University of Crete
Computer Science Department

A Programming Abstraction for Distributed Passive Network Monitoring

Michalis Polychronakis

Master's Thesis

October 2005

University of Crete
Computer Science Department

A Programming Abstraction for Distributed Passive Network Monitoring

Thesis submitted by

Michalis Polychronakis

in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Michalis Polychronakis

Committee approvals: _____
Evangelos P. Markatos
Professor, Thesis Supervisor

_____
Vasilios A. Siris
Assistant Professor

_____
Apostolos Traganitis
Professor

Departmental approval: _____
Dimitris Plexousakis
Associate Professor, Chairman of Graduate Studies

Heraklio, October 2005

Abstract

Network traffic monitoring and measurement has been increasingly used as the major mechanism to ensure the efficient and secure operation of computer networks. Although substantial effort has been put in research and development in this area, current approaches to passive network monitoring focus either on collecting flow-level statistics, which makes them unsuitable for applications that perform fine-grained operations like deep packet inspection, or in full packet capture, which significantly increases their operational overhead. In addition, emerging applications, such as detection of Internet worm outbreaks, detection of Distributed Denial-of-Service attacks, and accurate traffic characterization, would benefit from monitoring data gathered from multiple vantage points across the Internet.

In this thesis we present an expressive programming abstraction for distributed passive network monitoring. The Distributed Monitoring Application Programming Interface (DiMAPI) enables users to clearly communicate their monitoring needs to local or remote passive monitoring sensors, choose only the amount of information they are interested in, and therefore balance the overhead they pay with the amount of information they receive. DiMAPI builds on a generalized network flow abstraction flexible enough to capture emerging application needs, and expressive enough to allow the system to exploit specialized monitoring hardware, where available. Based on our implementation experience and our experimental results, we conclude that DiMAPI has more expressive power than competing approaches, enables the implementation of a wide variety of distributed network monitoring applications, and at the same time achieves significant performance improvements.

Supervisor: Professor Evangelos Markatos

Περίληψη

Η παθητική εποπτεία δικτύων χρησιμοποιείται ολοένα και περισσότερο ως ένας από τους σημαντικότερους μηχανισμούς για την εξασφάλιση της αποδοτικής και ασφαλούς λειτουργίας δικτύων υπολογιστών. Αν και έχουν γίνει αξιόλογες ερευνητικές προσπάθειες στην περιοχή αυτή, οι υπάρχουσες προσεγγίσεις εστιάζουν είτε στη συλλογή στατιστικών επιπέδου ροής, οπότε καθίστανται ακατάλληλες για εφαρμογές που απαιτούν πρόσβαση στα περιεχόμενα των πακέτων του δικτύου, είτε στην πλήρη καταγραφή της κίνησης του δικτύου, οπότε το λειτουργικό κόστος τους αυξάνει σημαντικά. Επιπλέον, επίκαιρες αναδυόμενες εφαρμογές όπως η ανίχνευση ξεσπασμάτων σκουληκιών του Διαδικτύου, η ανίχνευση κατανεμημένων επιθέσεων άρνησης υπηρεσίας, και ο ακριβής χαρακτηρισμός της κυκλοφορίας του δικτύου, θα ωφελούνταν σημαντικά αν υπήρχε η δυνατότητα ταυτόχρονης εποπτείας πολλών διαφορετικών σημείων στο Διαδίκτυο.

Σε αυτή την εργασία προτείνουμε μια προγραμματιστική αφαίρεση μεγάλης εκφραστικής δύναμης για κατανεμημένη παθητική εποπτεία δικτύων. Το Distributed Monitoring Application Programming Interface (DiMAPI) επιτρέπει στους χρήστες να διαβιβάζουν με σαφήνεια τις λειτουργίες παθητικής εποπτείας που επιθυμούν σε τοπικά ή απομακρυσμένα συστήματα εποπτείας, να επιλέγουν μόνο την πληροφορία που επιθυμούν, και έτσι να εξισορροπούν το λειτουργικό κόστος με το ποσό πληροφορία που λαμβάνουν. Το DiMAPI βασίζεται σε μια γενικευμένη αφαίρεση ροής δικτύου (network flow), αρκετά ευέλικτη ώστε να ικανοποιεί τις ανάγκες αναδυόμενων εφαρμογών, και αρκετά εκφραστική ώστε να επιτρέπει στο σύστημα να εκμεταλλεύεται εξειδικευμένες συσκευές εποπτείας δικτύου, σε περιπτώσεις που αυτές είναι διαθέσιμες. Με βάση την εμπειρία μας από την υλοποίηση και την πειραματική αξιολόγηση του συστήματος συμπεραίνουμε ότι το DiMAPI έχει μεγαλύτερη εκφραστική δύναμη από τις υπάρχουσες προσεγγίσεις, επιτρέπει την υλοποίηση μεγάλου εύρους κατανεμημένων εφαρμογών εποπτείας δικτύων, και συγχρόνως επιτυγχάνει σημαντικές βελτιώσεις στην απόδοση του συστήματος.

Επόπτης: Καθηγητής Ευάγγελος Μαρκατος

Acknowledgments

I am deeply grateful to my supervisor, Professor Evangelos Markatos, for his invaluable advice and assistance. He has been extremely patient in explaining the concepts and ideas in the simplest way, and his encouraging words and positive attitude always gave me the strength to go on. I am also grateful to Dr. Kostas Anagnostakis for his constant overnight support and for answering a million questions. A big thanks to them for providing me with such a great research experience.

I would like to thank Arne Øslebø and Panos Trimintzios for their numerous constructive comments and suggestions. Part of the work on the Distributed MAPI has been carried out by Antonis Papadogiannakis and Michalis Foukarakis, therefore I greatly appreciate their support. Spyros Antonatos implemented the Snort signature parser for SIDS, and Themis Bourdenas implemented the database export functionality of SIDS. I thank them a lot.

A big shout out to my friends Manos Moschous, Giorgos Dimitriou, Spyros Antonatos, Dimitris Koukis, Periklis Akritidis, and all the members of the DCS lab at FORTH-ICS.

I would like to thank my parents and my sister for their support and encouragement during all these years.

Finally, I never would have made it through without Marina. I thank her for all the love and understanding.

to Sylvia

Contents

List of Figures

Chapter 1

Introduction

Over the past few years, the Internet has evolved into the dominant communication and information infrastructure. Since the advent of the World Wide Web, the number of users, hosts, domains, and enterprise networks connected to the Internet has been growing explosively. Along with the phenomenal growth of the Internet, the volume and complexity of Internet traffic is constantly increasing, and faster networks are constantly being deployed. Emerging highly distributed applications, such as peer-to-peer systems and Grid computing, demand for increased bandwidth, while their operation relies on advanced communication protocols.

As networks grow larger and more complicated, 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. Network measurements and traffic analysis has been increasingly used to ensure the efficient and secure operation of modern computer networks. Although accurate network monitoring is an essential mechanism for the reliable and efficient operation of our cyber-infrastructure, current traffic monitoring systems do not provide the information needed to support emerging monitoring applications such as detection of Internet worms as soon as they start to spread, detection of Distributed Denial-of-Service attacks, and accurate traffic characterization. Current systems focus mostly on collecting low-level statistics, full packet capture, or in actively measuring latency, bandwidth, and similar properties of network links.

