Design of an Application Programming Interface for IP Network Monitoring*

M. Polychronakis, E. P. Markatos
Institute of Computer Science
Foundation for Research and Technology - Hellas
P.O.Box 1385 Heraklio, GR-711-10 GREECE
{mikepo, markatos}@ics.forth.gr

Arne Øslebø
UNINETT S.A.
N-7465 Trondheim, Norway
Arne.Oslebo@uninett.no

K. G. Anagnostakis
CIS Department, University of Pennsylvania
Philadelphia, PA, USA
anagnost@dsl.cis.upenn.edu

Abstract

We propose a novel general-purpose network traffic Monitoring Application Programming Interface (MAPI) for network monitoring applications. Our work builds on a generalized network flow model that we argue is flexible enough to capture emerging application needs, and expressive enough to allow the system to exploit specialized monitoring hardware, where available. We describe an implementation of MAPI using the DAG 4.2 Gigabit Ethernet monitoring card and a commodity Gigabit Ethernet adapter, we present a set of experiments measuring overheads, and we demonstrate potential applications. Our experimental results suggest that MAPI has more expressive power than competing approaches, while at the same time is able to achieve significant performance improvements.

Keywords

network monitoring, accounting, network flow, MAPI

1. Introduction

Effective network traffic monitoring is becoming increasingly vital for network management as well as for supporting a growing number of automated control mechanisms needed to make the IP-based Internet more robust, efficient, and secure.

The need for effective network traffic monitoring, along with increasing link speeds, has exposed limitations in existing network monitoring architectures that are deeply rooted in the basic abstractions used. The most widely used abstraction for network traffic monitoring has been that of flow-level traffic summaries, first demonstrated in software prototypes such as NeTraMet[4] and later incorporated as standard functionality in routers (c.f., Cisco's NetFlow[5]). This approach has been reasonably successful in supporting monitoring functions ranging from accounting to some rather simple forms of denial of service attack detection [21]. However, the information contained in flow-level summaries is usually not detailed enough for emerging monitoring functions. For instance, determining per-application network usage is not possible for some of the major new applications

*This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union. M. Polychronakis and E. P. Markatos are also with the University of Crete. The work of K. G. Anagnostakis was done while at ICS-FORTH.

that dynamically allocate ports, such as peer-to-peer file sharing, multimedia streaming, and conferencing applications. Additionally, traditional flow-level traffic summaries are usually not adequate for security monitoring as provided by intrusion detection systems. These security applications usually need much more information than provided by flow-level traffic summaries. For example, in order to detect and contain computer viruses and worms at times of emergency, intrusion detection systems need to be able to inspect and process network packet payloads, which are not available in flow-level traffic summaries.

In absence of any better abstraction, many network operators resort to full packet capture[8] or case-specific solutions usually supported by specialized hardware[7]. Such approaches have high hardware cost, significant processing needs and produce vast amounts of data, complicating the task of data analysis.

We argue that such limitations (i.e. too little information provided by flow-level traffic summaries vs. too much data provided by full packet capture) demonstrate the need for a portable general-purpose environment for running network monitoring applications on a variety of hardware platforms. If properly designed, such an environment could provide applications with just the right amount of information they need: neither more (such as the full packet capture approaches do), nor less (such as the flow-based statistics approaches do). In this paper we present an *expressive programming interface*, which enables users to clearly communicate their monitoring needs to the underlying traffic monitoring platform. The Monitoring Application Programming Interface (MAPI) builds on a generalized flow abstraction that allows users to tailor measurements to their own needs. The main novelty of MAPI is that it elevates flows to first class status, allowing programmers to perform a set of standard operations on flows similar to other system abstractions such as files and sockets. Where necessary and feasible, MAPI also allows the user to trigger custom processing routines not only on summarized data but also on the packets themselves, similar to programmable monitoring systems [1, 13]. The expressiveness of MAPI enables the underlying monitoring system to make informed decisions in choosing the most efficient implementation, while providing a coherent interface on top of different lower-level elements, including intelligent switches, high-performance network processors, and special-purpose network interface cards.

This paper presents the design of MAPI and evaluates an implementation on top of two network traffic monitoring systems: an off-the-shelf Gigabit Ethernet network interface, and a more sophisticated traffic capture card. Section 2 presents the design of MAPI, while Section 3 presents our implementation. In Section 4 we evaluate the performance of MAPI in comparison with existing approaches. Section 5 places our contribution in the context of related efforts, and Section 6 summarizes and concludes the paper.

