring as the network card is usually set in promiscuous mode. This means that the NIC is responsible only for capturing packets and moving them, with the help of the kernel, to user space for application consumption. Libpcap provides packet filtering capabilities based on BPF (Berkeley Packet Filter). If the kernel supports it (this is the case for Linux), packet filtering occurs into the kernel, otherwise in user-space. Literature [DBRV03] evaluated the packet filtering process and demonstrated that it need not be a very costly operation *per se*, compared for instance to packet capture. However the filtering process is very costly because all the incoming packets need to be captured and then filtered. This means that it is necessary to improve filtering by delegating it to another entity. For this reason, SCAMPI provides and enhanced version of libpcap that is used in conjunction with Juniper routers and allows packet filtering to be performed in ASIC directly by the router at wire-speed, regardless of the type of network adapter and speed (e.g., Ethernet/POS, 1 Gbit/OC192) as well as packet type (direct Ethernet packets or encapsulated into another protocol).

See Figure 5.5.1. In this case the Juniper router is part of the monitored network, and it is configured to mirror network packets towards the monitoring station. When the monitoring application creates a filter that is pushed directly to the router by means of JunoScript, as shown in Figure 5.5.1.

![Configuration with Juniper router](image)

**Figure 5.9: Configuration with Juniper router**

JunoScript employs an XML-RPC based library that allows Juniper routers to be configured from remote hosts. The JunoScript-enhanced libpcap configures the
CHAPTER 5. MIDDLEWARE

Juniper router transparently to the MAPI application which remains unaware of the presence of the Juniper box. In case the monitoring box is not connected to a Juniper router the packet filter still occurs inside libpcap or the Linux kernel. The advantage of pushing filtering into the router is manifold because the monitoring box receives only those packets that match the filter so that it is possible to monitor a high-speed network using a low-end PC (assuming that most of the packets do not match the filter).

Figure 5.10: JunoScript

5.5.2 Measurements

As already explained in other sections of this document, the MAPI software layer is quite light and it introduces a limited performance and memory overhead. Therefore MAPI over pcap offers basically the same performance of a pure pcap-based application with the little overhead due to the MAPI layer.

5.6 MAPI on DAG cards

When using MAPI on DAG cards, MAPId uses the MAPId DAG driver to read captured packets from the DAG cards. The MAPI DAG driver reads packets directly from the buffer in which the DAG card store packets in and has therefor very little extra overhead. All captured packets are forwarded to mapidlib and the performance is therefor dependent on the overhead added by mapidlib.

5.7 Authd

Authd is used to vet user requests at control time [PB04]. For instance, when a user wants to instantiate a flow, Authd will check the user’s credential and return either ACCEPT, or REJECT, depending on whether the request complies with the local security policy.

5.7.1 Analysis

Originally, the communication between Mapid and Authd could only be performed using shared memory. While the shared memory option is still supported,
it is also possible to connect to Authd using other communication primitives. For this purpose, the back-end of the daemon (where authorisation checking and resource calculation are performed) is fully decoupled from the front-end, which takes care of communication. The design is modular and replacing a communication package with another one is trivial, and we have implemented several such modules in the current distribution.

In particular we have implemented a solution that allows one to connect to Authd using sockets and OpenSSL.

- Connect to Authd from anywhere;
- All communication across the socket is encrypted;
- Client authentication;
- Server authentication.

The above features allow Authd to be used in a wider context than just on a single SCAMPI monitoring node. For instance, it is possible that an organisation installs a single authorisation daemon for an entire network. Moreover, the daemon is fully stand-alone and can be used without modification by projects other than SCAMPI as well. In fact, Authd provides a generic, application-neutral authorisation and admission control solution that can be applied to any project in need of such a facility.

In the Mapid context, the default option is to communicate with Authd using shared memory. At start-up, after reading the policy file, Authd tries to setup the shared memory segment with Mapid.

The daemon waits for a semaphore to be set which indicates there is a new request. At that point it will first check the authentication of the user. If this does not result in errors, it will extract all the information needed from the data supplied by Mapid and see if it complies with its security policy and resource availability. The result is passed back to Mapid via the shared memory area. For the authorisation procedure we have used the Keynote system [BFIK99]. Currently we only support RSA encryption and use 1024 bit keys.

In our opinion, the criterion for assessing Authd should be safety and flexibility, rather than speed. The authorisation process lies fully on the control path, and as long as it returns results at interactive speeds, the exact response times are not important. Nevertheless, given reasonable policy files and credentials, both the acceptance notification and the resource consumption are returned within milliseconds, despite fairly intensive public key cryptography. On the one hand, employing cryptographically protected requests, a well-known trustserver) and nonces, means

---

5 OpenSSL is an open source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library, see http://www.openssl.org/
that the system is as secure as the encryption methods themselves (which also implies that it is fairly simple to replace the encryption algorithm with another if the need arises). Moreover, requests cannot be replayed, or modified. On the other hand, the authorisation does not protect either itself, not Mapid from denial of service attacks.

Regarding flexibility and safety, we believe that the Authd, while not providing resource-safety at the granularity as provided for instance by the OKE, still provides sufficient access control over resources to handle the request of the userspace applications found in SCAMPI. For code running on lower levels of the processing hierarchy (typically the network card), the Authd approach is able only to return a acceptance verdict based on fairly static authorisation policies such as: the number of IP filters and string searches to be installed in the hardware should not exceed \( n \). This is the same problem faced by most implementations of devolved control in network devices (e.g., that of admission control in ATM networks).

As to resource control, the approach is deliberately also somewhat simplistic. It assumes that resource consumption (of various kinds of resources) can be expressed in fairly static mathematical formulas that do not depend directly on traffic. While this may be an oversimplification, it makes the problem tractable. Moreover, as the resource consumption concerns mainly userspace applications, strict enforcement of resource limitations is less important, because below these policies there are still resource restrictions enforced by the Linux kernel (e.g., using the MMU, timer interrupts, etc.)

To make life as easy as possible for users of the SCAMPI platform, we have tried to make the use of the authorisation process transparent. For instance, users need not worry about the precise nature of the credentials they supply. Indeed, the credentials are stored in a special directory in the user’s file space and supplied transparently to the authorisation daemon. Even so, if desired the authorisation daemon remains optional and may be omitted completely. This may be acceptable of all resources as well as the applications are under control of a single omnipotent administrator.

5.8 Summary

We have shown that all middleware components deliver speed and/or flexibility exceeding that of existing solutions. In particular, this chapter argued that we have developed a better monitoring API (with little intrinsic overhead and the possibility to push functionality to the hardware), a better filtering language (and optimisation of a popular existing language), support for several hardware platforms, low-level device drivers, etc.

In the next chapter, we will turn to the applications. In particular, we will assess the applications that were built and also look at how these applications benefit from the SCAMPI architecture.
Chapter 6

Applications

In this chapter, we discuss and assess the applications that have been developed within the project. Where possible, we use the same template where we first present an analysis of the system (e.g., concerning datapath length) and next discuss the measured results.

6.1 Accounting

Accounting applications are used extensively by Internet and Application Service Providers to improve their services and to bill clients. The ability to accurately measure various characteristics of network traffic such as byte count per source/destination addresses or byte count per source/destination addresses and TCP/UDP ports, allows providers to charge clients for their actual use, to verify that the usage was within the agreed-upon Service Level Agreements and to formulate new pricing policies that utilize better their available resources or are fairer for their customers.

In the context of SCAMPI project we developed two accounting applications based on MAPI functions, both of which produce formatted service detailed records (SDRs):

1. An application which captures the traffic generated by a range of IPs and classifies this per application (www, email etc) based on well known ports and direction (incoming, outgoing)

2. An application which captures the traffic generated by streaming servers, when many of them are running on the same host/port.

**IP Range Traffic Count**  This application monitors, using MAPI functions, the traffic of given IP ranges (clients IP pools) and counts the total number of bytes that each range sends or receives. It generates Service Detail Records (SDRs) based on IP pools and well-known services that can be used for applying per-application accounting policies.
Client IP pools are defined in the configuration file of the application that is parsed at startup. The format for each IP pool entry is as follows:

<Client, Network/Mask, CID>

where:

**Client** is a "key" string to identify that a new IP pool is defined

**Network/Mask** is the network IP and the mask number that define an IP pool (eg., 10.10.0.0/16)

**CID** is a number (ID) used to unique identify a customer

Then, per-client threads are created that make individual connections to the MAPI daemon and apply specific filters (using the `BPF_FILTER` function) to capture each client’s traffic. Finally the `FLOW_REPORT` function is used to generate the final records (SDRs) that are stored as plain text files in the directory defined also in the configuration file. The SDRs have the following format:

<CID, StartTime, Direction, Service, Volume, Duration>

where:

**CID** is a number (ID) used to correlate traffic data with a customer that has signed a contract

**StartTime** is the beginning of the measurement period

**Direction** is a field that represents the direction of the traffic (incoming, outgoing)

**Service** is how a service is discriminated by source/destination port numbers

**Volume** is the amount of traffic measured in bytes

**Duration** is the duration in msecs of the period measured

**Virtual Streaming Server Traffic Count** This application monitors, using MAPI functions, the traffic that different streaming services generate, when all of them are running on the same host and listen for requests to the same port. It generates SDRs with accounting data.

An example of a case where this application can be used, is depicted in Figure 6.1. Three different streaming servers (radio stations) are hosted on the same machine. All of them receive connection requests through the same port, e.g., port 80.

Streaming servers are defined in the configuration file of the application that is parsed at startup. The format of the configuration file is as follows:
Figure 6.1: "Virtual" streaming servers

<station, server_ip, station_name, station_id>

where:

**station** is a "key" string to identify that a new entry is defined and contains information about a streaming server;

**server_ip** is the IP of the host where the streaming server is located;

**station_name** is the name of the streaming server;

**station_id** is a unique ID for each streaming server.

For each streaming server a "monitor" thread is created. These "monitor" threads detect new connections (users) on their server respectively. Each one creates a new flow and applies to it a BPF_FILTER to separate the packets having as destination the it’s streaming server. Then a STRING_SEARCH is applied using the station_name field of configuration file. Furthermore, the TO_BUFFER function is applied in order to extract the IP and the port of the user where the packets of the stream will be sent. When a new user is detected, an "export" thread is created. The "export" thread deploys a new flow on the port found and measures, using the FLOW_REPORT function, the connection’s traffic. The SDRs that are produced have the following format:

<station_id, start_time, src_ip, src_port, dst_ip, dst_port, volume, duration>

where:

**station_id** is a unique id of the streaming service where the record corresponds;
start_time  is the timestamp when the first packet of the flow was sent;

src_ip  is the streaming server’s (source) ip;

src_port  is the streaming server’s (source) port;

dst_ip  is the listener’s (destination) ip;

dst_port  is the listener’s (destination) port;

volume  is flow traffic in bytes;

duration  is the duration of the flow in msecs.

Applications Testing and Usage  The accounting applications described above use the advanced features provided from MAPI. They evaluate and utilize mainly the functions for:

- creating a new flow on the MAPI daemon - `create_flow` function;
- searching for a specific string in a packet (e.g., The name of the streaming server in the URL) - the `STRING_SEARCH` function;
- filtering only the packets needed (e.g., those from/to a given IP) - the `BPF_FILTER` function;
- storing the full packet in order to process it (e.g., to find the port where a user will receive the packets of a stream) - the `TO_BUFFER` function;
- receiving the IPFIX records exported - the `FLOW_REPORT` function

They were installed and tested on FORTHnet’s R&D department network over regular NICs. Tests were also done using `athav01.forthnet.gr` where several radio-stations are hosted and transmit their program.

The data (SDRs) extracted from such accounting applications can be used on different tariff models for charging based on:

- total volume
- volume per direction
- volume per application
- volume per time zone
- total download time
- download time per zone
- number of distinct concurrent users
6.2. STAGER

Examples of product packages where such pricing policies can be applied are:

- Volume based charging for customers with static IP pools (e.g., leased lines)
- Time based charging for ADSL users
- Volume/Session based charging on “virtual” streaming services (e.g., hosted Radio Stations).