In this thesis, we propose an expressive programming abstraction for distributed passive network traffic monitoring, which enables users to clearly communicate their monitoring needs to local or remote passive monitoring platforms, and facilitates the development of advanced distributed monitoring applications.

1.1 The Need for Effective Traffic Monitoring

The need for elaborate network measurements and traffic analysis, along with increasing link speeds, has exposed limitations in existing network monitoring archi-

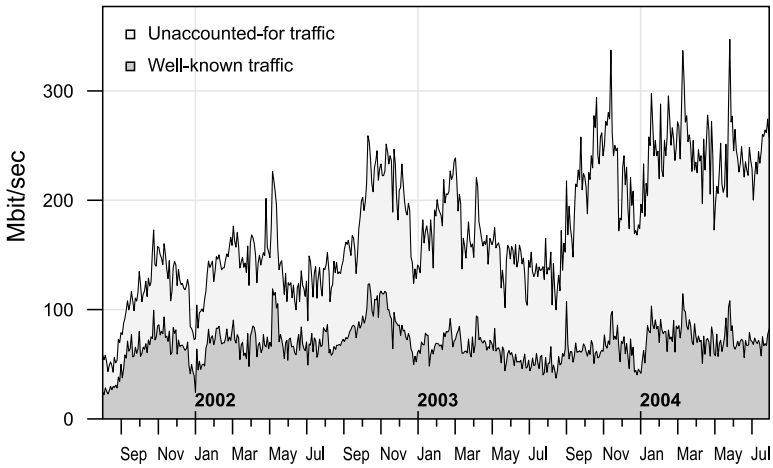

FIGURE 1.1: Outgoing traffic distribution over a three-year period for the University of Wisconsin. All traffic is divided into two broad sets: the Well-known traffic, corresponding to traffic that is destined to, or originates from, static ports associated with well-known applications, and the Unaccounted-for traffic, which is destined to, or originates from, dynamically-generated ports. We see that the Unaccounted-for traffic represents an increasing percentage of the total traffic, reaching as high as 65%.

tectures, which 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 [9], and later incorporated as standard functionality in routers (c.f., Cisco's NetFlow [12]). This approach has been reasonably successful in supporting monitoring functions ranging from accounting to some rather simple forms of Denial of Service attack detection [63]. However, the information contained in flow-level summaries is usually not detailed enough for emerging monitoring applications. For instance, determining per-application network usage is not possible for some of the major new applications that use dynamically allocated ports, such as peer-to-peer file sharing, multimedia streaming, and conferencing applications.

Take a look for example at Figure 1.1, which plots the traffic distribution of the University of Wisconsin, based on information gathered from sampling flow-level summaries over a period of three years. To calculate the distribution of traffic among different applications, each application is assumed to have one or more well known ports: all traffic that is destined to, or originates from, those ports is assumed to be associated with this application. Unfortunately, several recent applications use dynamically allocated ports, and therefore, a large percentage of traffic can not be accounted for. This "unaccounted-for" traffic is shown in Figure 1.1 as the lightly-colored band. It is interesting to see that the relative width of this band has increased between 2002 and 2004, which implies that the percentage of traffic

that can not be accounted for, has significantly increased. Indeed, in 2004, the unaccounted-for traffic has reached as much as 65%, implying that almost seven out of ten IP packets can not be attributed to their applications.

Besides not being sufficient to provide accurate traffic characterization, traditional flow-level traffic summaries are usually not adequate for the type of security monitoring provided by recent intrusion detection systems such as `snort` [53] and `bro` [47]. Such security systems usually need much more information than what is provided by flow-level traffic summaries. For instance, 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 [27] or case-specific solutions, usually supported by specialized hardware [23, 24]. Such approaches have high hardware cost, significant processing needs, and produce vast amounts of data, complicating the task of data analysis.

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.

1.2 The Need for Distributed Network Monitoring

Although accurate network monitoring is getting increasingly important for the reliable and efficient operation of our cyber-infrastructure, current passive traffic monitoring systems do not always provide the information needed to support emerging applications due to their limited view of the network, as they are commonly based on data gathered at a single observation point. For instance, intrusion detection systems usually run on a single monitoring host, which captures and inspects the monitored network traffic.

Several emerging applications would benefit from passive monitoring data gathered at multiple observation points across the network of an organization. For instance, Quality of Service (QoS) applications could be based on traffic characteristics that can be computed only by combining monitoring data from both the ingress and egress nodes of a network. However, a distributed monitoring infrastructure can be extended outside the border of a single organization and span multiple administrative domains across the Internet. The installation of several geographically distributed network monitoring sensors provides a broader view of the network, in which large-scale events could become apparent.

Probably the most important factor for the significant increase of cyber-attacks is the automation of the attack propagation, by means of computer *worms*. A com-

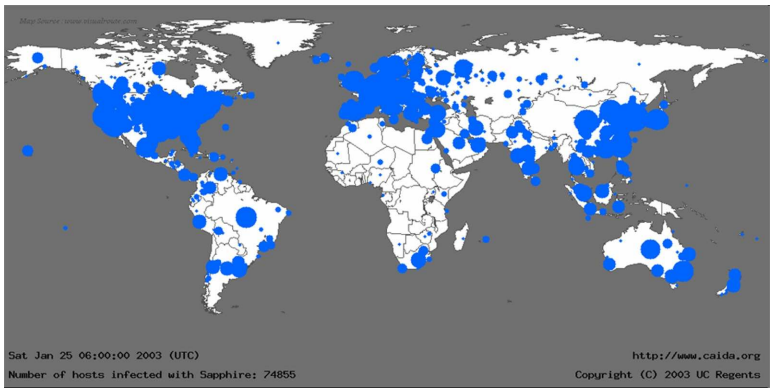

FIGURE 1.2: The geographic coverage of the Sapphire/Slammer worm on January 25th 2003.[1]

puter worm is a self-replicating program that propagates across a network by remotely exploiting defects in widely-used software running on victim hosts. After the infection, the worm executes a predefined routine that performs an operation according to the attacker's will. Since the compromised system is under the complete control of the attacker, the possibilities for its usage are endless. Compromised hosts are commonly used to launch Distributed Denial of Service (DDoS) attacks [41] targeted against well-protected systems. In such attacks, an attacker sends vast amounts of packets at the target server so that no legitimate traffic can reach it. Since the data come from thousands of different compromised hosts, DDoS attacks are very difficult to stop.

Computer worms outbreaks have been plaguing the Internet for the last several years, representing a major threat to the security of our cyber-infrastructure. Recent worm outbreaks have practically demonstrated that worms can infect tens of thousands of computers in as little as thirty minutes. Furthermore, research studies have shown that a carefully implemented *flash* worm could compromise the entire set of vulnerable hosts in under 30 seconds [58], or less [57].