2. The Design of MAPI: a network traffic monitoring API

The goal of an application programming interface is to provide a suitable abstraction which is both simple enough for programmers to use, and powerful enough for expressing complex application specifications. A good API should also relieve the programmer from the complexities of the underlying hardware while making sure that hardware features can be properly exploited.

MAPI builds on a simple and powerful abstraction, the *network flow*, but in a flexible and generalized way. In MAPI, a network flow is generally defined as a sequence of packets that satisfy a given set of conditions. These conditions can be arbitrary, ranging from simple header-based filters, to sophisticated protocol analysis and content inspection functions. For example, a very simple flow can be specified to include *all* packets, or all packets directed to a particular web server. A more complex flow may be composed of all TCP packets between a pair of subnets that contain the string "User-agent: Mozilla/5.0".

Our approach to network flows is therefore fundamentally different from existing models that constrain the definition of a flow to the set of packets with the same source and destination IP address and port numbers within a given time-window. In contrast with existing models, MAPI gives the "network flow" a *first-class status*: flows are named entities that can be manipulated in similar ways to other programming abstractions such as sockets, pipes, and files. In particular, users may create or destroy (close) flows, read, sample or count packets of a flow, apply functions to flows, and retrieve other traffic statistics from a flow.

By using first-class flows, users can express a wide variety of new monitoring operations. For example, MAPI flows allow users to develop simple intrusion detection schemes that require content inspection [2]. In contrast, traditional approaches to traffic/network flows, such as NetFlow, IPFIX, and related systems and proposals do not have the means of providing the advanced functions required for this task.

In the remainder of this Section we present an overview of the main operations provided by MAPI. A complete specification of MAPI and a more detailed description of each function is provided in [20].

2.1 Creating and Terminating Network Flows

Central to the operation of the MAPI is the action of creating a network flow:

`fd = mapi_create_flow(char *dev, cond *c, mode m)`

This call creates a network flow, and returns a flow descriptor `fd` that points to it. This network flow consists of all network packets which go through network device `dev` and which satisfy condition `c`. For example `mapi_create_flow("/dev/dag0", "dst port 80", RAW)` creates a network flow of all packets seen by the network interface `/dev/dag0` destined to port 80. The third argument enables the monitoring system to perform some pre-processing on the stream of packets. For example, when `m` is set to `COOKED`, then the individual packets are pre-processed according to their protocol (i.e. TCP or UDP) and concatenated into a data stream. This preprocessing will re-assemble fragmented packets, discard retransmitted packets, re-order out-of-order packets, etc.

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

`fd = mapi_close_flow(flow_desc fd)`

After closing a flow, all the structures that have been allocated for the flow are released.

2.2 Reading packets from a flow

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

`packet * mapi_get_next_packet(fd)`

which reads the next packet that belongs to flow `fd`. If the user does not want to read one packet at-a-time and possibly block, (s)he may register a callback function that will be called when a packet to the specific flow is available:

`mapi_loop(flow_desc fd, int cnt, mapi_handler callback)`

The above call makes sure that the handler `callback` will be invoked for each of the next `cnt` packets that will arrive in the flow `fd`. *

* Although the `mapi_loop` call is inspired from the `pcap_loop` call of the `libpcap` library [14], in contrary to `pcap_loop`, `mapi_loop` is non-blocking.

2.3 Applying functions to Network Flows

Besides neatly arranging packets, network flows allow users to treat packets that belong to different flows in different ways. For example, a user may be interested in *logging* all packets of a flow (e.g. to record an intrusion attempt), or in just *counting* the packets and their lengths (e.g. to count the bandwidth usage of an application), or in *sampling* the packets (e.g. to find the IP addresses that generate most of the traffic). The abstraction of the network flow allows the user to clearly communicate to the underlying monitoring system these different monitoring needs. To enable users to communicate these different requirements, MAPI enables users to associate functions with flows:

```
mapi_apply_function(flow_desc fd, function f, ...)
```

The above association applies function `f` to every packet of flow `fd`. Based on the header and payload of the packet, the function will perform some computation, and may optionally discard the packet.