High-level assessment Because of MAPI, the accounting applications described were implemented in a more simple, elegant and efficient way. The MAPI filtering functions help applications to reduce the number of packets they have to process, improving largely their performance and minimizing the resources needed. These features combined with packet buffering, payload search and flow data export give them all the necessary flexibility to implement the algorithm of each application more easily.

6.2 Stager

6.2.1 Analysis

The Stager application was originally designed as a report generator with a web frontend for Netflow data, but has evolved into a generic application that can be used for aggregating and presenting most types of network statistics collected from MAPI or other sources.

In today’s high speed networks most core routers are not able to generate full NetFlow data and has to result to sampling. With MAPI on a dedicated measurement PC it is possible to generate full NetFlow data even at high speeds without any sampling. Stager is an application that can be used for demonstrating this.

Figure 6.2 shows the basic design of the application. One or more backend systems collect, process and insert data into an SQL database. The web-based user interface is very generic and lets users view and navigate the available reports. To keep the volume down to manageable size, raw data is not stored directly in the database but instead aggregated reports are stored. For the Netflow backend, reports can be an overview of destination interfaces, the top X source IP addresses that generate the most traffic, the most commonly used source ports, etc. Reports for an SNMP backend can be, for example, the temperature or the CPU load in a router or line load, packet loss, etc.

Report types are described by a table in the SQL database and the frontend uses this information to display the reports properly. To keep things independent from each other and for better scalability, each report type is stored in a separate table in the database. There is also one table for each time resolution that is available.
6.2.2 Measurements

No formal measurements have been conducted on Stager performance. However, the application is used in the UNINETT production network for displaying Netflow statistics. In this installation the database runs on a dedicated PC and three PC’s collects and process Netflow data and insert reports into the database. A separate PC is also used as a HTTP server for the web frontend. This setup is shown in figure 6.3.

The hardware specification of the PC’s are as follows:

- Flow collectors and HTTP server: 3GHz Xeon CPU, 1GB RAM, 2x36GB SCSI disks
- SQL server: dual 3GHz Xeon CPU, 4GB RAM, 400GB 4 disk software RAID 5

Each of the flow collectors collect Netflow data from the routers that are closest to them. The system collects Netflow data from 23 routers with a total of 277 interfaces ranging in speed from 155Mbs to 10Gbs. During peak hours more than 30GB of raw NetFlow data is collected each hour. The flow collectors process the raw netflow data and insert more than 375000 new entries in various tables in the database each hour, or more than 100 inserts each second. Currently there are more than 140 million entries in the largest report tables.

So far the performance of Stager has surpassed all expectations. Even with a relatively high load and millions of entries in several tables, the web interface is...
usually very quick and responsive. Most queries to the database are completed in less than 1 second. The only reports that go slower are overview reports where data from all available interfaces are displayed. But even these reports usually completes in less than five seconds.

The main bottleneck in the above setup is the IO performance of the software RAID. This is the reason for the lower performance for overview reports. The software RAID does not have the necessary bandwidth for retrieving all the necessary data sufficiently quickly. This bottleneck can also be noticeable when the SQL server performs routine maintenance in the background as the response time for viewing reports goes down. The solution to this is to use a proper hardware RAID with higher IO bandwidth.

6.3 Intrusion detection and denial of service

MAPI offers all the functionality needed to build a fast intrusion detection system. We have implemented such an intrusion detection system using MAPI, called sids. Sids takes as an input simple rules describing potential attacks and translates them into the appropriate MAPI functions. Each rule is mapped into a flow and can perform header analysis as well as packet payload inspection. The format of rules given as input to sids is fixed. As an example, the following describes a rule that searches for the “/bin/perl.exe” pattern inside TCP packets that are destined to port 80.

```
  count tcp and dst port 80; content: /bin/perl.exe; id:100;
```

The first keyword describes what action should be done if a packet matches the rule. So far, sids supports counting the packets matching the rule (keyword “count” in the previous example) and copying them into files (keyword “copy”). The second argument is a BPF filter, describing the header we search for. Furthermore, using keyword “content” sids can perform pattern matching. One rule can contain more than one patterns to be searched.

As MAPI supports flow cooking, sids packet inspection can become stateful so as not to miss attacks on packet borders. Furthermore, packets can be read from trace files either in tcpdump or dag format. Additionally, packets can be logged to a file using standard MAPI functions. The writing of rules describing attacks is a time-consuming task that involves careful selection and refinement of rules. For practical purposes, we have developed snort2sids, a tool that converts precisely Snort rulesets into sids rules. This tool saves users from writing rules by hand as Snort1 rulesets are large, frequently updated and publicly available. The snort2sids converts all Snort options like TCP flags, depth and offset of patterns and IP ranges.

1http://www.snort.org

scampi@ist-scampi.org  71  January 12, 2005
6.3.1 Performance

We have measured the performance of *sids* under various conditions. The results presented were taken using a lightweight version of MAPI and a fully-developed *sids*. In our experiments, we used live Ethernet traffic at 100Mbps rate. Two machines connected back to back were involved in our measurements, a sender and a receiver. The intrusion detection system and MAPI daemon were running on the receiver, a Pentium 4 at 3GHz with 1GB main memory and 256KB L1 cache. All packets were uniform (tcp and destination port 5001) and were generated with *ttcp* tool.

First, we examined how the processor load scales with the number of rules. Each rule was describing an attack on a specific port and more precisely attacks on a Web server. We tested two implementations of MAPI: an unoptimized one, where all flows are traversed linearly and an optimized one, where only the flows needed are examined (optimization is based on destination port). In our experiments, we measured that *sids* can examine up to 500 rules without packet loss for the unoptimized version but the load remains stable for the optimized one. In the case of optimized version, no rules matched the traffic examined as we had no web traffic and consequently redundant flow traversal was avoided.

Our second experiment was focused on measuring the performance of *sids* in cases where rules search for patterns inside the packet. We used the unoptimized version of MAPI as no header description was provided by the rules, thus no optimization could be performed. The results showed that *sids* can examine up to 150 rules containing patterns without packet loss. The payload of each packet was constant, as generated by the *ttcp* tool (–abcde...–).

6.4 Netflow traffic analysis

6.4.1 Analysis

The NetFlow collector and traffic analyzer has been designed to collect NetFlow flows generated by both physical routers and software probes as the MAPI-based IPFIX probe. The main application design goals are:

- Create an application able to serve the needs of an ISP where there are a few machines that provide services to a large community. This means that it is complementary to the Stager collector.

- Focus on the application service (e.g., web and email) rather than playing at protocol and network link level.

- Ability to collect traffic from various sources and NetFlow versions (v5 and v9 as well as the IPFIX-draft)

http://www.pcausa.com/Utilities/pcatcp.htm

scampi@ist-scampi.org
6.4. NETFLOW TRAFFIC ANALYSIS

- Web interface for easy access to the traffic reports.

The main application features are:

1. Released under the GNU GPL license.

2. Ability to collect flows from various NetFlow probes and routers.

3. Data are saved in two different repositories:
   - RRD database: used to store long-run (one year or more), per host traffic counters. RRD files are very compact and can be profitably used for storing numeric data such as interface counters
   - SQL database: used to store all the incoming flows. The SQL database is used only for the recent past as it is:
     - Not very space efficient in the long-run because saving individual flows takes more space than saving host counters.
     - Often not required to have a detailed view (in term of flows) for past activities.

4. Detailed traffic statistics:
   - Per host
   - Per group of hosts (cluster).

5. Ability to generate alarms based on traffic thresholds.

6. Traffic reports are available in two formats:
   - HTML, for on line access.
   - PDF for producing paper reports (e.g., for billing purposes).


The architecture of the netflow analyzer is represented in Figure 6.4.1. The main application components are:

- Flow collector: responsible for receiving raw flows and saving them on disk. This is necessary for both keeping up with probe speeds and aggregating data at each sampling interval (usually each minute).

- Flow analyzer and aggregator: it reads flows stored by the collector, decodes and handled them according to the aggregation and filtering rules specified in the SQL databases that stores both traffic information as well as application preferences.
CHAPTER 6. APPLICATIONS

- Alarm generator: based on the configuration stored in the SQL DB, this component handles the traffic counters stored in the RRD files in order to check the thresholds and emit alarms.

Please refer to [S.p04] for a complete list of features, application usage and configuration.

Figure 6.4: Netflow analyzer

6.5 Traffic engineering

Traffic engineering requires feedback from monitoring at different levels. A first level is a pure “diagnostic” monitoring (i.e., call holding time, average data volume, etc.). This is optionally complemented by an “operational” monitoring feedback (in which case we talk about “adaptive TE”).

The SCAMPI Traffic Engineering application is an example of “adaptive” load balancer. Since flows do not have a known nor constant bitrate or user behaviour deviates from the expected estimations, one cannot simply rely on a fixed division of flows amongst the given paths. SCAMPI can be used to drive the division of traffic over the paths.
As illustrated in figure 6.5 the TE application is used to split the traffic of a local network over different Internet uplinks. We assume we have 2 operational Internet connections (ADSL and cable) and we have one ISDN overflow link. The traffic is divided such that cable access receives 60%, and ADSL 40% of the total traffic. If both connections can not handle the total amount of traffic, the ISDN overflow connection is used as well. In this application it is assumed that every flow is requested explicitly, after which a configuration for that flow is added to the egress router. When an “over-limit” event is received from one path, all traffic goes to the other paths. When an “under-limit” event is received, a path start attracting traffic again. When both paths are either under- or over used, traffic is divided according to the 60-40 division. Since flows have only a limited lifetime, this mechanism allows to balance the traffic close to the configured value.

Because an instance of SCAMPI should run on an operational egress router, performance is of crucial importance. However, the configured actions are quite simple. For both uplinks a bytecounter is configured. If one of the counter exceeds its pre-configured upload bandwidth, an event should be generated. Contrary, if the load drops after an over-limit, an under-limit event will be generated. The processing of these events and the re-configuration of the egress router is the responsibility of the controller.

The main shortcoming of this application is that flows have to be requested explicitly. In a more advanced implementation, new flows should be detected automatically (e.g., by means of syn-ack messages) and the flow can be mapped implicitly. This means that the configuration of the SCAMPI monitor in the egress router needs to be extended in order to detect these packets.
6.5.1 Measurements

Because only 2 counters and complimentary asynchronous events need to be configured, the performance of the TE application can be deduced from the general SCAMPI performance measurement.

6.6 End-to-end QoS Monitoring

6.6.1 Analysis

Quality of Service monitoring analyzes 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). This section describes the design and implementation of an architecture for such a scalable end-to-end QoS monitor. The monitor provides QoS statistics such as delay, jitter and packet loss (which reflects throughput), without revealing any sensitive network topology information to the outside world.

![Architecture of the QoS monitor](image)

Figure 6.6: Architecture of the QoS monitor

The integration of the monitor with the SCAMPI framework is illustrated in figure 6.6. A detailed overview of the application is given in SCAMPI deliverable D2.3. The QoS monitoring application allows to measure the packet loss, delay and jitter. By means of a small extension, also the used bandwidth can be measured. This can be done by including a “bytecounter” to measure the traffic to/from every access point.

When monitoring the end-to-end QoS of a network, monitoring informations needs to be synchronised in order to deduce the required statistics. Synchronising this data doesn’t only require processing power, but bandwidth as well. All local databases need to write their data to a synchronised database, which means that there should be a network between the databases. One solution is to use the existing network, i.e. the one where the QoS is measured from. The drawback here is that the additional traffic caused by the monitoring synchronisation can have an influence on the QoS of the network. By sending too much monitoring data over the network, the load will increase. To avoid this, a dedicated monitoring network can be used. Now the monitoring data won’t send any data on the existing network, resulting in a more accurate measurement of the current QoS. This approach

scampi@ist-scampi.org 76 January 12, 2005
however requires an additional network, which is in most cases not available. A possible solution is to increase the synchronisation interval or delay the synchronisation until after the QoS measurements. Of course, this implies that the QoS statistics will not be generated real-time.