Indeed, the Sapphire/Slammer worm [11, 43, 44] infected more than 90% of the vulnerable hosts within 10 minutes. Eventually, more than 75.000 hosts were infected in no more than 30 minutes, causing devastating effects [13,55]. Figure 1.2 shows the geographic coverage of the Sapphire worm, half an hour after the worm was released. It is clearly depicted that the worm infected computers practically all over the globe. By comparison, Sapphire was two orders of magnitude faster than the CodeRed worm, which infected over 359.000 hosts [45, 73].

Clearly, with the ever-increasing connectivity and bandwidth of today's Internet infrastructure, self-propagating worms can spread across the Internet rapidly, rendering any human-initiated intervention to detect, identify, and stop a new worm outbreak practically ineffective. A single monitoring host is not sufficient for the

[1] © CAIDA – `http://www.caida.org/`

early warning and detection of a worm outbreak, since the worm may have already spread irreversibly until it reaches that host. However, recent research efforts [69,70,72] have demonstrated that a large-scale monitoring infrastructure can be used for building Internet worm detection systems based on distributed cooperative monitors. Furthermore, distributed DoS attack detection applications would also benefit from monitoring data gathered at multiple vantage points across the Internet.

Another practice of attackers is to install *backdoors* on victim machines, i.e., programs which allow them to send commands to the victim machine and operate it remotely. Using coordinated management through readily available mediums, such as Internet Relay Chat (IRC) and Instant Messaging (IM), attackers build and control networks comprising thousands of compromised machines, also known as *botnets* [33,46]. Besides acting as launching points for attacks and other malicious activities [60], these networks allow attackers to hide their tracks, and make tracing their real source extremely difficult.

The situation described so far gets worse as the software quality remains low. Vendors of commercial off-the-shelf software products usually give security design a secondary priority, in favor of rich features, time to market, performance, and overall cost. As a consequence, new vulnerabilities are continually being discovered in ubiquitous applications. Experience has shown that once a vulnerability is discovered in a popular application and the relevant exploit comes out, a worm outbreak based on this vulnerability is usually a matter of time [25]. The relevant patches provided by vendors to fix the flaws normally come late, since they require time-consuming human intervention. Furthermore, administrators are usually reluctant to install them before they first verify through extensive testing that they do not cause any stability problems [51].

It is clear from the above that *distributed* network traffic monitoring is becoming necessary for understanding the performance of modern networks and for protecting them against possible security breaches. This promising approach involves the cooperation of many, possibly heterogeneous, monitoring hosts distributed over a network of several collaborating autonomous systems. Such infrastructures can elevate our perception of the Internet and lead to its better use in the long-run, by combining and correlating the data gathered at each host. This provides a broader perspective in which related incidents become easier to detect.

For instance, by capitalizing on the combination of information from several different geographical regions, such a wide-area network of cooperative traffic monitoring sensors can increase the speed and accuracy of early worm detection and fingerprinting. Furthermore, besides contributing towards increasing the security of the cyber-space, distributed monitoring systems can contribute towards increasing our understanding of Internet traffic, which has been significantly diminished with the advent of modern Internet-centric systems, such as Peer-to-Peer and GRID-enabled applications. Wide-area application debugging, for example, which requires information across the whole network, can be facilitated by a distributed monitoring infrastructure. During the development of a Peer-to-Peer sys-

tem, for instance, a distributed monitoring infrastructure could be used to extract information regarding the request and response latencies and the induced traffic patterns. Such characteristics are highly unpredictable in wide-area networks due to the best-effort nature of IP networks and the constant network changes. Additionally, application level characteristics such as the query cache hit ratio can be easier to derive. Finally, user mobility necessitates distributed monitoring due to nomadic users who frequently change locations across different networks.

A distributed infrastructure also offers benefits inherent to its architecture, such as scalability and resilience to attacks. A network of monitoring sensors is easy to expand and reduces the cost per-participant, considering the related economies of scale. At the same time, such a system comprises numerous hosts, and thus, withstands denial of service attacks targeted to harm its availability. Additionally, a distributed infrastructure makes it harder for attackers to evade detection by spotting and blacklisting specific monitoring hosts, or portions of the address space that are being monitored for random attacks.

1.3 Contributions

In this thesis, we present an *expressive* programming abstraction for distributed network traffic monitoring, which enables users to clearly communicate their monitoring needs to local or remote passive monitoring platforms, and facilitates the development of distributed monitoring applications.

The Monitoring Application Programming Interface (MAPI) [49] builds on a generalized *network flow* abstraction that allows users to express complex monitoring needs, choose only the amount of information they are interested in, and therefore balance the overhead they pay with the amount of information they receive. The major contribution of MAPI is that it elevates network flows to a *first-class status*, enabling programmers to define and operate on flows in a flexible and efficient way. Although traditional environments define network flows in a limited and restrictive manner[2], MAPI enables applications to define network flows in a much more expressive fashion. 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, similarly to programmable monitoring systems [4, 38].

The Distributed Monitoring Application Programming Interface (DiMAPI) extends MAPI to facilitate the programming and coordination of distributed monitoring sensors in a flexible and efficient way. The main contribution of DiMAPI is that it extends the network flow abstraction to enable the definition of network flows that receive traffic from many monitoring interfaces through the notion of the network flow *scope*. DiMAPI provides a uniform platform for building distributed monitoring applications that utilize several geographically distributed—and possibly heterogeneous—passive monitoring sensors.

[2]Loosely speaking, in traditional environments, a network flow is defined as the set of all packets exchanged between two IP addresses and two given ports.

1.4 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 introduces the network flow abstraction, presents the most important MAPI operations, and describes our prototype implementation. Chapter 3 discusses distributed passive network monitoring and describes the extended version of MAPI which supports distributed monitoring. Chapter 4 discusses in detail the software architecture of DiMAPI. Chapter 5 presents our experimental evaluation, and Chapter 6 presents some real-world applications build on top of MAPI and DiMAPI. Finally, Chapter 7 presents related work, and Chapter 8 concludes the thesis.

Chapter 2

MAPI: A Network Traffic Monitoring API

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

Towards these targets, MAPI builds on a simple, yet powerful, abstraction: the *network flow*. 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, the simplest network flow consists of "*all packets captured by a monitoring interface.*" A more complex flow may be composed of "all the TCP packets between a *pair of subnets that contain the string* `User-agent: Mozilla/5.0`."

Figure 2.1 illustrates the concept of the network flow abstraction with some simple examples. On the top of the figure we see a portion of the monitored network traffic, and below, three different network flows. Each flow consists of a subset of the packets of the monitored traffic. For instance, network flow A consists of "*all packets with destination port* `80`," i.e., packets destined to some web server. Network flow B comprises "*all HTTP GET request packets*," while C contains only "*packets of the CodeRed worm*" [22]. Note that the packets of the network flow B are a subset of A, and similarly, CodeRed packets are a subset of all HTTP GET requests. The network flow abstraction allows for fine-grained control of the conditions that the packets of a flow should satisfy.