MAPI provides several *predefined* functions that cover some standard monitoring needs. For example, function `PACKET_COUNT` counts all packets in a flow, function `SAMPLE_PACKETS` can be used to sample packets, etc. There also exist functions that count various traffic metrics, like bandwidth or fragmented packets. MAPI also provides parameterized hashing functions that will take user defined arguments. Based on the value of the hashing function, the packet may be kept or discarded. Although these functions will enable users to process packets, and compute the network traffic metrics they are interested in without receiving the packets in their own address space, they must somehow communicate their results to the interested users. For example, a user that will define that the function `packet_count` will be applied to a flow, will be interested in reading what is the number of packets that have been counted so far. This can be achieved by allocating a small amount of memory or a data structure to each network flow. The functions that will be applied to the packets of the flow will write their results into this data structure. The user who is interested in reading the results will read the data structure using:

```
mapi_read_results(flow_desc fd, function f, void * result)
```

2.4 Dynamically Generated Flows

Although network flows provide users with a powerful abstraction to group related packets together, there are cases where users cannot provide an exact specification of the flows they are interested in. In addition, users may be interested in viewing traffic data from several different points of view. For example, they may want to know the distribution of traffic among applications, the distribution of traffic among source or destination IP addresses, etc. Overall, users may be generally interested in gathering traffic statistics, without being particularly interested in observing packet payloads - they just need header information. To cater to these user needs, MAPI defines the `hierarchical` network flows:

```
fd = mapi_create_flow("dag0", "port 80" , HIERARCHICAL)
```

The above call creates the `hierarchical` flow `fd` composed of all packets destined to port 80. In contrast to `raw` and `cooked` flows, `hierarchical` flows do not deliver packets to end users. Instead they are composed of several `sub-flows`. A sub-flow is defined to contain all packets (within the parent flow) that have a common 5-tuple of source and destination IP address, source and destination port, and protocol number. When a new packet of a flow arrives, it is sent to its appropriate sub-flow. If no such flow exists, a new flow is created. Each `hierarchical` flow has a predefined limit of sub-flows and when a sub-flow is idle for more than a predefined interval, it is considered expired. Users may also define an upper limit for the duration of a flow. If a flow continues to receive packets for an interval longer than that, it is considered expired. When a new flow is created, but the number of active sub-flows has reached the limit, one of them expires.

2.5 MAPI example: monitoring FTP traffic

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

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

```
    packet *p;
    flow_descriptor fd, xfers[1024];
    struct byte_count_results br;
    int src_port, dst_port, count, total_ftp_traffic=0;
    char new_flow[64];

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

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

    /* Track the next 100 transfers */
 3: for(count=0; count<100; count++){
 4:   p = mapi_get_next_packet(fd);
      /* extract_ports gets a packet which indicates the beginning */
      /* of a new transfer and extracts the dynamic data port      */
 5:   extract_ports(p, &src_port, &dst_port);
      /* Create a flow to track the forthcoming transfer according */
      /* to the information contained in the control packet        */
 6:   sprintf(new_flow, "tcp src port %d and dest port %d", port[0], port[1]);
      /* Create a new flow for this data transfer */
 7:   xfers[count] = mapi_create_flow(/dev/scampi, new_flow, RAW);
      /* Count the bytes transfered in this flow */
 8:   mapi_apply_function(xfers[count], BYTE_COUNT);
    }
    /* summary */
 9: for(count=0; count<100; count++){
10:   mapi_read_results(xfers[count],BYTE_COUNT,&br);
11:   total_ftp_traffic += br.bytes;
    }
```

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

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

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

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

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

3. Implementation

We have implemented a subset of MAPI on top of two different network monitoring platforms: (i) an Intel Pro 1000 MT (Gigabit Ethernet) desktop adapter and (ii) a DAG 4.2 GE monitoring card for Gigabit Ethernet. The Intel Pro 1000 MT is a commodity adapter, which, if put in promiscuous mode can capture all packets that go through it and can forward them to the kernel of the host computer. The DAG 4.2 GE monitoring card is capable of passive full packet capture at the speed of one Gigabit per second. Contrary to the Intel Pro 1000 MT commodity adapter, the DAG card, is capable of retrieving and mapping to user space network packets through a zero-copy interface, which avoids costly interrupt processing. It can also stamp each packet with a high precision time stamp. A large static circular buffer memory-mapped to user-space is used to hold arriving packets, avoiding wasted time for costly packet copies. User applications can directly access this buffer without the invocation of the operating system kernel.

Figure 1 shows the main modules of the MAPI system. On the top of the Figure we see a set of monitoring applications that, via a MAPI stub, communicate with the MAPI daemon. The daemon consists of two threads, one thread for packet processing and one for the communication with the monitoring applications. All active applications and their defined flows are internally stored in the daemon in a two-dimensional list. List nodes contain all necessary data structures for application or flow definition and accounting. Each captured packet is checked by the main processing thread against the defined flow filters. Then, for every flow it belongs to, the appropriate actions are made: counters are incremented, sampling or substring search functions are applied and the packet might be sent to the application or dumped to disk by the daemon. In the current implementation, filtering is accomplished using the `bpf_filter()` function of the `libpcap` library which applies a compiled BPF filter to a packet in user level. Each compiled filter is stored into the corresponding flow structure.