In the current implementation of the QoS measurement application, each SCAMPI monitor is configured locally. In order to make the configuration of all QoS monitors more scalable, the implementation should be ported to a distributed MAPI. This way, each QoS monitor can be configured or updated from a centralised control server.

### 6.6.2 Measurements

Before using the QoS monitor for live measurements, some experiments were done in a controlled environment. These tests will validate the correctness of the measured QoS characteristics by introducing artificial delay and loss in a test set-up. To verify the measured QoS characteristics, they were compared to the results obtained by a Smartbits [sma] network performance analyzer. The traffic generated by the Smartbits contains random payload (256 bytes). This allows the QoS monitor to compute a unique hash over the individual packets. For each test a 100Mbit stream was generated during a period of 5 minutes.

Figure 6.7 shows the measured delay by both the QoS monitor (5% sampling rate) and Smartbits, subject to a variable introduced delay. The introduced delay is progressively increased from 0 to 0.5ms. The delay measured by the Smartbits is generally 0.12ms higher than the one obtained by the QoS monitor. This is due to the extra delay between the Smartbits hardware and the test set-up. Ignoring this additional constant delay, the measured delays of the QoS monitor are very close to the Smartbits results (maximum deviation of 2.8%).

Figure 6.7: Measured packet delay

Figure 6.8: Measured packet loss

In figure 6.8, packet loss was generated by an impairment node. Using a probabilistic dropper, the loss was progressively raised from 0 to 100%. The measurements with the Smartbits system show the correct packet loss. Using the QoS monitor, we try to approximate this correct result by choosing an appropriate packet sampling rate. Comparing the results to the previously measured ac-
CHAPTER 6. APPLICATIONS

accuracy of the delay measurements, the packet loss deviates much more from the correct result. This is because, in case of the delay measurements, each sampled packet contributes to the average delay. Even though there is a high delay variation, the average will stay about the same. This however is not the case with packet loss. A subset of sampled packets does not necessarily contain the same number of dropped packets as the entire flow. For a sampling rate of 1%, we obtain very poor results, 5% is much better, while 10% already gives a good approximation of the real packet loss. In the live QoS measurements, a sampling rate of 10% will be chosen.

6.7 Summary

A fairly extensive set of applications has been built within the project’s lifetime. Where the applications were performance critical, they benefitted from the efficiency of the Mapid approach.

However, besides the performance benefits, all the application programmers commented on the fact that the MAPI was more convenient for writing their applications. Where they used to write long and complex procedures for capturing and processing network traffic, by virtue of employing the MAPI developed in the SCAMPI project, they were often able to reduce the code to a handful of lines.

In the next chapter, we will look at the results one last time and assess whether they live up to the expectations.
Chapter 7

Results

In this chapter, we assess the analytical and experimental results discussed in the previous chapters as well as in deliverable D3.4. The aim is to come to an evaluation of the overall system.

7.1 Analysis of the system architecture

In the SCAMPI project a large number of efforts in many different directions have been combined to produce a monitoring platform that is truly versatile, truly scalable and truly safe. As a result, the consortium members have conducted research in many areas and the following list just mentions a few of them that appeared in various publications:

- hardware design:
  - various versions of high-speed monitoring boards have been designed, implemented and tested;
  - various related daughterboards have been designed, implemented and tested (e.g., the timestamp unit);

- programming of programmable network cards:
  - different monitoring functions use features provided by the SCAMPI card (e.g., counters, statistics, filters);
  - we ported a full packet language compiler and various other tools to implement the SCAMPI platform to the Intel IXP1200 network processor [NdBCB04];
  - intrusion detection algorithms on programmable network cards (e.g., using Intel network processors [BH04], FPGAs and CAMs);

1Some things that were developed for the IXP, like the OKE on IXPds [BS03] were eventually not retained in the SCAMPI distribution.
CHAPTER 7. RESULTS

- development of new string matching algorithms (e.g., $E_2X^b$);

- development new packet processing languages (e.g., FPL-1 [BP04] and FPL-2 [CB04, BdBC+04]);

- development of multiple packet filtering optimisation techniques ([CSdB+04]).

- new applications to monitor large network infrastructures at various levels of granularity in the time, protocol and topology domains (e.g., ‘Stager’$^2$);

- anomaly detection algorithms for discovering SYN flooding.

- minimising packet copying and context switching when packets are passed to userspace.

- security and trust (e.g., by means of an independent authorisation daemon [PB04]).

Besides “pure” research and dissemination, there have been important efforts in other areas as well:

- various high-level monitoring application in the fields of security, traffic engineering, accounting, etc.;

- contributions to standards bodies (e.g., IPFIX);

- manageability of the system (e.g., by implementation of a BIB);

Despite the large number of areas in which the consortium has been active, it has managed to produce a single, consistent implementation of a monitoring platform in which all these efforts come together. Rather than defining the SCAMPI architecture as all the code produced in this project, we will define the architecture as: “the software running on the network card, possibly the kernel and certainly Mapid that contributes to providing client applications with information about the network traffic”. Note that this is a fairly narrow definition that excludes the applications and the hardware itself. Applications and hardware have been evaluated in previous chapters directly.

The part of the SCAMPI architecture in which we are mainly interested is the code that processes the packets, from the point of arrival in the network card, to the point of consumption by the application. In other words, our main interest is in the datapath. However, we will also consider briefly security, extensibility and manageability aspects. For an overview of the overall system including both hardware and applications the reader is referred to Section 3.1.

$^2$http://stager.uninett.no

scampi@ist-scampi.org 80 January 12, 2005
7.1. ANALYSIS OF THE SYSTEM ARCHITECTURE

7.1.1 Context

To assess the efficiency and usefulness of the SCAMPI architecture, we need to bear in mind the context that is (sometimes implicitly) assumed by the SCAMPI consortium. We do so by summarising some of the important design principles used in the SCAMPI architecture:

1. Users of the platform should be able to use fairly inexpensive commodity PCs to monitor the network.
2. However, to process packets at really high speeds, we need support from the network card.
3. While the system design is geared for high-speed traffic monitoring using special-purpose network cards, one should also be able to use normal NICs, although in that case, many of the advantages of the design disappear.
4. Multiple, concurrent monitoring application should be supported.
5. Packet copying and context switching should be minimised.

The implication of points (1–3) is that while SCAMPI will attempt to make monitoring with traditional NICs better, it will not necessarily make it much faster. For instance, if a NIC needs to execute an interrupt handler in the kernel for every packet that is received, this will generate considerable overhead in terms of context switching, especially since all the packet processing functions are executed in userspace. In other words, the assumption is that for high link rates, network cards are able to bypass the kernel by storing packets directly in an application’s address space.

As the system should support multiple, concurrent monitoring applications, point (5) implies that most of the processing takes place in a single entity that can be run almost constantly. In the case of the SCAMPI architecture that entity is embodied by Mapid.

7.1.2 Issues for analysis

To assess the efficiency and usefulness of the SCAMPI architecture, we need to determine what are the most important aspects that should be considered. For the SCAMPI architecture, we will consider the following aspects in detail:

- packet and result copying,
- context switching,
- security, access and resource control,
- usability and support for legacy applications,
CHAPTER 7. RESULTS

- extensibility and scaleability,
- portability.

Each of these points will be considered in some detail below.

Copying  We analyse copying behaviour in a bottom-up approach. In other words, we start by considering the way packets are handled by the card and continue until the data arrives in the end application.

Packets arrive at the network card and need to be processed. In the SCAMPI architecture the network card is assumed to be capable of handling some of the processing independently. This means that packets may, for instance, be filtered, and various statistics may be derived from the traffic, without involvement from the host processor. In this sense, if packets are not needed themselves by the end application may be dropped in the card, rather than travel across the PCI bus to host memory. In many application domains, this severely reduces the load on both bus and memory bandwidth. For instance, applications that are interested only in statistics (e.g., for accounting or traffic engineering purposes), will never see the packets at all. No copying takes place. Instead, statistics are periodically read from the card by the userspace application.

If a (part of a) packet is needed by the code running on the host processor a decision is needed as to how to deliver the packet to the host. The same is true for results (e.g., counters) that are stored on the network card. There are various options: the data may be left on the card (meaning that the host needs to access the data across the PCI bus), or the data may be copied explicitly. While true zero copy solutions sound attractive, in many practical scenarios they are less efficient than solutions that perform at least a single copy (see [BdBC+04]).

In the default behaviour of the SCAMPI network card, this is exactly what happens to the packets: if they are considered interesting, they are explicitly copied to host memory (if needed, they may be truncated, e.g., to capture only the header). We call this a copy-once solution. The only scenario in which copy-once is inferior to zero-copy is when most processing occurs on the card, and the accesses on the host processor are few and fast compared to copying the entire packet. This is rare.

On the host, the packet is immediately in the address space of Mapid. Without further copies, the different functions that make up an application’s flow may be applied to the packet. Indeed, it is quite permissible to allow functions corresponding to multiple flows and multiple applications to access the same packets. In other words, there is no copying at this stage.

Finally, to get the results to the end applications, there are two possibilities that are explicitly supported. An application may either receive a pointer to the result (or packet), or it may receive the result directly. The two options correspond to pass-by-reference and pass-by-value in a programming language’s procedure

---

3On the other hand, as the entire card is programmable, it is possible to change this behaviour to zero copy at some point in the future.
7.1. ANALYSIS OF THE SYSTEM ARCHITECTURE

calls and have the same copying semantics. It is anticipated that references will
normally be used for larger structures, while values will be used for smaller results
(e.g., counters).

Note that it is not precluded that an application is passed a direct pointer to a
memory area on the network card itself. In this case, it may read results (and/or
packets) directly, without involvement of the Mapid.

In case an external adapter, such as a JUNIPER router with filtering capabili-
ties, is connected to the monitoring platform, there is always an extra copy for all
packets that pass the filter. In fact, everything that is sent by the router is simply
forwarded to the monitoring host, where it is picked up in the usual way.

For less sophisticated NICs, we are not able to reap the same benefits regard-
ing copying (and context switching), as these cards always involve both a copy
and kernel involvement. The SCAMPI consortium has made several contribution
to Linux kernel code that reduce copying and processing for such NICs (e.g., by
memory mapping the buffers and bypassing the protocol stack), but these solutions
are not particular to the SCAMPI approach and will benefit other methods also.

We conclude, for now, that copying in the SCAMPI architecture can be brought
down to the bare minimum.

**Context switching** Like copying, context switching is reduced by two features
of the SCAMPI architecture: (1) the ability to perform complex processing in the
card, and (2) the single point at which SCAMPI functions are applied to the traffic
stream (Mapid).

If packets can be fully handled by the card (e.g., by making it produce periodic
statistics), there is no need for either Mapid or the end applications to be scheduled
very frequently. An occasional wake-up to read the results would suffice.

Similarly, in a scenario where multiple applications open network flows that
all require some functions to be executed on specific packets, doing so in Mapid
reduces context switching for the same reason. For instance, Mapid executes all the
functions for all flows on a single copy of the packet, without requiring a switch
to another process for each flow. Again, in case the application is not interested
in most (or any) of the packets, it need not be scheduled for each of them. A
less-frequent wake-up to obtain just the results in which the packet is interested is
sufficient.

We conclude that it is the ability to perform complex processing at the low-
est levels of the processing hierarchy (network card and Mapid) that reduces the
number of context switching that is needed.

**Security, access and resources** Most existing network monitoring solutions pro-
vide no security other than the fact that only the root user is allowed to do this. This
is very restrictive and quite unnecessary. In addition, it is also limited in terms of
resource safety, as a root user is allowed to do anything (including running code
that overloads the resources). In SCAMPI we have taken a more flexible approach
to security, access and resource control.