The approach to network flows in MAPI is therefore fundamentally different from existing flow-based models, e.g., NetFlow [12] or IPFIX [50], which 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

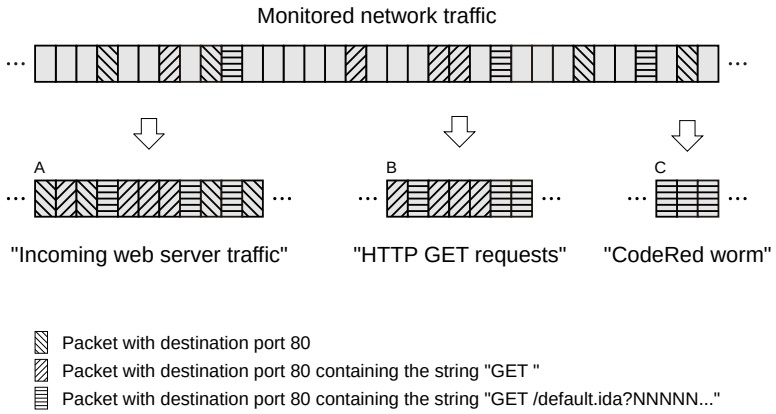

FIGURE 2.1: Examples of network flows with different characteristics.

existing models, MAPI gives the network flow a *first-class status*: flows are named entities, which 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 the packets of a flow, apply functions to flows, and retrieve information and statistics from a flow.

Using this definition for the network flow, users can express a wide variety of monitoring operations. For instance, MAPI flows allow users to develop simple intrusion detection schemes that require content (payload) inspection. In contrast, traditional approaches to network flows like NetFlow, do not have the means of providing the advanced functions required for this task. Going back to the example of Figure 2.1, although NetFlow could be used to capture several characteristics of the traffic defined by network flow `A`, it is not sufficient for monitoring the traffic of flows `B` and `C`.

2.1 MAPI Operations

In this section we present an overview of the most important operations provided by MAPI, summarized in Figure 2.2. A complete list of the available functions, along with detailed descriptions, is provided in `mapi(3)` and `mapi_stdlib(3)` man pages [1, 61].

2.1.1 Creating and Terminating Network Flows

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

```
int mapi_create_flow(char *dev)
```

This call creates a network flow and returns a flow descriptor `fd` that refers to it. By default, the newly created network flow consists of all network packets that

- `int mapi_create_flow(char *dev)`
  Creates a new network flow.
- `int mapi_connect(int fd)`
  Connects to the flow `fd` to start receiving information.
- `int mapi_close_flow(int fd)`
  Closes the flow defined by descriptor `fd`.
- `int mapi_apply_function(int fd, char * funct, ...)`
  Applies the function `funct` to all the packets of the flow `fd`.
- `void * mapi_read_results(int fd, int fid)`
  Receives the results computed by the application of the function `fid` in the packets of the flow `fd`.
- `struct mapipkt * mapi_get_next_packet(int fd, int fid)`
  Reads the next packet of the flow `fd`.
- `int mapi_loop(int fd, int fid, int cnt, mapi_handler callback)`
  Invokes the handler `callback` for each of the packets of the flow `fd`

FIGURE 2.2: The most frequently used MAPI calls.

go through network device `dev`. The packets of this flow can be further restricted to those that satisfy an appropriate filter or some other condition, as described in Section 2.1.2.

Besides creating a network flow, monitoring applications may also close a flow when it is no longer needed:

```
int mapi_close_flow(int fd)
```

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

2.1.2 Applying Functions to Network Flows

The abstraction of the network flow allows users to treat packets belonging to different flows in different ways. For example, after specifying which packets will constitute the flow, a user may be interested in *capturing* the packets (e.g., to record an intrusion attempt), or in just *counting* the number of packets and their lengths (e.g., to measure the bandwidth usage of an application), or in *sampling* the packets (e.g., to find the IP addresses that generate most of the traffic).

MAPI allows users to clearly communicate to the underlying monitoring system these different monitoring needs, by allowing the association of functions with network flows:

```
int mapi_apply_function(int fd, char * funct, ...)
```

The above association applies the function `funct` to every packet of the network flow `fd`, and returns a relevant function descriptor `fid`. Depending on the

applied function, additional arguments may be passed. 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 a broad range of standard monitoring needs through the MAPI Standard Library (`stdlib`) [61].

Several functions are provided for restricting the packets that will constitute a network flow. For example, applying the `BPF_FILTER` function with parameter `"tcp and dst port 80"` restricts the packets of a network flow to the TCP packets destined to port `80`, as in flow `A` of Figure 2.1. `STR_SEARCH` can be used to restrict the packets of a flow to only those that contain a specified byte sequence. Network flows `B` and `C` in Figure 2.1 would be configured by applying both `BPF_FILTER` and `STR_SEARCH`.

Many other functions are provided for processing the traffic of a flow. Such functions include `PKT_COUNTER` and `BYTE_COUNTER`, which count the number of packets and bytes of a flow, respectively, `SAMPLE`, which can be used to sample packets, and `HASH`, which computes a digest of each packet.

Although these functions enable users to process packets and compute network traffic metrics without receiving the actual packets in the address space of the application, they must somehow communicate their results back to the application. For example, a user that has applied the function `PKT_COUNTER` to a network 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 are applied to the packets of a flow will write their results into this data structure. The user who is interested in reading the results will read that data structure using:

```
void * mapi_read_results(int fd, int fid)
```

The above function returns a pointer to the memory where the result of the function with the identifier `fid`, which has been applied to the network flow `fd`, has been stored.

2.1.3 Reading Packets from a Flow

Once a flow is established, packets belonging to that flow can be read one-at-a-time using the following blocking call:

```
struct mapipkt * mapi_get_next_pkt(int fd, int fid)
```

The above function reads the next packet that belongs to flow `fd`. In order to read packets, the function `TO_BUFFER` (which returns the relevant `fid` parameter) must have previously been applied to the flow. `TO_BUFFER` instructs the monitoring system to store the captured packets into a shared memory area, from where the user can directly read the packet, supporting this way efficient zero-copy packet capturing platforms [20, 23, 24].

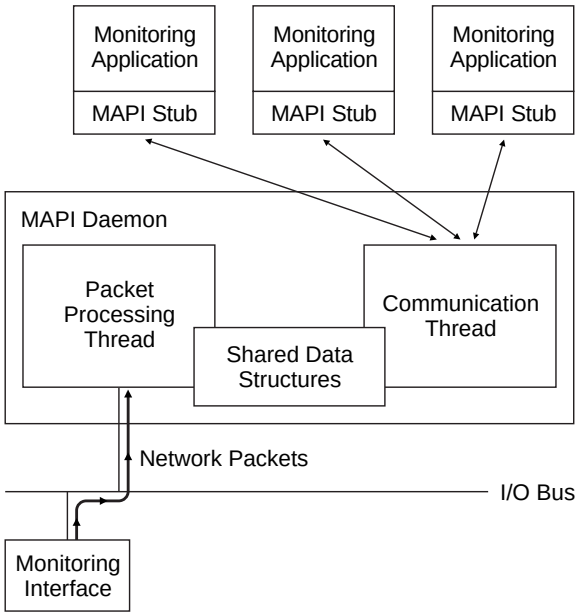

FIGURE 2.3: MAPI Daemon Architecture.