All communication of the daemon with the monitoring applications is handled by the

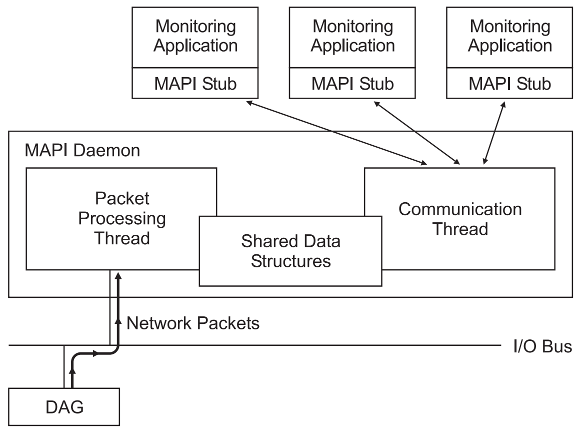

Figure 1: Mapi Daemon Architecture.

	NIC		DAG	
	`libpcap`	MAPI	`libpcap`	MAPI
Cycles per Packet	11897	13082	453	235

Table 1 Packet fetching performance for MAPI and `pcap`.

"communication thread". This thread constantly listens for requests made by the monitoring application through MAPI functions and sets up the appropriate shared data structures. When monitoring applications need to read data, the communication thread reads these data from the shared data structures. All communication between monitoring applications and the MAPI daemon is through Unix domain datagram sockets.

4. Experimental Evaluation

In this Section we experimentally evaluate the performance of our MAPI implementation and compare it to alternative approaches such as the commonly-used `libpcap` library. Our experimental environment consists of three PCs connected to a Gigabit Ethernet switch (an SMC 8506T). The first PC (equipped with a 1.1 GHz Pentium III processor) generates and transmits traffic to the second PC (equipped with a 2.5 GHz Pentium IV). Traffic consisting of 1460-byte UDP packets is generated at constant rate using `iperf`[22]. The traffic is mirrored by the switch and sent to the third PC, which performs the network monitoring. The monitor PC is a dual 1.8 GHz AMD Athlon MP 2200+, with 512 MB of main memory, a DAG 4.2 network traffic monitoring card and a Gigabit Ethernet MT Intel Pro network interface. The host operating system is Debian Linux 3.0, kernel version 2.4.20.

4.1 Basic Packet Processing Cost

In our first experiment we set out to find the basic operating cost per received packet. This is the cost to receive the packet to the place where it will be further processed. Such processing may probably include sampling, hashing, update of counters, application of functions, etc. MAPI performs all these functions within the MAPI daemon, while

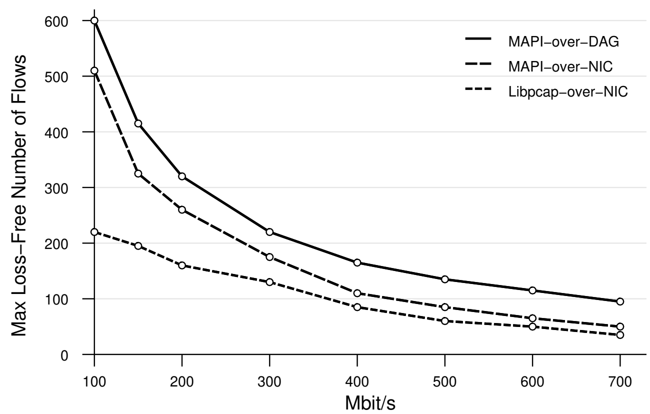

Figure 2: Maximum number of flows when gathering packet and byte statistics for each flow.

`libpcap` delegates these functions to user applications. Thus, the basic operating cost per received packet for MAPI is the cost to receive each packet in the MAPI daemon's address space, while for `libpcap` it is the cost to receive the packet to the user application's address space.

We generated a 10 Mbit/s traffic stream consisting of 1460-bytes long packets and measured the number of cycles the processor spends in order to receive each packet. The number of cycles was measured using the PAPI Performance Application Programming Interface [12]. Table 1 presents our results for MAPI and `libpcap` on top of the commodity network interface (NIC) and on top of the DAG card. We see that `libpcap` consumes 11897 cycles per packet, while MAPI consumes slightly more at 13082 cycles per packet. This is as expected, since the implementation of MAPI on top of the NIC is built on top of `libpcap`. The two left columns of table 1 reflect the cost of `libpcap` and MAPI on top of the DAG card. We see that the DAG card allows for a more efficient implementation than the NIC, avoiding packet copying between kernel and user space, as well as operating system calls, because the DAG card delivers packets directly in user space via a memory mapped buffer. Therefore, `libpcap` spends 453 cycles per packet, while MAPI spends 235 cycles per packet, more than an order of magnitude improvement compared to the implementations on top of the NIC. Overall, we see that MAPI is 10% more expensive than `libpcap` on top of the commodity interface, while it is 1.9 times faster than `libpcap` on top of the DAG card.