In contrast to most traditional monitoring solutions\(^4\), the SCAMPI architecture does not restrict monitoring to ‘superusers’. Rather, anyone is permitted to connect to the Mapid, provided they have the appropriate credentials. Security and access control are protected with the combined strength of the encryption algorithms (in the current release we are using: RSA, SHA and the OpenSSL library) and the trust server (for which we have used a KeyNote backend [BFIK99]).

The authorisation daemon \texttt{Authd} permits administrators to regulate access to resources in a much more fine-grained manner than the all-or-nothing paradigm used in most existing solutions [PB04]. For instance, it is possible to enforce a policy that allows certain users to monitor only the traffic that has first passed an initial filter (e.g., a filter that passes only specific IP addresses). Moreover, the authorisation policies can be used to guard against ‘silly mistakes’, e.g., by preventing the instantiation of a flow that attempts to apply a very computationally expensive function on every packet on a high-speed link (instead, it may ordain that said function be only allowed after sampling has been applied).

Moreover, \texttt{Authd} explicitly allows for policies to deal with resource consumption. For instance, one may specify how much resources certain functions take in specific domain, and for each resource specify explicitly the maximum capacity.

\textbf{Usability and support for legacy applications}   Throughout the deliverables, the consortium has shown examples of code that illustrate that certain things that would very hard to do on top of existing network monitoring tools, require just a few lines of code using the MAPI (see D3.4, appendices A–C for examples). We therefore believe that the programming side of usability (using the MAPI library) is well-catered to.

Similarly, by porting the \texttt{pcap} library to the SCAMPI architecture, the consortium has ensured backward compatibility with a host of legacy applications, including such long standing favourites as \texttt{tcpdump} [PaAO04]. We do believe that more work here is needed, as not a small number of applications is built on different APIs. A well-known example is \texttt{libipq}, found in combination with the Linux netfilter framework\(^5\). Some efficient versions of \texttt{snort} [Roe99] and indeed most of the \texttt{iptables} applications use this or similar libraries. To run such applications without modification requires implementing the libraries on top of the MAPI.

Finally, there is still a lack of commandline tools that exploit all the features offered by SCAMPI, in a way that resembles the way \texttt{tcpdump} harnesses most of the functionality offered by BPF. To persuade network administrators of the usefulness of the SCAMPI approach, such tools are instrumental. Phrased differently, we should not require network administrators to become programmers in order to monitor their networks.

\(^4\) a notable exception being the original packet filter on ULTRIX [MRA87].

\(^5\) \url{http://www.netfilter.org}
7.2. ASSESSMENT OF THE MEASUREMENTS

**Extensibility and scalability** In Section 3.2 we have explained how new functions and new hardware can be added to the system. Any vendor with a new type of network card is able to export the specific features of the card in the SCAMPI context, and these features can subsequently be used by application, just like the functions in the standard SCAMPI function library. Similarly, programmers that implement a new function in software that meets with approval from the administrator can add the function in a library that is linked against by Mapid. In this way, client any application can benefit from the fruits of other people in the field of network monitoring, maximising code reuse.

While having this consistent and open view makes the SCAMPI architecture quite amenable to the addition of features and/or hardware within the architecture, extending the architecture itself requires significant work. For instance, to change the way flows are defined as the application of ‘linear lists of functions’ to one in which there can be an arbitrary flow graph (as found, for instance, in FFPF), requires a major code rewrite. Since the ability to apply a list of consecutive functions to a stream of traffic at the lowest possible level is already a major leap forward compared to existing solution, we do not expect this to be a major problem in the near future.

**Portability** Within the lifetime of the project, we have focused mainly on a single implementation of the SCAMPI architecture, centered around Mapid. The reasons for doing so, concern mainly time constraints, and the fact that SCAMPI is intended as a research project, albeit one that is intended and expected to be actually used by various organisations by the time the project is completed.

Nevertheless, we have demonstrated in FFPF that the architecture is portable both to different hardware and indeed to a different implementation [BdBC+04]. In our opinion, this is a useful feature, as it allows faster and better implementations to be developed in the future, which would permit users to select the most appropriate one.

### 7.2  Assessment of the measurements

**This section describes the results as measured and described in D3.4v2.**

The hardware developed within the SCAMPI project is currently not yet in fully mature state. This is no surprise as the hardware development was delayed by the replacement of the original manufacturer by a new one. Moreover, the development cycle for hardware is longer than that of the software, partly we are operating at the cutting edge of technology, with components that are only available as evaluation samples, and some incur long delays before they arrive. Even so, within the project’s timeframe the consortium has built both a 1 Gbps and a 10 Gbps SCAMPI card. While neither of these cards has seen many cycles of redesign and improvement, and for this reason do not yet perform at the level that is
expected for new versions of the cards, they contain all the features required by the SCAMPI architecture and show how high-speed links can be monitored.

7.2.1 The SCAMPI card as regular NIC

First we consider the simplest of all tests (described in Section 2.2 of D3.4) in which a packet is sent straight to userspace without any processing (i.e., without exploiting any of the advanced features supported by the card). In this case the SCAMPI card contains firmware that makes it function like a regular NIC (and therefore incurs kernel overhead). We observe that the 1 Gbps card is able to handle more packets than a mature Intel adapter, before packet loss is incurred. On the other hand, we also note that the SCAMPI card performs less well than the Intel card in terms of the loss percentage from 50,000 packets onward. A peculiar aspect of the Intel experiments is that the drop rate for small packets first increases (as expected), but then suddenly drops to very low values between 50,000 and 100,000 packets per second. After that, the packet loss ratio suddenly explodes to almost 60%. A possible (very speculative) explanation for this phenomenon may be that at some point Intel processes multiple packets per interrupt and uses polling in between interrupts. If the rate goes up even higher, it can no longer cope.

Second, when the same configuration is used for IP header filtering (D3.4, Section 2.3), we observe that the performance does not differ significantly from the results without filtering. Indeed, for a rate of 75,000 packets, the drop rate with filtering was significantly less than without filtering. This can be explained by the newer firmware that was installed on the network card in this second test. This also demonstrates that in the beginning the performance of these cards improves rapidly with each generation. The fact that performance deteriorates quickly as the percentage of packets passing through the filter increases, suggest that the bottleneck resides in the code that passes the packets to the applications. In this scenario, the SCAMPI card functions as a regular NIC, none of the specific SCAMPI features that reduce copying are employed.

In a subsequent test with a new version of the firmware (D3.4, Section 2.4) shows that even without filtering we are able to handle high packet rates but that the drop rates are still considerably worse than those of an Intel card. To our surprise, the test also showed that at high rates the MAPI overhead is significant. One plausible reason is that in an overloaded system, processing a set of packets in an application (as performed by the MAPI application) immediately causes a large number of new packets to be dropped. Newer firmware which requires less processing time may fix this. Similarly, it could be caused by careless sleeps (if the application sleeps too long packets are dropped before the application wakes up and processes them. Better tuning of application and buffer size may well remedy the problem.
7.2. ASSESSMENT OF THE MEASUREMENTS

7.2.2 The SCAMPI card as SCAMPI card

The more interesting scenarios concern the use of the advanced features of the SCAMPI card. For instance, in D3.4, Section 2.4, we show that filtering on the network card has a high pay-off. By filtering packets and alleviating the burden on the PCI bus and host memory, we were able to sustain significantly higher link rates with reasonable packet loss compared to the tests in which no filters were applied (e.g., in case 10% of the traffic passes through the filter, we were able to sustain 500,000 packets per second with just 0.004% loss). Even though the full system is only in its teething age, we believe that these results show that the SCAMPI approach is the right one. As network speed increases faster than bus/memory speed, the ability to handle packets before they hit the bus will become increasingly important. For large packets we sustain traffic rates that approach link speed (up to 900 Mbps), while with small packets we sustained roughly 250 Mbps with negligible packet loss. While these results are encouraging for a first batch of cards, we realise that significant tuning is needed to really handle full link rate.

7.2.3 Overhead of the architecture

Perhaps more telling than the capabilities and limitations of the hardware is the overhead incurred by the SCAMPI architecture itself. In D3.4, Section 4 we described several experiments to evaluate the intrinsic costs of the framework. These will be summarised below. The overhead includes the overhead of the Performance API (PAPI) except where explicitly stated.

The results show that the overhead of the SCAMPI architecture on the SCAMPI card is comparable to that on a regular Intel card in terms of cycles (although the overhead in number of instructions is sometimes a little bit less). An important reason for this is that with the current version of the firmware all packet processing takes place in software, rather than on the card. In the next version of the firmware, this should be fixed. It was also shown that the overhead of getting a result off a network card across the PCI bus is much more expensive than getting the result from host memory (the difference is approximately an order of magnitude).

In existing solutions to network monitoring (pcap in combination with a regular NIC), it has been observed that most of the overhead is incurred by: (1) getting the packet to the host (up to thousands of cycles), (2) executing the BPF filter (thousands of cycles), and (3) pushing the packet to user space (ten thousand cycles). SCAMPI will help reduce the 1st and 3rd causes of overhead, while implementation of filtering on the SCAMPI card will even reduce the 2nd cause of overhead.

This is supported by efforts in related cards. E.g., in DAG cards, the overhead before a function on the host processor can be applied to a packet is just a few hundred cycles (e.g., less than 300 when measured using MAPI on DAG cards). Moreover, there now also exist DAG coprocessor cards with IP filtering on board. This is similar to , albeit less generic than, the capabilities found on the SCAMPI adapter. The DAG card also doesn’t support advanced features such as (for exam-
CHAPTER 7. RESULTS

ple) computation of statistics in the hardware.

Again, the conclusion is that the SCAMPI adapter still has teething problems. Even though it currently does not outperform the Intel adapter, its performance is also not significantly worse. This is encouraging as these results were obtained with a version of the firmware that did not yet allow for any complex packet processing on the card. With hardware filtration we can handle more packets than the Intel NIC, so we see that hardware acceleration is indeed promising.

7.2.4 The applications

In this section, we evaluate briefly the results of just a few of the applications and tools that were developed on top of the MAPI. They are chosen because they stand to benefit most from the architecture and allow for assessment of the architecture in terms of performance.

7.2.4.1 Network intrusion detection

The sids tool for network intrusion detection show that MAPI is quite suitable for building a network intrusion detection system that involves both header and payload inspection. As the MAPI allows the number of packets that are sent to userspace for further inspection to be limited to the number of packets whose headers indicate that they are vulnerable to attacks, sids is able to sustain 100 Mbps (almost) irrespective of the number of rules, as long as the number of payload scans is relatively low. This would be difficult to achieve with existing solutions.

On top of the sids base, ACID, a web-based tool written in PHP, displays the results of the monitoring activity. As a result, the sids tool is an easy-to-use and rather efficient solution for monitoring a relatively low-speed network for intrusions. A final attractive feature of sids is that it is able to work with the existing and continuously updated ruleset of the well-known snort tool.

In addition a new pattern matching algorithm has been developed that has come to be known as Piranha. The algorithm combines several desirable properties. It is fast, has a small memory footprint and its runtime component is very simple. This suggests that it may be possible to execute the entire algorithm on a network card, supporting our claim that a programmable network card, such as the SCAMPI adapter will be beneficial for high-speed packet processing.

The claim is also supported by a recent implementation of a network intrusion detection and prevention system on a IXP1200-based network card, known as NPIDS [BH04]. NPIDS is able to sustain the full 100 Mbps link rate under worst-case assumptions: every packet was TCP and required full payload inspection. While the algorithm requires more space than Piranha and the system is not based on the SCAMPI architecture, it validates the approach in which more and more processing tasks are pushed to the lower levels of the processing hierarchy. Moreover, NPIDS is a solution to one very specific problem, while the SCAMPI architecture allows the card to be used for different purposes simultaneously.
In summary, we conclude that network intrusion detection is becoming increasingly important and would benefit significantly from approaches such as SCAMPI that bring processing closer to the wire.