If the user does not want to read one packet at-a-time and possibly block, she may register a callback function that will be called when a packet to the specific flow is available:

```
int mapi_loop(int fd, int fid, 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`.

2.2 Software Structure

Figure 2.3 shows the main modules of the MAPI system. On the top of the Figure we see a set of monitoring applications that, via the MAPI stub, communicate with the MAPI daemon: a monitoring process running in a separate address space. The monitoring daemon, called `mapid`, was chosen to be implemented as a user-level process, instead of a library connected to an operating system module, because it lead to faster implementation, due to shorter debugging cycles, and to a more robust system, due to fault isolation. Indeed, debugging operating system kernels usually involves slow kernel rebooting processes which adds significant slowdown in the overall development cycle. Besides achieving rapid prototyping, our choice of implementing the daemon as a user-level process also increased the robustness of the system, since any fatal daemon bug would only crash the monitoring daemon process and not the entire computer.

The daemon, which has exclusive access to the captured packets, consists of two threads: one data thread for packet processing, and one control thread 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, substring search, or other functions are applied, and finally the packet may be sent to the application, dumped to disk by the daemon, or dropped. In our prototype implementation, filtering is accomplished using the `bpf_filter()` function of the `libpcap` library [40], which applies a compiled BPF filter to captured packets in user level. Each compiled filter is stored into the corresponding flow structure.

All communication between the daemon and the monitoring applications is handled by the "control thread." This thread constantly listens for requests made by the monitoring application through calls of MAPI functions, and sets up the appropriate shared data structures. When monitoring applications need to read data, the control thread reads these data from the shared data structures. Communication between the MAPI stub and `mapid` is performed through Unix sockets.

2.3 MAPI Example: Simple Packet Count

In this section we present a simple MAPI-based application which introduces the concept of the network flow and demonstrates the basic steps that must be taken in order to create and use a network flow. In the following application, a network flow is used for counting the number of packets destined to a web server during a certain time period.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include "mapi.h"
5
6  int main() {
7
8      int fd, fid;
9      unsigned long long *cnt;
10
11     /* create a flow using the eth0 interface */
12     fd = mapi_create_flow("eth0");
13     if (fd < 0) {
14         printf("Could not create flow\n");
15         exit(EXIT_FAILURE);
16     }
17
18     /* keep only the packets directed to the web server */
19     mapi_apply_function(fd, "BPF_FILTER", "tcp and dst port 80");
20
21     /* and just count them */
```

```
22      fid = mapi_apply_function(fd, "PKT_COUNTER");
23
24      /* connect to the flow */
25      if(mapi_connect(fd) < 0) {
26          printf("Could not connect to flow %d\n", fd);
27          exit(EXIT_FAILURE);
28      }
29
30      sleep(10);
31
32      /* read the results of the applied PKT_COUNTER function */
33      cnt = (unsigned long long *)mapi_read_results(fd, fid);
34      printf("pkts:%llu\n", *cnt);
35
36      mapi_close_flow(fd);
37      return 0;
38  }
```

The control flow of the code is as follows: We begin by creating a network flow (line 12) that will receive the packets we are interested in. We specify that we are going to use the `eth0` network interface for monitoring the traffic. For a different monitoring adapter we would use something like `/dev/scampi/0` in case of a Scampi adapter [14, 15], or `/dev/dag0` for a DAG card [24], depending on the configuration. We store the returned flow descriptor in the variable `fd` for future reference to the flow.

In the next step, we restrict the packets of flow to those destined to our web server, by applying the function `BPF_FILTER` (line 19) using the filter `tcp and dst port 80`. The filtering expression is written using the `tcpdump(8)` syntax [62]. Since we are interested in just counting the packets, we also apply the `PKT_COUNTER` function (line 22). In order to later read the results of that function, we store the returned function descriptor in `fid`.

The final step is to start the operation of the network flow by connecting to it (line 25). The call to `mapi_connect()` actually activates the flow inside `mapid`, which then starts processing the monitored traffic according to the specifications of the flow. In our case, it simply keeps a count of the packets that match the filtering condition.

After 10 seconds, we read the packet count by passing the relevant flow descriptor `fd` and function descriptor `fid` to `mapi_read_results()` (line 33). Our work is done, so we close the network flow in order to free the resources allocated in `mapid` (line 36).

Chapter 3

Distributed Passive Network Monitoring

The need for elaborate monitoring of large-scale network events and characteristics requires the cooperation of many, possibly heterogeneous, monitoring sensors, distributed over a wide-area network, or several collaborating Autonomous Systems (AS). In such an environment, the processing and correlation of the data gathered at each sensor gives a broader perspective of the state of the monitored network, in which related events become easier to identify.

Figure 3.1 illustrates a high-level view of such a distributed passive network monitoring infrastructure. Monitoring sensors are distributed across several autonomous systems, with each AS having one or more monitoring sensors. Each sensor may monitor the link between the AS and the Internet (as in AS 1 and 3), or an internal link of a local sub-network (as in AS 2). An authorized user, who may not be located in one of the participating ASes, can run monitoring applications that require the involvement of an arbitrary number of the available monitoring sensors.

In order to take advantage of information from multiple vantage points, distributed monitoring applications need concurrent access to the remote monitoring sensors. In this chapter, we present the Distributed version of MAPI (DiMAPI), which fulfils this requirement by facilitating the programming and coordination of several remote sensors from within a single monitoring application. This is achieved by building on the abstraction of the network flow introduced with MAPI. However, MAPI supports the creation of network flows associated with a *single* local monitoring interface, and thus, in MAPI, a network flow receives network packets that are always captured at a single monitoring point.

One of the main novelties of DiMAPI is the introduction of the network flow *scope*, a new network flow attribute. In DiMAPI, each flow is associated with a scope, which defines a set of monitoring interfaces that are collectively used for network traffic monitoring. Generally, given an input packet stream, a network flow is defined as a sequence of packets that satisfy a given set of conditions. In

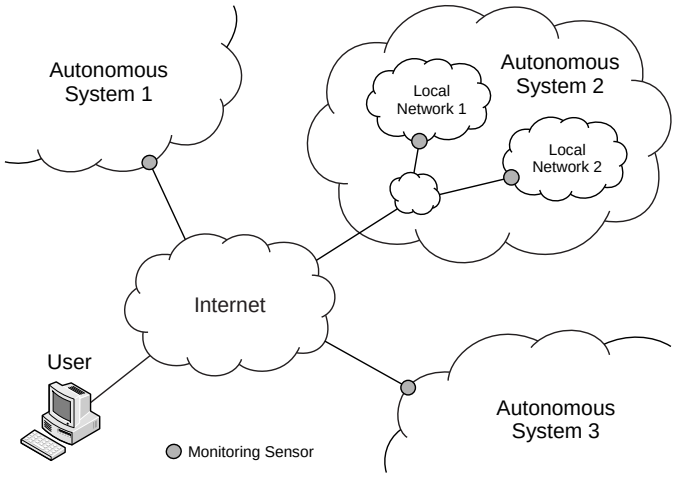

FIGURE 3.1: A high-level view of a distributed passive network monitoring infrastructure across several collaborating autonomous systems.