4.2 Performance vs. number of flows

In our next experiment we set out to explore what is the performance of MAPI as a function of the number of active network flows. To do so, we wrote a simple monitoring benchmark that creates a varying number of flows. Flow `i` consists of all packets destined to port `i`. We generated traffic that was not destined to any of these ports and measured the maximum number of flows that the monitoring application may sustain before it saturates the processor and starts dropping packets. Figure 2 shows the maximum number of flows that can be sustained for a given input traffic. The solid line shows the performance of MAPI on top of the DAG card. We see that when the network traffic is low, MAPI can

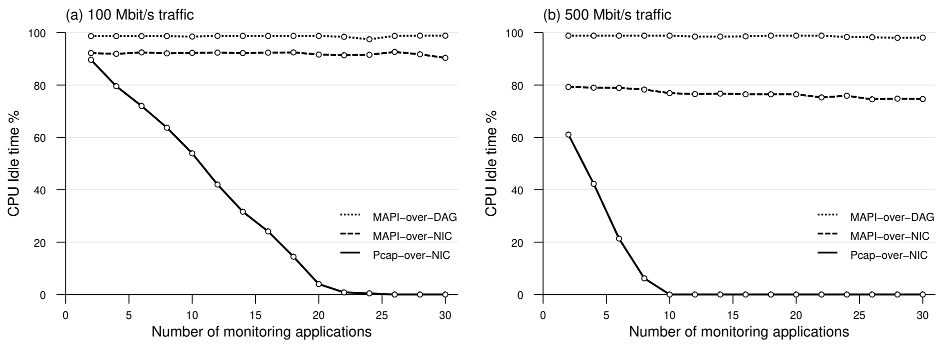

Figure 3: Performance of multiple packet sampling applications.

sustain up to 600 different network flows. As expected the maximum number of loss-free flows decreases with the amount of monitored traffic. Thus, when the network traffic reaches 700 Mbit/s, the maximum number of loss-free flows is close to 100. Figure 2 also plots the performance of MAPI on top of the commodity Ethernet adaptor (MAPI-over-NIC), which is somewhat lower than the performance of MAPI-over-DAG. This is because the MAPI-over-NIC induces higher processor overhead as compared to MAPI-over-DAG. Indeed, for each network packet received, MAPI-over-NIC needs to suffer the overhead of at least on interrupt, while MAPI-over-DAG is able to deliver network packets in user space without any processor intervention.

Figure 2 also compares MAPI with a `libpcap`-based implementation of the same monitoring application. That is, we re-wrote the same application using only calls of the `libpcap` library and measured its performance. We see that the performance of MAPI is better than that of `libpcap`, especially for low traffic when we create a large number of flows. This performance improvement of MAPI compared to `libpcap` is due to the different ways MAPI and `libpcap` handle asynchrony. In MAPI, each network flow registers its interest in network packets and blocks waiting for suitable packets to arrive. Thus, the MAPI-based application will block waiting for packets that match a network flow to arrive. Since no packets will match any of the flows, the application will remain blocked without wasting any processor cycles. On the contrary, the relevant calls of `libpcap` for asynchronously receiving of packets (i.e. `pcap_open_live` and `pcap_dispatch`) are based on polling which introduces an additional source of overhead. If for example, we want to create 500 network flows in `libpcap`, we will need to create 500 different `pcap_open_live` entities which will periodically poll for packets, effectively wasting the processor's cycles.

4.3 Supporting several Sampling Monitoring Applications

In this experiment we set out to explore the performance of MAPI implementation when required to support several different monitoring applications. We compare the performance of MAPI over the DAG card (MAPI-over-DAG), with MAPI over the commodity 1 Gbit/s interface (MAPI-over-NIC), and `libpcap` over the same interface (PCAP-over-NIC). *

*Note that we do not present performance results for the performance of `libpcap` over the DAG card, since the current implementation of `libpcap` over the DAG card does not support more than one monitoring application.