### 7.2.4.2 QoS monitoring

In these experiments, the SCAMPI architecture is employed to perform QoS monitoring. For the purpose of this deliverable they are intended to show that the SCAMPI architecture is suitable for this application domain. The results indicated that this was indeed the case, both for those measurements that may include sampling (e.g., monitoring packet loss, delay and jitter) and those that do not (e.g., throughput measurements).

As an aside, the results show that one should be careful to choose an appropriate sampling rate to be able to obtain statistics that match actual traffic characteristics. This, of course, is a simple operation with the MAPI which has several configurable built-in sampling functions.

### 7.2.4.3 Traffic Engineering application

The traffic engineering application uses the MAPI threshold function to signal overloaded network interfaces. Once such an interface is detected, the traffic engineering application will dynamically redistribute the load in order to omit the overloaded link. The signaling by the MAPI is not part of the control path of the traffic engineering applications, i.e. the mapping of flows to paths does not have to wait for a MAPI signal. Compared to the time in which flows are requested by different applications (order of seconds), the detection and signaling of over/under utilisation of interfaces by the MAPI is very fast (order of microseconds). This way, the time critical part is not the detection by the MAPI, but the configuration of the flows by the TE controller.

In the case of the TE application, no intervention of the user is needed. In other applications, it might be needed to raise an alert to the system administrator when an event occurs, e.g. by sending an email. Obviously, the time it takes to check the threshold will be negligible compared to the time it takes to send an email.

### 7.2.4.4 Stager

The Stager application demonstrates that the SCAMPI architecture for handling high-volume netflow reports even with a large number of reports and millions of table entries. The system is quite capable of collecting Netflow data from 23 routers with a total of 227 interfaces with speeds ranging between 155Mbps and 10Gbps. The system is able to process more than 30GB of raw Netflow data every hour.
CHAPTER 7. RESULTS

7.3 Summary

In summary, we believe that the architecture is strong and the design decisions that were taken are right. Most of the issues that existed three years ago and sparked these decisions are equally true if not more so. A notable exception is the speed of the bus, which has received a significant boost.

As for the performance of the components, we conclude that the applications and middleware are well on target. It is fair to say that the hardware does not quite perform up to its full potential. On the other hand, it is likely that this will be fixed in the very near future. At that point, the full architecture provides a very fast and very flexible monitoring platform that scales well with future line rates.

In the next and final chapter, we compare the SCAMPI results both with the original objectives for the project and with alternative approaches.
Chapter 8

Comparison with objectives and competition

We conclude this deliverable with a chapter that evaluates the results by comparison. It consists of two parts. First we look at how well we have met the original objectives. Second, we compare our results and architecture with the competition: which bits remain relevant and which bits have been overtaken by new developments.

8.1 Comparison against objectives

This section is devoted to what has been achieved in the project. We will compare that to the original objectives, listed in Chapter 2, as well as to alternative approaches. We believe that the overall objective, “to build a monitoring platform for backbone links that provides flexibility and scales with future link rates”, has been largely achieved. It may be argued that the current implementation of the SCAMPI adapter still experiences growing pains and is not yet performing at its full potential. Even here, however, we are convinced that the design shows great promise.

Three years down the line and despite major developments in PCI bus technology, the consortium is convinced that the need to process packets before they hit a host’s bus, is as valid as it was three years ago, if not more so. The reason is that even if PCI will be fast enough to keep up with line speed in the future (which remains to be seen), this certainly is not the case for memory. As more complex packet processing is needed for many purposes (e.g., security) and network speed increases at a rate beyond Moore’s law, processing as close to the hardware as possible and exploiting parallelism are both important methods to cope with link rates.

As most of the objectives listed in Chapter 2 were achieved, we only highlight goals that warrant additional explanation.

Protocol analysis Some of the original requirements in D0.2 were not met. In particular, while D0.2 required SONET/SDH (OC-48 and OC-192), we currently
provide only Ethernet. As explained in Section 2, this is caused by the fact that
the original deliverable was based on the specifications of the 4Plus card. Since
then 4Plus has left the consortium and Cesnet and Masaryk have taken over the
development of the card and the driver software, with great experiences in the Eth-
ernet technologies. As the development of hardware was delayed and demand for
Ethernet technologies (especially 10GbE) has significantly increased (compared to
SONET/SDH) during 2003 we have made the decision to put all effort to Ethernet
technologies and leave other protocols as ‘optional’.

Even so, the transceivers on both the 1GbE and 10GbE interface cards are
placed in hot swap SFP(1GbE) and XFP(10GbE) cages. The test were succesfully
made using multimode 850nm, monomode 1310 and 1550 nm) as well as copper
transceivers on the 1GbE interface card. In the case of the 10GbE card both
multimode 850nm and monomode 1310nm transceivers were tested. The 10GbE
1550nm transceivers are not available on the market yet. The first generation of
COMBO interface cards is equipped with the phyter chips supporting 1GbE, re-
spectively 10GbE. The second generation (which is in manufacturing process now)
use Rocket IO blocks inside of XILINX FPGA for support of 1GbE (with Virtex II
PRO) and OC48, OC192, 10GbE (with Virtex II PRO-X) on link level.

The SCAMPI architecture has achieved the goals related to inspecting any type
of header and indeed the payload of packets. The consortium has made initial steps
to supporting IPv6, but most effort has been involved in IPv4 for practical reasons
(the testbeds for most consortium members were based on IPv4). The requirement
of 1 Gbps Ethernet has been achieved on top of several types of hardware (e.g., Intel
NICs, DAG card, SCAMPI adapter, and Intel IXP1200). The SCAMPI consortium
has included several different hashing algorithms and included a flow option to
determine how many bytes of a packet will be returned to the user.

**Monitoring requirements** It is assumed that a SCAMPI monitoring system will
run on a dedicated computer. The computer may be owned and managed by the
network operator, or by another party. Running the SCAMPI monitoring system is
the only task for this computer.

- An SNMP MIB was added to the network node for management purposes. It
  allows one to use SNMP to get hold of most useful information. In addition,
as the system stores management info in local repositories (e.g., /proc file
system, applications may access the info directly also. Although the original
objectives contained a requirement that information about system memory,
and disk usage alo be made available, the SCAMPI MIB contains no infor-
mation about these items. However, these are available via the normal Linux
system tools.

- An API for monitoring API, a network management API and an implemen-
tation of pcap have all been produced.
Regarding passive monitoring we required that the following must be supported:

- configuration of filtering and sampling on special purpose hardware when available;
- possibility to read all data in packet subsets;
- possible in MAPI to define how to separate network flows from packet subsets;
- reading flow records generated from packet subsets;
- both data captured in real time and previously captured data read from storage.

These has all been accomplished. Indeed, configuration of filtering and sampling on special hardware has been implemented on three different implementations: the SCAMPI adapter, the IXP1200 network card, and JUNIPER routers.

**Hardware requirements**  Again, we have implemented the MAPI on a fairly large number of interfaces, including regular NICs, DAG cards and different types of programmable network cards (SCAMPI adapter, IXP1200, and JUNIPER). The clock requirements were added to the SCAMPI adapter in the form of a special timestamping unit on a separate board. The implementation easily exceeds the requirements listed in Chapter 2.

**Application requirements**  Within SCAMPI several applications have been developed. These include a Netflow/IPFIX probe, a network intrusion detection tool, an application for QoS monitoring which correlates traffic captured at two sites, a tool that aggregates and displays traffic statistics from multiple nodes and at configurable granularity, an accounting application, a threshold-alerting application which aims to create flow alarm notifications based on thresholds violations and various others.

- QoS Monitoring:
  - SCAMPI must provide provisions for correlating packets captured by two different SCAMPI platforms, such as precise timestamps and packet signatures for passive measurements or sequence numbers for active measurements; based on these provisions, the QoS monitoring application MUST be able to correlate packets from two different SCAMPI platforms. The correlation in the SCAMPI QoS monitoring application is done throughout a centralised SQL database. All participating sites write their measured information (packet summary and timestamp) to this central database or to local databases which are merged periodically. For the correlation, packets can be identified by their TCP sequence number of by a computed hash over the content.
CHAPTER 8. COMPARISON WITH OBJECTIVES AND COMPETITION

- built-in knowledge of IPv4, IPv6, UDP, TCP and RTP protocols needed to perform monitoring of QoS characteristics related to these network protocols. The SCAMPI QoS monitoring application supports all protocols that are covered by the SCAMPI system.

- generic way of supporting other protocols (e.g., by using byte offsets, etc.); With some minor adjustments to extract the required header information out the packets, other protocols can be supported by the QoS monitoring application.

- ability of user to configure rules to specify a flow to be monitored, location of measurement points in case of two-point measurement, QoS characteristics to be monitored including parameters such as precision, time granularity, etc. All previous options can be configured in the SCAMPI QoS monitoring application.

- Threshold Alerting:
  - ability to receive notifications via email. Once a threshold is reached, it is up to the application to handle the processing of this event, e.g. by sending an email to the system administrator. In the traffic engineering application, thresholds are used to dynamically reconfigure a loadbalancer. Sending an email when a threshold is reached would be trivial to add.

In addition, several other application have built, including:

- A netflow collector and analyzer. It has the look and feel of the ‘Stager’ tool, but it is able to do the inverse: rather than aggregate the data in an entire network, it is able to track traffic to and from specific hosts, services, etc.

8.2 Comparison with competitors

8.2.1 Normal 10Gig network card with PCI express board

Several manufactures now provide NICs for 10 Gigabit Ethernet that plug in a regular PC. Particularly these adapters can be obtained from Intel, S2io and Chelsio Communications. They are available with fixed 850 nm or 1310 nm transceivers. None of these adapters allows to use interchangeable transceivers (as available in the Combo6 cards). The adapters are designed for PCI-X bus operating at 64-bits and 133 MHz. Therefore, they can transfer a higher volume of data to the host computer than the current version of Combo6 mainboard, which has a PCI bus operating at 32-bits and 32 MHz. However, all these adapters provide only packet receiving and sending. Some of these adapters support computing packet checksum or the whole TCP implementation in hardware. In contrast, the Combo6 card uses an FPGA to implement smart data processing (header filtering, sampling,
8.2. COMPARISON WITH COMPETITORS

statistics and payload searching) and pass only selected packets or statistics to the host computer.

8.2.2 DAG with coprocessor

When the SCAMPI project started the available DAG cards from Endace did not have any processing capabilities on the card. They were only designed for highly optimized packet capture so that packets could be processed in realtime or stored to disk.

Since then Endace has released a new series of cards which supports an optional coprocessor card\(^1\) which adds processing capabilities. The coprocessor contains an FPGA, 128 MB DRAM, 4MB SRAM and a 2Mb CAM. Firmware is available for doing ATM packet reassembly, IP packet header classification and string search.

For the TCP/IP header classification, the coprocessor supports up to 16384 simultaneous rules and around 2000 payload search strings.

The new DAG cards together with the coprocessor is very similar to the SCAMPI adapter. The DAG coprocessor has somewhat weaker hardware specifications, but offers some of the same features. However, it doesn’t support advance hardware features such as computation of statistic on the board. In addition as the DAG firmware is closed, it is very hard to extend the functionality.

8.2.3 IXP28xx

The Intel IXP28xx is a network processor for 10 Gbps links that shares quite a few characteristics with the SCAMPI hardware\(^2\). The network processor contains on-chip a fairly large number (16) multi-threaded RISC cores that handle the incoming traffic, by sharing the load. They also contain small amounts of fast on-chip memory and interfaces to external memory and a 10 Gbps network link. The IXP28xx is a family of processors. Specific versions have specific coprocessors on-board that reflect the application domain (e.g., the IXP2850 has a number of hardware assists for encryption and security purposes).