MAPI, the input packet stream comes from a single monitoring interface. The notion of scope allows a network flow to receive packets from several monitoring interfaces. With this definition, the abstraction of the network flow remains intact: a network flow with scope is still a subset of the packets of an input packet stream. However, the input packet stream over which the network flow is defined may come from more than one monitoring points. In this way, when an application applies functions to manipulate or extract information from a network flow with a scope of multiple sensors, effectively it manipulates and extracts information concurrently from all these monitoring points.

3.1 DiMAPI: Extending MAPI for Distributed Network Monitoring

MAPI supports the operation of monitoring applications that run on the same computer that hosts the monitoring hardware. Monitoring applications address the monitoring platform throughout MAPI. An application uses the functions defined in the MAPI interface to configure the monitoring daemon and retrieve results from it. In DiMAPI, however, whether this daemon is running locally, at the same host with the application, or remotely at a different host, should be of no concern to the functionality of the application.

In order to support the abstraction of scope in DiMAPI, the interface and implementation of `mapi_create_flow()` function has been extended to support the definition of multiple remote monitoring interfaces. A remote monitoring interface can be defined as a `host:interface` pair, where `host` is the host name or IP address of the remote sensor and `interface` is the device name of the monitoring interface or the name of a packet trace file. The scope of a network flow

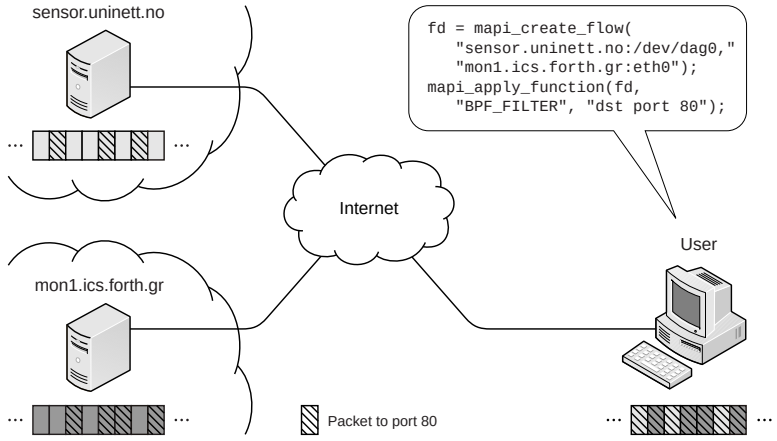

FIGURE 3.2: An example of a network flow with a scope of multiple sensors. A user application manipulates a network flow associated with two remote monitoring sensors, located in different administrative domains.

is defined by concatenating several comma-separated `host:interface` pairs as a string argument to `mapi_create_flow()`. For example, the following call creates a network flow associated with two monitoring interfaces located at two different hosts across the Internet:

```
fd = mapi_create_flow("sensor.uninett.no:/dev/dag0,"
                      "mon1.ics.forth.gr:eth0");
```

The notion of scope is illustrated in the example of Figure 3.2. A monitoring application creates a network flow associated with two remote sensors located in two different organizations, FORTH and UNINETT. The user's monitoring application then applies the function `BPF_FILTER` in order to restrict the packets of the flow to only those that are destined to some web server (some code has been omitted for clarity). As a result, the network flow consists of web packets that are captured from both UNINETT's and FORTH's sensors.

The scope abstraction also allows the creation of flows associated with multiple interfaces (or trace files) located at the same host. For example, the following call creates a network flow associated with a commodity Ethernet interface and a DAG card, both installed on the host `mon2.ics.forth.gr`:

```
fd = mapi_create_flow("mon2.ics.forth.gr:eth1,"
                      "mon2.ics.forth.gr:/dev/dag0");
```

Note that the scope notation in DiMAPI preserves the semantics of the existing `mapi_create_flow()` function, ensuring backwards compatibility with existing MAPI applications. A local network flow can still be created by specifying one monitoring interface without prepending a host.

3.2 DiMAPI Example: Measuring HTTP Requests

In this section we describe an example monitoring application built on top of DiMAPI. Since the structure is similar to the example presented in Section 2.3, we have omitted certain parts of the code. The main difference here, besides the different flow configuration, is that the network flow is configured with a scope of two remote monitoring sensors. The following pseudocode illustrates a simple DiMAPI application that counts the number of HTTP GET requests destined to the web servers of several monitored networks within a predefined interval.

```
1  fd = mapi_create_flow("host1:eth2, host2:/dev/dag0");
2
3  /* keep only packets directed to a web server */
4  mapi_apply_function(fd, "BPF_FILTER", "tcp and dst port 80");
5
6  /* that contain an HTTP GET request */
7  mapi_apply_function(fd, "STR_SEARCH", "GET /", 0, 50);
8
9  /* and just count them */
10 fid = mapi_apply_function(fd, "PKT_COUNTER");
11
12 mapi_connect(fd);
13 sleep(10);
14
15 bytes = mapi_read_results(fd, fid);
16 ...
```

The above application operates as follows. We initially define a network flow with a scope of two remote monitoring sensors (line 1). Then, we restrict the packets of the flow to only those destined to some web server, by applying the `BPF_FILTER` function (line 4). Since we are interested only in HTTP GET requests, we use the function `STR_SEARCH` to further restrict the packets of the flow to those containing the characteristic string `"GET /"` (line 7) within the first 50 bytes of the payload, starting at offset 0. After specifying the characteristics of the network flow, we instruct the monitoring system that we are interested in just counting the number of packets of the flow, by applying the `PKT_COUNTER` function (line 10). Finally, we activate the flow (line 12). After 10 seconds, the application reads the result by calling `mapi_read_results()` (line 15).

Implementing the above—rather simple—distributed monitoring application using existing tools and libraries would have been a tedious process, resulting in longer code and higher overheads. For example, we could use passive monitoring tools running on top of the `libpcap` [40] library, such as `snort` [53] or `ngrep` [52], for capturing the HTTP GET packets at each sensor, then dump the packet records to a file or print them in `stdout`, and periodically report the packet count using some script. At the end-host, we could use some additional scripts for starting and stopping the remote monitoring applications at each remote sensor through some remote shell, such as `ssh`, and for retrieving and collectively reporting the results from all the sensors.

Clearly, such custom schemes do not scale well and cannot offer the ease of use and flexibility of DiMAPI for building distributed monitoring applications. Furthermore, DiMAPI exploits any specialized hardware available at the monitoring sensors and efficiently shares the monitoring infrastructure among many users, by grouping and optimizing their monitoring needs into the monitoring daemon that runs at each sensor [49], as discussed in Section 2.2.