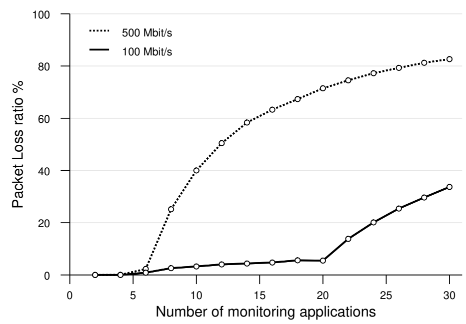

Figure 4: Packet Loss ratio of `libpcap` implementation. Note that the packet loss ratio for both MAPI-over-DAG and MAPI-over-NIC was under 1%.

In this experiment we created a number of monitoring applications, where each application was sampling one out of every 20,000 packets of the monitored traffic. We generated traffic at constant rate, at 100 Mbit/s and at 500 Mbit/s and plotted the results in Figure 3. The metric of interest here is CPU idle time as a function of the number of monitoring applications. Figure 3 shows that as the number of monitoring applications increases, the performance of `libpcap` deteriorates rapidly. When the monitored traffic is at 100 Mbit/s, `libpcap` saturates the processor at about 25 applications, while when the monitored traffic is at 500 Mbit/s, `libpcap` saturates the processor at around 10 applications. Indeed, Figure 4 shows the percentage of packets that were lost by `libpcap` as a function of the number of monitoring applications. We see that the `libpcap`-based monitoring system starts dropping packets for as low as five monitoring applications. When the traffic is high 500 Mbit/s, the percentage of lost packets increases sharply at around 6 applications, reaching 50% in the area around 10-15 applications. When the traffic is low (i.e. 100 Mbit/s), this sharp increase is encountered when the number of applications exceeds 20-25.

In contrast, Figure 3 suggests that the performance of MAPI is practically independent of the number of applications. This is because MAPI requires fewer packet copy operations compared to `libpcap`. Applications written on top of `libpcap` can not instruct the system to perform sampling. Thus, all packets have to be copied to all applications' address spaces, only to be discarded (e.g., 19,999 out of 20,000 packets in the particular experiment). In contrast, applications written on top of MAPI are able to express that they are not interested in receiving most of the packets: they are only interested in receiving the sampled packets. Therefore, a MAPI-based monitoring system performs substantially better compared to a `libpcap`-based system.

It is also interesting to compare the performance of MAPI when running on top of DAG and when running on top of the commodity network interface (NIC). Figure 3 shows that the performance of MAPI applications on top of DAG (MAPI-over-DAG) is better than the performance of the same applications on top of the commodity NIC (MAPI-over-NIC). This is because the commodity NIC suffers at lease one interrupt for each incoming packet, while DAG cards write arriving packets in buffers mapped directly in user space.

4.4 Content matching

In our next experiment we set out to explore how the performance of MAPI changes with an increasing number of non-trivial monitoring applications. To do so, we created a

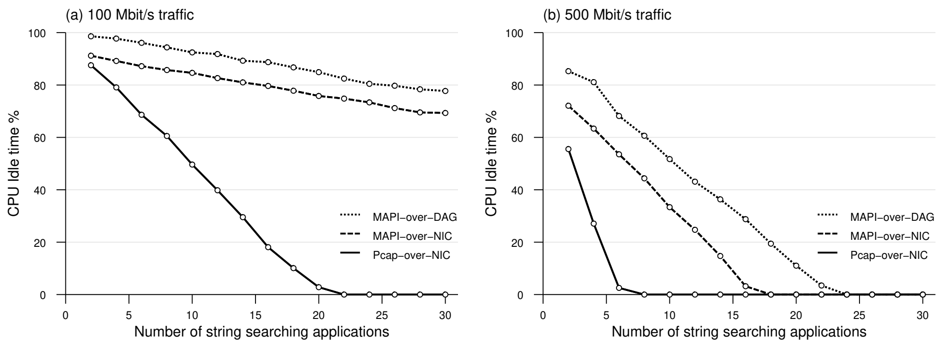

Figure 5: Performance of multiple string searching applications.

number of monitoring applications. Each application is interested in receiving all packets whose payload contains a given string. Each application searches for a different 8-characters long string, which is never found on any payload. Monitoring applications on top of MAPI register their monitoring needs by applying function `SUBSTRING_SEARCH` to all packets of a network flow. On the contrary, since the `libpcap` library does not provide such an ability, monitoring applications written on top of `libpcap` receive all packets in their address space and search for the substring using the Boyer-Moore algorithm [3].