The main differences between the SCAMPI card and the IXP28xx is that SCAMPI has chosen for an FPGA design, while the Intel cards are based on multiple ‘normal’ RISC processors. The advantage of the latter is that it is easier to program. The advantage of the former is that many high-speed tasks can be performed in hardware. In the future, we may well see that both approaches merge, as is already witnessed on the one hand by the fact that small processors are realised on the SCAMPI FPGA (and even more so on the new generation of FPGAs, as the VIRTEX II PRO family is equiped with one or more PowerPC processors),and on the other by the inclusion of special-purpose hardware on the IXP network processors.

\(^1\)http://www.endace.com/dagCoPro.htm
\(^2\)www.intel.com/design/network/products/npfamily/ixp2800.htm
In summary, the IXP28xx is an interesting competitor that seems quite suitable for implementing the SCAMPI architecture. Indeed, as we have already implemented the MAPI on the IXP1200, the port to the IXP2800 should be straightforward. We expect to realise an implementation in the near future. The main disadvantage of the IXP is the price, which currently comes to almost twice that of the SCAMPI card.

8.2.4 FFPF

It should come as no surprise that the project that is most closely related to the SCAMPI architecture is the Fairly Fast Packet Filter (FFPF) framework currently maintained as a SourceForge project by researchers at the Vrije Universiteit Amsterdam\(^3\). After all, FFPF was started by one of the SCAMPI consortium members and used to try out new ideas. Because FFPF is a little more lightweight and especially because it has been maintained by a single partner for a long time, it has been easier to completely change the design and introduce new features to the existing system. Several of these ideas, once proved useful, were subsequently adopted in the SCAMPI architecture.

However, FFPF also differs from the SCAMPI architecture in some important aspects:

**Kernel approach.** Unlike the SCAMPI architecture in which functions are executed on the network card or in the Mapid, FFPF explicitly includes kernel processing. In fact, the system started as a kernel/network card approach and the userspace component was added more recently.

**Limited platform support.** So far, FFPF has limited itself to regular NICs and Intel network processors, while the SCAMPI architecture had targeted more diverse hardware.

**Complex flow graphs.** Rather than the SCAMPI architecture’s linear list of functions that are applied in sequence to the traffic stream, as supported by the SCAMPI architecture, FFPF allows complex flow graphs to be built. For instance, such flow graph may take the shape of trees, or other complex graphs. Moreover, if two flows share a prefix of packet processing functions, that prefix will be executed only once.

**Simplified interaction with applications.** The way communication between application and packet processing functions takes places (e.g., passing results to the application, or configuration parameters to the function) takes a very simple form in FFPF. All communication is channelled through the three buffers: the packet buffer \(PBuf\), the index buffer \(IBuf\), and the persistent memory area \(MBuf\). Every packet processing function will provide these three buffers (although the size of \(MBuf\) may differ from function to function)

\(^3\)http://ffpf.sourceforge.net/
8.2. COMPARISON WITH COMPETITORS

and even be zero). In contrast, the SCAMPI architecture has defined more complex interaction models, with specific options for results that should be copies, and results that should be passed by reference.

However, as the MAPI has been ported to FFPF, this difference can be easily masked.

**Multiple buffer management systems.** As explained earlier FFPF supports multiple ways of dealing with buffer overflow, e.g., slow reader preference (SRP) and fast reader preference (FRP), while the SCAMPI architecture supports only SRP.

**Management.** While the SCAMPI architecture has taken the requirements for production-grade deployment very seriously in terms of the manageability of the system, FFPF as a research system has focused mainly on the datapath. As a result, there is no equivalent to SCAMPI’s SNMP MIB in FFPF.

**Flow groups.** Applications in FFPF can be divided in flow groups. Members of the same flow group have the same access rights to the network traffic and, hence, may share buffers. Applications in different flow groups are considered mutually untrusting and will automatically use different packet buffers. Different access and security policies may be applied to different flow groups. SCAMPI lacks the notion of flow groups (all applications have the same rights).

While it may seem that the difference between a kernel-based approach and a userspace approach is very significant, we argue that this is not the case, as long as the hardware is able to deliver the packets directly to the applications. As this is one of the assumptions of the SCAMPI architecture when applied to high-speed links, the only issue may well be that it is easier to develop userspace than kernel software. For regular NICs, however, FFPF has an advantage over SCAMPI, since now kernel involvement is required. For the SCAMPI architecture this involves passing packets up to userspace, while for FFPF this may involve all sorts of complex processing that reduces the number of context switches and packet copies. However, as the SCAMPI architecture is targetted mainly at high-speed networks beyond the capacity of regular NICs, this is hardly a disadvantage in SCAMPI’s application domain.

In summary, we argue that the SCAMPI architecture has made the right decisions for the purpose of developing a system that is stable and close to production-quality. For this goal it has had to pass on certain attractive (but arguably not essential in the near term) features, such as complex flow graphs and different buffer management schemes, that would add complexity and jeopardise the completion of a fully functional prototype within the project’s timeframe.
Appendix A

MAPI functions tutorial

A.1 Introduction

This is a tutorial that shows how to implement new functions in MAPI. It starts by giving detailed description of all structures and interfaces that can be used by functions. It then shows several examples on how some of the functions in the standard library is implemented, starting with a simple function like PKT_COUNTER and gradually moving towards more advanced functions.

A.2 Structures

This is an overview of the structures that are used by MAPI functions. Figure A.1 shows how the various structures are connected.

A.2.1 mapidflib_function_def

The mapidflib_function_def structure defines the static properties of a function and has the following entries:

char* libname  name of library that the function is part of.
char* name  name of the function
char* descr  description of the function
char* argdescr  description of the arguments that a function needs. Each character in the string specifies the type of an argument. The following types are supported:

s  string
i  integer
c  single character
A.2. STRUCTURES

1 unsigned long long
u UID of the application
p path of where the application is running

A function that takes two integers and a string as arguments, will then set argdescr to “iis”

char* devtype specifies the type of device that this function is compatible with. Valid device types are specified in the file mapidevices.h.

mapi_result_method_t restype specifies which method that should be used for returning results to the client application. Valid options are:

- MAPIRES_NONE the function do not return any results
- MAPIRES_IPC socket based IPC is used every time a result is sent from the MAPI daemon to the client application.
- MAPIRES_SHM results are returned using shared memory.
- MAPIRES_FUNCT function specific method. The function must implement its own method for returning results.

short modifies_pkts if the function modifies packets this should be set to 1. Support for functions that modifies packets is optional. If a function that modifies packets is applied to a flow in a MAPI system where support for it is turned off, an error message will be returned to the user application.

instance pointer to the instance interface of the function. See section A.3.1 for details.
init pointer to the init interface of the function. See section A.3.2 for details.
process pointer to the processing interface of the function. See section A.3.3 for details.
get_result pointer to the get_result interface. See section A.3.4 for details.
change_args pointer to the change_args interface. See section A.3.5 for details.
reset pointer to the reset interface. See section A.3.6 for details.
cleanup pointer to the cleanup interface. See section A.3.7 for details.
client_init pointer to the client_init interface. See section A.3.8 for details.
client_read_result pointer to the client_read_result interface. See section A.3.9 for details.
client_cleanup pointer to the client_cleanup interface. See section A.3.10 for details.
APPENDIX A. MAPI FUNCTIONS TUTORIAL

A.2.2 mapidflib_function_instance

The `mapidflib_function_instance` structure defines the dynamic properties of a running function.

- `int fd` flow descriptor of the flow that the function belongs to.
- `int fid` unique ID of the function
- `mapidflib_status_t status` current status of the function. Valid values are:
  - `MAPIFUNC_UNINIT` the function is uninitialized
  - `MAPIFUNC_INIT` the function has been initialized
- `mapiFunctArg args` arguments that were passed to the function
- `mapidflib_result_t result` structure that contains the results of the function. See section A.2.3 for more details.
- `void* internal_data` pointer to internal data. The function is free to use this as it wants.
- `mapid_hw_info_t *hwinfo` pointer to hardware specific information about the adapter the function is running on. See section A.2.5 for details.
- `unsigned long long pkts` number of packets that has been sent to the function.
- `unsigned long long processed_pkts` number of packets that has been processed by the function and passed on to the next.
- `mapidflib_function_def_t *def` pointer to a copy of the function definition. Since this is a copy, the definition can be changed by the function during the initialization.

A.2.3 mapidflib_result

The `mapidflib_result` structure contains information about the results of a function.

- `mapid_result_t info` information about results that are sent to the client.
- `void *data` pointer to memory that contains the results. Functions that need access to results from other functions, uses this pointer to read the results.
- `unsigned long data_size` size of the result data.

scampi@ist-scampi.org 102 January 12, 2005
A.2.4 mapi_result
Contains information about MAPI function results that is sent back to the client.

void *funct_res  pointer to function specific information
unsigned funct_res_size  size of the function specific information
mapid_shm  shm contains information about shared memory. Functions should normally not need to access this structure themselves as it is handled automatically by code in mapidlib.

A.2.5 mapid_hw_info
This structure contains information about a hardware adapter.

unsigned int link_type  the link type of the adapter.
unsigned int cap_length  the maximum length of a packet that can be captured by the adapter
unsigned long long pkts  number of packets captured by the adapter
char *devtype  type of device as defined in mapidevices.h
short offline  the value 1 indicates that the device is used for analyzing already captured trace files
int devid  unique ID of the adapter
void *adapterinfo  adapter specific information
flist ff_list  list of all active flows running on the adapter. Each entry in the list is another list that contains all functions belonging to the specific flow.

A.2.6 mapidflib_flow_mod
This structure contains various variables that can be used by a function to modify the behavior of a flow.

int reorder  used for reordering the execution order of functions in the flow. A function can set this to indicate that this function should be processed before the function that has a function ID equal to the the value of reorder.

int identical  a function can check already applied functions from the ff_list in the hw_info structure. If it finds an identical function it can set this variable to point to it. The internal functions in MAPId can then use this information to do global optimization.

scampi@ist-scampi.org  103  January 12, 2005
APPENDIX A. MAPI FUNCTIONS TUTORIAL

int *delete  pointer to a list of function IDs that can be deleted. This can for example be used by a function if one instance of this function can replace already applied functions.

int delete_size number of entries in the delete array.

mapid_add_function add_funct pointer to a function that can be used for adding new functions to the flow.

mapidlib_instance_t *mi pointer to the mapidlib instance. Used as an argument to mapid_add_function.

A.2.7 mapid_pkthdr

This structure contains information about a packet captured by a MAPI driver.

unsigned long long ts 64-bit timestamp of packet.

unsigned short ifindex interface index identifying which interface on the adapter that was used to capture the packet.

unsigned caplen the number of bytes that was captured

unsigned wlen the actual length of the packet including link layer header

A.3 MAPI function interfaces

This is a description of the various interfaces that can be implemented by a MAPI function. Many of the interfaces are optional.

A.3.1 instance

int instance(mapidlib_function_instance_t *instance,
             flist_t *flow_flist,
             mapidlib_flow_mod_t *flow_mod);

The instance interface is called when an application uses mapi_apply_function. This interface should do a syntax check of arguments that are passed to the function. The function can also use this interface to indicate similar functions, apply new functions or delete functions. No resources should be allocated in this interface.

A.3.1.1 Arguments

mapidlib_function_instance_t *instance pointer to the instance of the function. See section A.2.2

scampi@ist-scampi.org 104 January 12, 2005
A.3. MAPI FUNCTION INTERFACES

flist_t *flow_flist  list of functions already applied to the flow

mapidflib_flow_mod_t *flow_mod  pointer to the structure used for modifying the flow. See section A.2.6.

A.3.2 init

    int init(mapidflib_function_instance_t* instance,
              flist_t *flow_flist);

The init interface is called when an application uses mapi_connect and after the flow has been authorized. This interface allocates and initializes resources needed by the function.