Chapter 4

DiMAPI Architecture

In this chapter we discuss in detail the software architecture of DiMAPI. Figure 4.1 illustrates the architecture of a monitoring sensor that supports DiMAPI. The host of the monitoring sensor is equipped with one or more interfaces for packet capture, and optionally an additional network interface for remote access. The latter is the sensor's "control" interface, and ideally it should be separate from the network "taps." Packets are captured and processed by `mapid`, as discussed in Section 2.2. Local monitoring applications still communicate directly with `mapid` as before, via the subset of the DiMAPI stub that implements the MAPI functionality.

Remote applications must be able to communicate their monitoring requirements to each sensor through the Internet. A straightforward approach for enabling applications to communicate with a remote sensor would be to modify `mapid` to interact directly with the remote applications through the DiMAPI stub. This would be in a similar way to local applications, which interact directly with `mapid` through shared memory and UNIX sockets. This design alternative has the advantage that it maintains the basic architecture of the monitoring agent and the applications, since the DiMAPI stub still communicates directly with `mapid`. However, this design requires changes to be made to `mapid`, which poses several risks. Indeed, `mapid` is a complex part of the software monitoring architecture and is already responsible for handling important "heavy-duty" tasks, as this is where all the processing of the monitoring requirements of the user applications takes place. The monitoring daemon should keep up with intensive high-speed packet processing. Besides increasing its complexity, extending `mapid` to handle communication directly with remote clients would probably introduce additional performance overhead. Furthermore, allowing remote clients to connect directly to `mapid`, which has exclusive access to the captured packets, may introduce significant security risks.

For the above reasons, we have chosen an alternative design, which avoids any modifications to `mapid`. This is possible by introducing an *intermediate* agent between `mapid` and the remote applications, for handling all communication between them, as depicted in Figure 4.1. This *communication agent* (`commd`), which

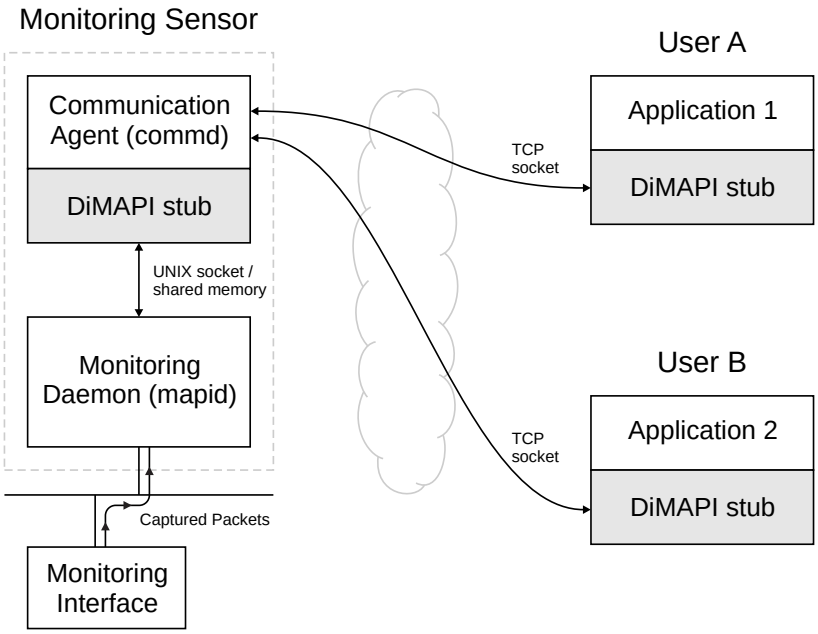

FIGURE 4.1: Architecture of a DiMAPI monitoring sensor. User applications may run remotely on hosts other than the monitoring sensor, and interact with `mapid` through the Communication Agent.

runs on the same host with `mapid`, acts as a proxy for remote applications, forwarding their monitoring requests to `mapid`, and sending back to them the computed results. The presence of `commd` is completely transparent to user applications, which continue to operate as if they were interacting directly with `mapid`—only the DiMAPI stub is aware of the presence of `commd`. Furthermore, the presence of `commd` is also transparent to `mapid`, since `commd` operates as a common local monitoring application.

The main benefit of this design is that it does not require any changes to `mapid`. The only software component that needs extension and modification is that of the stub. This needs to be extended in order to support the DiMAPI functionality, as discussed in Section 3.1. At the monitoring sensor side, the DiMAPI functionality is solely implemented by `commd`, which is built as a monitoring application that interacts locally with the `mapid`. This allows for a cleaner implementation with shorter debugging cycles, and a more *robust* system due to fault isolation. Indeed, the system becomes more robust, as communication failures will not result in failure of the monitoring processes. Furthermore, in case that the remote monitoring functionality of a sensor is not required any more, it can be easily left out by simply not starting up `commd`.

In the following sections we look more closely into the structure and operation of `commd`, the DiMAPI stub, and the communication mechanism between them. We also discuss security and privacy issues related to our system.

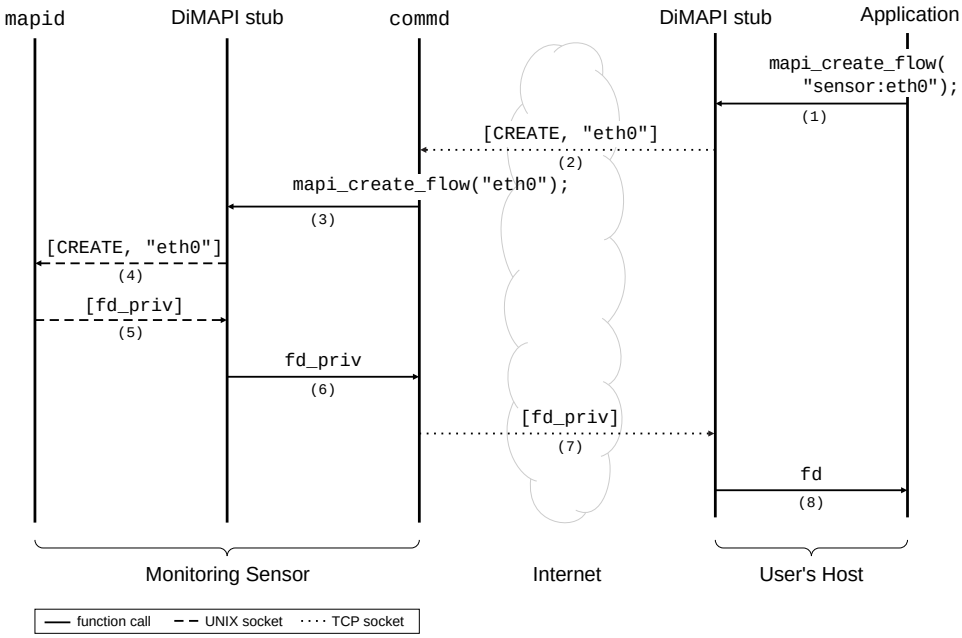

FIGURE 4.2: Control sequence diagram for the remote execution of the function `mapi_create_flow()`.