Figure 5 shows the performance of different network monitoring systems for an input traffic of 100 Mbit/s and 500 Mbit/s respectively. We see that in all cases, performance decreases with the number of string-searching monitoring applications. However, the performance of `libpcap` decreases much more rapidly than the performance of MAPI. This is probably due to the fact that `libpcap` copies *all* network packets to the address spaces of all applications and *then* searches the packets to see if they contain the substring. On the contrary, MAPI first searches all substrings in all network packets, and then copies to the address spaces of the applications only the packets that contain the given substring, which in our example are none.

5. Related Work and Discussion

Although MAPI presents a novel approach to passive network monitoring, it shares some functionality with previously defined network monitoring systems. The Berkeley Packet Filter (BPF)[15] has been used extensively for network traffic capture and analysis. BPF provides the basic mechanism for capturing packets from a network interface and has been used in several systems including tracing tools such as `tcpdump`[10], and intrusion detection systems such as snort [19]. Linux Socket Filters[9], provide similar functionality to Berkeley Packet Filters, although they are implemented in the Linux operating system only. Berkeley Packet Filters and Linux Socket Filters are frequently used with the packet capture library, more well known as `libpcap` [14]. `libpcap` provides users with a flexible tool to filter and capture packets based on header fields. CoralReef provides set of tools and support functions for capturing and analyzing network traces [11]. `libcoral` is a central component of the CoralReef suite and provides an API for monitoring applications that is independent of the underlying monitoring hardware. Nprobe is a monitoring tool for network protocol analysis [16]. Although it is based on commodity hardware, it speeds up network monitoring tasks by using filters implemented in the firmware of a programmable network interface.

Our work is closely related to such tools. However, to the best of our knowledge MAPI appears to have significantly more expressive power than existing tools. For example, it can filter packets based on values in the packet payload. It can also apply arbitrary functions on packets, and keep statistics based on the results of these functions. The expressiveness of MAPI makes programming network monitoring applications easier and also increases efficiency, as demonstrated in Section 4. By being expressive, MAPI enables the underlying monitoring system to choose the most appropriate implementation that matches application needs.

Mmdump[23] is a specialized tool which extends `tcpdump` by parsing messages from RTSP, H.323 multimedia session control protocols to set up and tear down packet filters as needed to gather traces of multimedia sessions. It parses the control messages to extract the dynamically assigned port numbers. Then the packet filter is changed to capture the packets of the stream. Mmdump is a custom solution for tracking applications that use dynamic ports, while MAPI is a general-purpose traffic monitoring API which can be used for a much broader spectrum of applications (including those targeted by Mmdump).

Except for packet-capture oriented systems, there has been significant activity in the design of systems providing flow summary statistics [5]. For example, Cisco IOS NetFlow technology is an integral part of Cisco IOS software that collects and measures traffic data on a per-flow basis. In this context, a flow is usually defined by all packets that share a common protocol, source and destination IP address and port numbers. In contrast to capture systems based on `libpcap` and `libcoral`, NetFlow only extracts and maintains high-level statistics. NeTraMet, much like NetFlow, can collect traffic data on a per-flow basis [4], focusing only on flows that match a specific rule. FlowScan[17] analyzes and reports on NetFlow format data and produces graph images that provide a continuous, near real-time view of the network border traffic. MAPI can be used to buid a visualization application with the same capabilities but for a broader range of network characteristics.

There is also significant activity within the IETF for defining network traffic monitoring standards, such as RMON [24], PSAMP [6] and IPFIX [18].

Although MAPI shares some goals with the above flow-based monitoring systems, we believe that it has significantly more functionality. For example, by being able to examine packet payloads, MAPI is able to provide sophisticated traffic statistics. As Section 2.5 shows, MAPI is able to provide traffic statistics for applications that use dynamically allocated ports (such as FTP and peer-to-peer systems), while traditional monitoring approaches, such as NetFlow and NeTraMet, can not provide such traffic statistics.

6. Summary and concluding remarks

We have presented the design of MAPI, a flexible and expressive application programming interface for network traffic monitoring. The main goal of MAPI is to *provide a highly expressive interface for applications to specify their monitoring needs*. This is achieved by building an API around a generalized network flow abstraction that users can fine-tune for their particular application, and by providing an intuitive set of operations inspired by the UNIX socket-based network programming model. As a concrete example, we have discussed how MAPI can be used to properly account for applications that use dynamically-generated ports.

We have implemented MAPI on top of a commodity Gigabit Ethernet network interface (Intel Pro 1000 MT), as well as on top of a DAG4.2 special-purpose adaptor designed for high-speed packet capture. We have evaluated the implementation of MAPI and compared its performance with the `libpcap` library used for network monitoring in state-of-the-art systems. Our analysis suggests that *MAPI improves application performance* compared to `libpcap` as the number and complexity of applications sharing the monitoring infrastructure increases.