A.3.2.1 Arguments

mapidflib_function_instance_t *instance  pointer to the instance of the function. See section A.2.2

flist_t *flow_flist  list of functions already applied to the flow

A.3.3 process

    int process(mapidflib_function_instance_t* instance,
                unsigned char* dev_pkt,
                unsigned char* link_pkt,
                mapid_pkthdr_t* pkt_head);

This is the interface where the actual processing of a packet takes place. If the function returns the value 1, then the next function applied to the MAPI flow will be called. The value 0 means that further processing of this flow should stop.

A.3.3.1 Arguments

mapidflib_function_instance_t *instance  pointer to the instance of the function. See section A.2.2

unsigned char *dev_pkt  pointer to the packet as captured by the device. This includes any device specific header.

unsigned char *link_pkt  pointer to the packet starting at the link layer header.

mapid_pkthdr_t *pkt_head  pointer to the packet header structure. See section A.2.7.
A.3.4 get_result

```c
int get_result(mapidflib_function_instance_t * instance,
               mapidflib_result_t **res);
```

This interface returns the results of this function to other functions or to the client application. MAPId has built in support for returning results through IPC or shared memory. This interface is only needed if these built in methods are not enough, for example if synchronization between MAPId and the client is needed or if results are read directly from the hardware adapter.

A.3.4.1 Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section A.2.2.

mapidflib_result_t **res pointer to the result structure that is used for returning information needed by others to get hold of the results. See section A.2.4.

A.3.5 change_args

```c
int change_args(mapidflib_function_instance_t * instance,
                 mapiFunctArg *fargs);
```

This interface is used for changing the arguments of an already running function. It is optional to implement support for this.

A.3.5.1 Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section A.2.2

mapiFunctArg *fargs pointer to the new function arguments.

A.3.6 reset

```c
int reset(mapidflib_function_instance_t * instance);
```

This interface is called by other functions and is used for resetting the results of the function.

A.3.6.1 Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section A.2.2
A.3.7 cleanup

    int cleanup(mapidflib_function_instance_t *instance);

This interface that is called when a flow is closed and should release all allocated resources.

A.3.7.1 Arguments

mapidflib_function_instance_t *instance  pointer to the instance of the function. See section A.2.2

A.3.8 client_init

    int client_init(mapidflib_function_instance_t *instance,
                    void* data);

This interface runs on the client side and is called the first time mapi_read_results is called. It is used for initializing function specific code that runs on the client side.

A.3.8.1 Arguments

mapidflib_function_instance_t *instance  pointer to the instance of the function. See section A.2.2

void *data  pointer to the function specific data that was returned by the get_result interface.

A.3.9 client_read_result

    int client_read_result(mapidflib_function_instance_t *instance,
                           mapi_result_t *res);

This interface is called each time mapi_read_results is used. It runs on the client side and is used for implementing function specific methods for returning results to the user application.

A.3.9.1 Arguments

mapidflib_function_instance_t *instance  pointer to the instance of the function. See section A.2.2

mapi_result_t *res  pointer to the result structure. See section A.2.4.
A.3.10 client_cleanup

    int client_cleanup(mapidflib_function_instance_t* instance);

This interface is called when the flow closes and should release all resources allocated by the function on the client side.

A.3.10.1 Arguments

mapidflib_function_instance_t *instance  pointer to the instance of the function. See section A.2.2

A.4 Tutorials

In this section we will demonstrate how to create new MAPI functions through a series of tutorials. All the functions are taken from the standard library, but during the tutorials we will create a new library, tutlib, to which we will add one function at the time.

When there exists two functions with the same name and for the same device type, the first one found will be used. To make sure that functions from the new library is chosen, one must specify the name of the library in the mapi_apply_function call like this:

    mapi_apply_function(fd, "tutlib:<function name>")

A.4.1 Simple function: PKT_COUNTER

The first function we will create is a simple packet counter. All this function does is to increment a counter each time the process interface is called. Results are sent to the client using shared memory.

A.4.1.1 Implementation

To start off we first have to create a new directory for the new library. In the main mapi directory do:

    mkdir tutlib
    cd tutlib

To create the new function we can use the function template located in the mapi main directory:

    cp ../funct_template.c pktcounter.c
In this template all the MAPI function interfaces are present. Most of these are not needed for the simple packet counter functions so the first thing to do is to remove them. For this function only process and reset are needed.

We will use the prefix `pktc` for each function in the source code, so we should replace the text `funct_name` with `pktc`. We are then left with only two interfaces like this:

```c
static int pktc_process(mapidflib_function_instance_t *instance,
                        const unsigned char* dev_pkt,
                        const unsigned char* link_pkt,
                        mapid_pkthdr_t* pkt_head)
{
    return 1;
}
static int pktc_reset(mapidflib_function_instance_t *instance)
{
    return 0;
}
```

We can now move on to the `mapidflib_function_def` structure that defines the function. The entries in this structure should be modified like this:

- **libname**: empty string. This is set by the library that includes the function
- **name**: The function name can be set to any value but in this example we will call the function `PKT_COUNTER`.
- **descr**: free text describing the function.
- **argdescr**: the function do not take any arguments so this should be an empty string.
- **devtype**: we will implement a generic software function that runs on top of all adapters. The devtype should therefore be set to `MAPI_DEVICE_ALL`.
- **restype**: results are returned by shared memory so this should be set to `MAPIRES_SHM`.
- **shm_size**: the function returns an unsigned long long counter so this should be set to `sizeof(unsigned long long)`.
- **modifies_pkts**: the function do not modify any packets so this should be set to 0.
- **process**: should be set to the address of the `pktc_process` interface
- **reset**: should be set to the address of the `pktc_reset` interface

scampi@ist-scampi.org 109 January 12, 2005
APPENDIX A. MAPI FUNCTIONS TUTORIAL

For the rest of the MAPI function interfaces in the structure, the value should be set to NULL since these functions do not exist.

With these modifications we get a structure looking like this:

```c
static mapidflib_function_def info=
    "", //libname
    "PKT_COUNTER", //name
    "Counts number of packets\n\nReturn value: unsigned long long", //descr
    "", //argdesc
    MAPI_DEVICE_ALL, //devtype
    MAPIRES_SHM, //Method for returning results
    sizeof(unsigned long long), //shm size
    0, //modifies
    NULL, //instance
    NULL, //init
    pktc_process, //process
    NULL, //get_result,
    NULL, //change_args
    pktc_reset, //reset
    NULL, //cleanup
    NULL, //client_init
    NULL, //client_read_result
    NULL //client_cleanup
};
```

At the end of the file there is also a function called get\_funct\_info. This is used by the library to return information from the function. The only change necessary for this function is to change the \_funct\_name to pktc so that the full name of the function becomes pktc\_funct\_info.

We can now start to look at the actual code for this packet counter MAPI function. Since we use shared memory, allocation and initializing the memory to 0 is already taken care of. In the processing interface all that is needed is to increment the unsigned long long counter in the shared memory. The location of this counter is pointed to by the data pointer in the result structure of the current instance. The code inside the pktc\_process interface should then simply be:

```c
(*((unsigned long long*)instance->result.data))++;
```

The code for the rest interface is similar and just reset the shared memory counter to zero:

```c
(*((unsigned long long*)instance->result.data))=0;
```

The entire source code for the packet counter function should then be:

scampi@ist-scampi.org 110 January 12, 2005
A.4. TUTORIALS

A.4.1.2 Function library

To be able to use this new MAPI function from an application, it must be included in a library. For this tutorial we will create a new library, tutlib, by using createlib.pl:

```bash
../createlib.pl tutlib tutlib.c
```

This will create a new library called `tutlib` in the file `tutlib.c`. All functions in the current directory will automatically be included in the library.

A.4.1.3 Makefile

We can now compile the new function and library. To do this it is best to create a `Makefile`. The contents of a `makefile` could look like this:

```bash
INCLUDE=-I. -I..
CFLAGS=-g -Wall -Wsign-compare -Wpointer-arith -Wnested-
        externs \ 
        -Wmissing-declarations -Wcast-align - 
D_GNU_SOURCE $(INCLUDE) \ 
        -DDDEBUG=1
TARGETS=tutlib.so
all: $(TARGETS)
tutlib.o: tutlib.c ../mapidflib.h ../mapi.h 
gcc $(CFLAGS) -c $<
tutlib.so: tutlib.o pktcounter.o 
gcc $(CFLAGS) -shared -o $@ $^ -lfl -lrt -L. -L $(LIB_DIR) 
pktcounter.o: pktcounter.c 
gcc $(CFLAGS) -c $<
clean:
    @/bin/rm -f *.o *.so *~ $(TARGETS)
```

The new MAPI function and library can now be compiled by running `make`. The last step to be able to use the new function is to make sure that the `tutlib` library is loaded when MAPId is started. To do this modify `mapi.conf` so that the `libpath` entry includes the `tutlib` directory and the `libs` entry includes the `tutlib` library. It should look something like this:

```bash
libpath=.:ipfixlib:tutlib 
libs=mapidstdflib.so:ipfixlib.so:tutlib.so
```

A.4.2 Function with internal data: GAP

This is a slightly more advanced function that uses internal data to store information between each call to the process interface. The function calculates the gap or the time that has elapsed between to consecutive packets in a flow.

scampi@ist-scampi.org 111 January 12, 2005
A.4.2.1 Implementation

Go to the tutlib directory created in the previous section. Copy the function template:

```c
cp ../funct_template.c gap.c
```

Interfaces that are not needed should be removed. Since this function uses internal data, the init and cleanup interfaces are needed in addition to the process interface. The string `funct_name` should be replaced with `gap` and the mapidflib_function_def structure needs modification so that it looks like:

```c
static mapidflib_function_def_t finfo=
  "",  //libname
  "GAP", //name
  "Returns the gap between to consecutive packets\n
  Return value: unsigned long long", //descr
  "", //argdescr
  MAPI_DEVICE_ALL, //devoid
  MAPIRES_SHM, //Method for returning results
  sizeof(unsigned long long), //shm size
  0, //modifies_pkts
  NULL, //instance
  gap_init, // gap_init,
  gap_process,
  NULL, //get_result,
  NULL, //change_args
  NULL, //reset
  gap_cleanup, //cleanup
  NULL, //client_init
  NULL, //client_read_result
  NULL //client_cleanup
};
```

We are now ready to implement the code for the three interfaces this MAPI function needs. What this function do is to store the time stamp of a packet and when the next packet is captured it calculates the time difference between the new and old timestamp. This means that it needs to store the timestamp of a packet as internal data so that it can be accessed again when the next packet is processed.

Timestamps are unsigned long long values so the init interface should simply allocate memory to store such a value and let the internal data pointer point to this memory. The function should also initialize the unsigned long long value to 0. The code for this is then:

```c
instance->internal_data=malloc(sizeof(unsigned long long));
*((unsigned long*)instance->internal_data)=0;
```

The process interface stores the difference between the timestamp of the current packet and the previous one and stores it in the result data:
unsigned long long *gap,*old;
old=instance->internal_data;
gap=instance->result.data;
if(*old!=0)
  *gap=pkt_head->ts-*old;
  *old=pkt_head->ts;

When the flow closes, the internal data has to be freed. This is done by the cleanup interface:

  free(instance->internal_data);

The entire source code for this function would then be:

A.4.2.2 Function library

The function library is recreated using the createlib.pl script as in the previous section:

  ../createlib.pl tutlib tutlib.c

A.4.2.3 Makefile

The Makefile created in the previous section must be modified so that it compiles the new function:

  INCLUDE=-I. -I..
  CFLAGS=-g -Wall -Wsign-compare -Wpointer-arith -Wnested-externs \
    -Wmissing-declarations -Wcast-align -D_GNU_SOURCE $(INCLUDE)
  TARGETS=tutlib.so
  all: $(TARGETS)
    tutlib.o: tutlib.c ../mapidflib.h ../mapi.h
      gcc $(CFLAGS) -c $<
    tutlib.so: tutlib.o pktcounter.o gap.o
      gcc $(CFLAGS) -shared -o $@ $^ -lfl -lrt -L. -L $(LIB_DIR)
      cp $@ ..
    pktcounter.o: pktcounter.c
      gcc $(CFLAGS) -c $<
    gap.o: gap.c
      gcc $(CFLAGS) -c $<
    clean:
      @/bin/rm -f *.o *.so "$@"

A.4.3 Function with multiple process interfaces: PKTINFO

We are now going to implement a function that returns some information about a packet. To keep things simple we will limit the information to either packet length or timestamp. Results from this function will typical be used by other functions for further processing.
APPENDIX A. MAPI FUNCTIONS TUTORIAL

A.4.3.1 Implementation

As in the previous examples, start by copying the function template and set the function name to pktinfo:

\[
\text{cp ../funct_template.c pktinfo.c}
\]

This function will need an argument to decide which information that should be returned. This means that we need to specify the type of argument in the mapidflib_structure. We will also need to implement the instance interface to do a syntax check of the argument. The argument type will be an integer so the argdescr should be set to “i”.

Since the information returned by the function is the same for the entire lifetime of the function it is a waste of clock cycles to do a check for every packet to decide which type of information should be returned. Instead we can implement two version of the process interface, one that returns packet length and one that returns the timestamp. We can decide in the instance interface which process interface to use.

The mapidflib_structure should then look like this:

\[
\text{static mapidflib_structure pktinfo=}
\]

To make things more user friendly we will first create an enum that can be used for specifying the type of information that should be returned by the function. This is placed in the file pktinfo.h:

\[
\text{#ifndef pktinfo_h}
\]

\[
\text{#define pktinfo_h}
\]

\[
\text{enum pktinfo \{ PKT_TS,PKT_SIZE \};}
\]

\[
\text{#endif}
\]

scampi@ist-scampi.org

January 12, 2005
The code in the instance interface should then verify that the value of the argument is either \texttt{PKT\_TS} or \texttt{PKT\_SIZE}. If the value is not one of these, an error message should be returned. Based on the argument the instance interface should then modify the function instance copy of the mapi\_lib\_function\_def structure so that the process pointer points to one of the two processing interfaces that are implemented.

To implement the instance interface we should start by defining two variables, one for storing the type of information that should be returned and one pointer for pointing to the arguments:

\begin{verbatim}
int type;
mapiFunctArg* fargs;
\end{verbatim}

To parse function arguments, the getarg functions found in mapiipc.h can be used. To get hold of the integer that specifies the information type we use:

\begin{verbatim}
fargs=instance->args;
type = getargint(&fargs);
\end{verbatim}

We can now check the value of \texttt{type} and set the pointer to the correct process interface accordingly:

\begin{verbatim}
if(type==PKT\_SIZE)
    instance->def->process=pktinfo\_process\_size;
else if(type==PKT\_TS)
    instance->def->process=pktinfo\_process\_ts;
else
    return MFUNCT\_INVALID\_ARGUMENT\_2;
\end{verbatim}

The last step is to implement the two processing interfaces, \texttt{pktinfo\_process\_size} and \texttt{pktinfo\_process\_ts}. The code for the interface returning packet size will be:

\begin{verbatim}
(*\(\text{unsigned long long}\)*instance->result.data)=
(\text{unsigned long long})pkt\_head->wlen;
return 1;
\end{verbatim}

For returning the packet timestamp we get:

\begin{verbatim}
(*\(\text{unsigned long long}\)*instance->result.data)=
(\text{unsigned long long})pkt\_head->ts;
return 1;
\end{verbatim}

The entire code for the pktinfo function will then be:

\section*{A.4.3.2 Function library and Makefile}

The function library and Makefile must be updated in the same way as explained in the previous section.
A.4.4 Function that reads results from other functions: STAT

The statistical function reads unsigned long long values from other functions and calculates the total number of values read, the sum, square of sum, maximum and minimum values. It is a good example of how one function can read results from another function.

A.4.4.1 Implementation

Copy the function template from the mapi main directory:

```bash
cp ../funct_template.c stat.c
```

Replace the text “func_name” with the text “stat”. Delete interfaces that are not needed. This function needs arguments and uses internal data to store information so it need to implement the instance, init, process and cleanup interfaces.

The function will take two arguments, one integer that specifies the function ID of the function that the results are read from and one function that specifies if the result from the first packet should be skipped. The last parameter is needed if we are reading results from a function like GAP. This function will always return a 0 when the very first packet is processed and if we did not skip this first result, the minimum value would always become 0 in the STAT function.

This function will count the number of values read from another function, the sum, square of sum, maximum and minimum value. All this information will be stored in a structure placed in a stats.h file:

```c
#ifndef STATS_H
#define STATS_H 1
typedef struct stats {
    unsigned long long count; //Number of elements
    long double sum; //Sum
    long double sum2; //Sum of square
    double max; //Maximum value
    double min; //Minimum value
} stats;
#endif
```

It is also this structure that is returned to an application that calls mapi_read_result.

The function will then need an instance interface for verifying the arguments, an init interface to initialize internal resources that stores the statistics, a processing interface to do the processing of packets and a reset interface for resetting the results. The mapidflib_function_def interface should then be:

```c
static mapidflib_function_def finfo=
    "", //libname
    "STATS", //name
    "Returns statistical information about unsigned long long values from other functions",
```

scampi@ist-scampi.org  116 January 12, 2005
In the instance interface we must check that the first argument points to an already existing function applied to the flow and we should check that the second parameter is either a 0 or a 1. To get the values of the two arguments that is passed to the function getargint and getargchar is used. To check the first parameter we can simply look in the function list flist to see if the function exists. The code would then be:

```c
static int stats_instance(mapidflib_function_instance_t *instance,
                          flist_t *flist,
                          mapidflib_flow_mod_t *flow_mod)
{
    int fid;
    mapiFunctArg* fargs;
    int skip;

    fargs=instance->args;
    fid = getargint(&fargs);
    skip=getargchar(&fargs);

    if(flist->get(flist,fid)==NULL)
        return MFUNCT_INVALID_ARGUMENT_1;
    if(skip>1 || skip<0)
        return MFUNCT_INVALID_ARGUMENT_2;

    return 0;
}
```

In the init interface we have to store a reference to the function which results are read from as well as the value for the skip argument so that these can be used when processing packets. To do this we define a structure that the internal data pointer can point to:

```c
typedef struct stats_inst {
```
The implementation of the init interface would then look like this:

```c
static int stats_init(mapidflib_function_instance_t *instance,
                      flist_t *flist)
{
    int fid;
    stats_instance_t *i;
    mapiFunctArg* fargs=instance->args;
    i=instance->internal_data=malloc(sizeof(stats_instance_t));
    fid=getargint(&fargs);
    i->skip=getargchar(&fargs);
    if((i->res_instance=flist_get(flist,fid))==NULL)
    {
        free(i);
        return MFUNCT_INVALID_ARGUMENT;
    }
    return 0;
}
```

The code for processing packets is relatively straightforward. The results are stored in `result.data` and information about which function to read values from is located in `internal_data`. The only problem is how to read the results from another function. The method used depends on how the other function returns results. If the function uses shared memory, the result should be read directly from memory. If however, the function implements its own read result interface, this interface must be used. To ease the implementation a function in `fhelp.[ch]` can be used:

```c
fhlp_get_res(i->res_instance)
```

This function will always return a pointer to a `mapidflib_result_t` structure containing the result. The code for the processing interface will then be:

```c
unsigned long long *res;
stats_instance_t *i=instance->internal_data;
stats_t *s=instance->result.data;
if(i->skip==1)
    i->skip=0; //Skip first packet
else {
    //Get result from other function
    res=((mapidflib_result_t*)fhlp_get_res(i->res_instance))->data;

    //Calculate statistics
    if(s->max<*res)
        s->max=*res;
    if(s->min>*res || s->min==0)
```

scampi@ist-scampi.org 118 January 12, 2005
The reset interface simply has to set the various variables in the result structure to 0:

```c
stats_t *stats;
stats=instance->result.data;
stats->count=0;
stats->sum=0;
stats->sum2=0;
stats->max=0;
stats->min=0;
```

The last interface to implement is cleanup which releases the internal data:

```c
free(instance->internal-data);
```

The entire code for the stats function then becomes:

```c
    s->min=*res;
    s->count++;
    s->sum+=*res;
    s->sum2+=((long double)*res*(long double)*res);
}
return 1;
```

### A.4.5 Function with function specific get_result: TOBUFFER

The last function we will look at is the TOBUFFER function which store packets in a circular buffer so that clients can read them using `mapi_get_next_pkt`. The circular buffer uses shared memory, but since two way communication using semaphores are needed to control reading and writing to this buffer, the TOBUFFER function must implement its own `get_result` interface on the client side.

The description here is not a full description of how to implement the TOBUFFER function, but focus on explaining the challenges of using a function specific method for reading results.

#### A.4.5.1 Implementation

In the TOBUFFER function we use shared memory to transfer the packets to the client application. One problem is that we want to be able to store the same number of packets on different types of hardware, which means that the size of this buffer will depend on the maximum packet capture length of the adapter that the function is running on. Because of this it is not possible to put a fixed number in the `mapidflib_def` structure to define the size of shared memory that the function needs.

What we can do instead is to implement the instance function interface and modify the `shm_size` entry inside this interface based on the value of `hwinf->cap_length`.
APPENDIX A. MAPI FUNCTIONS TUTORIAL

In addition to the buffer we also need to share a structure that can be used for accessing the circular buffer in a controlled manner:

```c
typedef struct to_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
    int cap_length; // Maximum size of a captured packet
    unsigned bufsize; // Size of buffer
    fhlp_sem_t sem; // Struct containing semaphore info
    char *buf; // Pointer to buffer
} to_buffer_t;
```

The code for deciding the amount of shared memory needed will then be:

```c
instance->def->shm_size=sizeof(to_buffer_t) +
    NUM_PKTS*(sizeof(struct mapid_pkthdr)+
    instance->hwinfo->cap_length);
```

Since there are no arguments to this function, there is no other work to be done in the instance interface.

In the init interface we now need to initialize the circular buffer and create semaphores for coordinating the access to the buffer. The `to_buffer_t` structure is placed in the beginning of the shared memory and the circular buffer occupies the rest. To create a semaphore a function from `fhelp.c` can be used. The code for the init interface will then be:

```c
to_buffer_t *mbuf;
int ret;
mbuf=instance->result.data;
mbuf->buf=(char*)instance->result.data+sizeof(to_buffer_t);
// Adding semaphore
if((ret=fhlp_create_semaphore(&mbuf->sem,2))!=0)
{
    DEBUG_CMD(printf("Error initializing semaphore: %d\n",ret));
    return ret;
}
mbuf->read_ptr=0;
mbuf->write_ptr=0;
mbuf->bufsize=NUM_PKTS*(sizeof(struct mapid_pkthdr)+
    instance->hwinfo->cap_length);
mbuf->cap_length=instance->hwinfo->cap_length;
return 0;
```

The processing interface is not described here, but to get hold of the `to_buffer_t` structure it simply uses the results.data pointer.

The cleanup interface needs to release the semaphore created in init. This can also be done by a function in `fhelp.c`:
To be able to return packets to the user, the TOBUFFER function also has to implement a client\textunderscore get\textunderscore result interface. It is important to realize that while the code for this interface is in the same source code, it runs on the client side and not inside MAPlID as the rest of the interfaces.

Since TOBUFFER uses shared memory the instance\textunderscore result\textunderscore data argument to the client\textunderscore read\textunderscore result interface will automatically point to the shared memoy. This means that instance\textunderscore result\textunderscore data points to the same to\textunderscore buffer\textunderscore t structure as seen by the processing interface and that the circular buffer is located right after this structure:

```c
    to\textunderscore buffer\textunderscore t *tb=instance->result\textunderscore data;
    char *buf=(char*)instance->result\textunderscore data+sizeof(to\textunderscore buffer\textunderscore t);
```

The entire source code for this function can be found in the file stdlib/tobuffer\textunderscore all\textunderscore c
Bibliography


[pap] Performance application programming interface (papi).