The impact of MAPI on efficiency and ease of programming is expected to increase as networks get faster, monitoring applications become more complex, and hardware-supported monitoring becomes more prevalent. The two hardware platforms used in this study are representative of low-end deployment scenarios and the benefits demonstrated are strictly on the conservative side. Although the specifics of implementing MAPI on top of more advanced hardware are subject for future work, the interaction between the expressiveness of MAPI and more sophisticated lower-level components (like Network Processors) is likely to improve performance further, considering that certain functions can be pushed to the hardware.

We expect that MAPI will provide an effective interface for applications to express their needs and allows the underlying monitoring system to optimize the implementation in the best possible way.

References

[1] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, Michael B. Greenwald, and J. M. Smith. Efficient packet monitoring for network management. In *Proceedings of the 8th IFIP/IEEE Network Operations and Management Symposium (NOMS)*, pages 423–436, April 2002.

[2] R. Bace and P. Mell. *Intrusion Detection Systems*. National Institute of Standards and Technology (NIST), Special Publication 800-31, 2001.

[3] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[4] Nevil Brownlee. Traffic Flow Measurement: Experiences with NeTraMet. RFC2123, http://www.rfc-editor.org/, March 1997.

[5] Cisco Systems. Cisco IOS Netflow, 2003. *http://www.cisco.com/warp/public/732/netflow/*.

[6] Nick Duffield. A framework for passive packet measurement, 2003. Internet Draft draft-ietf-psamp-framework-03.txt. *(work-in-progress)*.

[7] Endace measurement systems. *DAGH 4.2GE dual gigabit ethernet network interface card*, 2002. *http://www.endace.com/*.

[8] Chuck Fraleigh, Sue Moon, Christophe Diot, Bryan Lyles, and Fou ad Tobagi. Architecture of a Passive Monitoring System for IP Networks. Technical Report TR00-ATL-101801, Sprint, 2000.

[9] Gianluca Insolvibile. Kernel korner: The linux socket filter: Sniffing bytes over the network. *The Linux Journal*, 86, June 2001.

[10] V. Jacobson, C. Leres, and S. McCanne. TCPDUMP. *http://www.tcpdump.org/*.

[11] Ken Keys, David Moore, Ryan Koga, Edouard Lagache, Michael Tesch, and k claffy. The architecture of CoralReef: an Internet traffic monitoring software suite. In *PAM2001 — A workshop on Passive and Active Measurements*. RIPE NCC, April 2001.

[12] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis, using hardware counters. In *International Conference on Parallel and Distributed Computing Systems*, August 2001.

[13] G. Robert Malan and Farnam Jahanian. An extensible probe architecture for network protocol

performance measur ement. In *Proceedings of ACM SIGCOMM*, pages 215–227, August 1998.

[14] S. McCanne, C. Leres, and V. Jacobson. libpcap. Lawrence Berkeley Laboratory, Berkeley, CA, available via anonymous ftp to ftp.ee.lbl.gov.

[15] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–270, January 1993.

[16] Andrew Moore, James Hall, Euan Harris, Christian Kreibich, and Ian Pratt. Architecture of a network monitor. In *Proc. of the Fourth Passive and Active Measurement Workshop (PAM 2003)*, April 2003.

[17] Dave Plonka. FlowScan: A network traffic flow reporting and visualization tool. In *Proceedings of the 2000 USENIX LISA Conference*, 2000.

[18] J. Quittek, T. Zseby, B. Claise, and S. Zander. Requirements for IP Flow Information Export, February 2003. Internet Draft, draft-ietf-ipfix-reqs-09.txt (*work-in-progress)*.

[19] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, November 1999. (software available from *http://www.snort.org/*).

[20] The SCAMPI Consortium. SCAMPI Architecture and components: SCAMPI Deliverable D1.2, 2002. Available from http://www.ist-scampi.org.

[21] Rob Thomas. Monitoring DoS Attacks with the VIP Console and NetFlow v1.0. *http://www.cymru.com/Documents/dos-and-vip.html*.

[22] A. Tirumala and J. Ferguson. iperf 1.2 - The TCP/UDP Bandwidth Measurement Tool, May 2001. http://dast.nlanr.net/Projects/Iperf/.

[23] J. van der Merwe, R. Cceres, Y. Chu, and C. Sreenan. Mmdump - a tool for monitoring internet multimedia traffic. *ACM Computer Communication Review*, 30(4), October 2000.

[24] S. Waldbusser. Remote network monitoring management information base. RFC2819/STD0059, http://www.rfc-editor.org/, May 2000.