4.1 Communication Agent

The communication agent runs on the monitoring host and acts as an intermediary between remote applications and `mapid`. Upon the reception of a monitoring request from the DiMAPI stub of a remote application, it forwards the relevant call to the local `mapid`, which in turn processes the request and sends back to the user the computed results, again through `commd`. The communication agent is a simple user-level process implemented on top of DiMAPI, i.e., it looks like an ordinary DiMAPI-based monitoring application. However, its key characteristic is that it can receive monitoring requests from *other* monitoring applications that run on different hosts and are written with DiMAPI. This is achieved by directly handling the control messages sent by the DiMAPI stub of remote applications, and transforming them to the relevant local calls.

The communication agent listens for monitoring requests from DiMAPI applications to a known predefined port. A new thread is spawned for each new remote application, which thereafter handles all the communication between the monitoring application and `commd`. The DiMAPI stub of the remote application sends a control message for each DiMAPI call invocation to `commd`, which in turn repeats the call, though this time the stub of `commd` will interact directly with the `mapid` running on the same host. `commd` then returns the result to the stub of the remote application, which in turn returns it to the user. Note that although `commd` is implemented on top of DiMAPI, it uses only the subset of DiMAPI that is intended

for local monitoring, as discussed in Chapter 2, and communicates with `mapid` solely through shared memory and UNIX sockets, since it never manipulates network flows associated with remote monitoring sensors.

The control sequence diagram in Figure 4.2 shows the operation of the communication agent in more detail, using a concrete example of the control sequence for an invocation of the `mapi_create_flow()` function. Initially, a monitoring application calls `mapi_create_flow()` in order to create a network flow at a remote monitoring sensor (step 1). The DiMAPI stub retrieves the IP address of the sensor and sends a respective control message to the `commd` running on that host through a TCP socket (step 2). The message contains the type of the DiMAPI call to be executed (`CREATE`), along with the monitoring interface that will be used (`eth0`). Upon the receipt of the message, `commd` repeats the call to `mapi_create_flow` (step 3). This time, however, the call is destined to the local `mapid`, thus the stub of `commd` sends the respective message through a UNIX socket (step 4). Assuming a successful creation of the flow, `mapid` returns the flow descriptor `fd_priv` of the newly created flow to the stub of `commd` (step 5), which in turn finishes the execution of the `mapi_create_flow()` call by returning `fd_priv` to `commd` (step 6). The agent constructs a corresponding reply message that contains the flow descriptor, and sends it back to the DiMAPI stub of the user application (step 7).

Finally, the stub of the application has to return a flow identifier back to the user. However, in case that the network flow is associated with more than one monitoring sensors, the DiMAPI stub of the application will receive several flow descriptors, one for each of the monitoring interfaces constituting the scope of the network flow.[1] Consider for example an application that creates two network flows associated with different remote monitoring sensors. Assume also that the MAPI daemons on the two sensors have served the same number of flows so far, and are occupied by the same number of active flows at that moment. Since `mapid` generates the values for the flow descriptors sequentially, starting from the same predefined number, the agents of the two sensors will return to the DiMAPI stub of the application the same flow descriptor for each of the two flows. Clearly, this is not acceptable, since the flow descriptor becomes ambiguous, and does not identify uniquely the network flow anymore. Although each descriptor is unique within the scope of the `mapid` that generated it, it is not unique within the scope of the remote application.

Therefore, the stub generates and returns a new unique flow identifier, and internally stores the mapping between the received flow descriptors and the newly created identifier. Thus, even if two or more remote sensors return the same flow descriptor, the stub always returns to the user a unique flow identifier. In our example, the stub of the user application receives the descriptor `fd_priv`, generates

[1] In that case, steps 2–7 in Figure 4.2 are repeated for each one of the sensors that constitute the scope of the network flow.

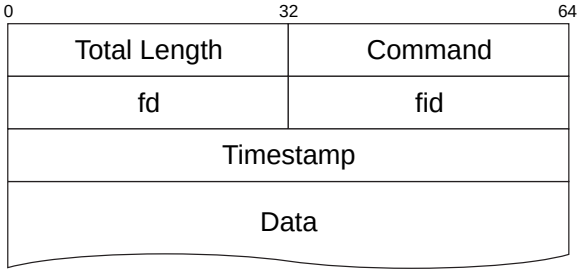

FIGURE 4.3: Format of the control messages exchanged between the DiMAPI stub and `commd`.

the new unique flow descriptor `fd`, saves the mapping between `fd_priv` and `fd`, and finally returns `fd` to the user (step 8).

Although at first sight it may seem that the overhead for a DiMAPI call is quite high, since it results in several control flow transitions, we should stress that most of the above steps are function calls or inter-process communication that takes place on the same host, and thus, incur very small overhead. The operations responsible for the largest part of the cost are the send and receive operations through the TCP socket (steps 2 and 7), which incur an unavoidable overhead due to network latency. We look in more detail into this issue in Chapter 5.

4.2 Communication Protocol

Monitoring applications reside on a host that may be located remotely from the monitoring sensors, probably even in a different administrative domain. The communication protocol between the monitoring sensors and the remote applications is one of the main factors that affect the performance of a distributed monitoring application. Our design target was to have communication with minimal overhead, which scales well over a large number of monitoring sensors.

The DiMAPI stub encapsulates the communication with the remote monitoring sensors. In DiMAPI, all communication between the stub and the monitoring sensors is performed trough TCP sockets. DiMAPI stub library calls exchange control messages with `commd` that describe the operation to be executed. Each message contains all the necessary information for the execution of a function instance. After sending a request, the stub waits for the corresponding acknowledgement from the sensor, indicating the successful completion of the requested action, or a specific error in case of failure.

The format of the messages exchanged between the DiMAPI stub and `commd` is shown in Figure 4.3. Message have variable length, denoted by the field `Total Length`. The `Command` field contains the operation type, sent by the stub to `commd`, or the acknowledgment value for a request that `commd` has processed. It takes values from an enumeration of all message types that can be exchanged between the stub and `commd`. For example, for a call to `mapi_create_flow()`,

the stub will send a message with a `Command` value of `CREATE_FLOW`, for a call to `mapi_apply_function()` `Command` will have the value `APPLY_FUNCTION`, and so on. The field `fd` is the descriptor of the network flow being manipulated, `fid` is the descriptor of the applied function instance being manipulated, and `Timestamp` is a timestamp of the specific moment that the result included in the communication message was produced. Finally, the field `Data` is the only field of variable size, serving several purposes depending on the contents of the `Command` field. More specifically, when the message is a reply (acknowledgement) from `commd` to a call of `mapi_read_results()`, it contains the results of an applied function. If the `Command` field contains a request, sent from the DiMAPI stub, e.g., to apply some function to a network flow, it may contain the arguments of the relevant DiMAPIfunction. For example, in a call to `mapi_apply_funtion()`, it contains the name of the function to be applied along with its arguments.

This implementation is similar to the existing IPC mechanism between local applications and `mapid`. Indeed, whether the MAPI daemon is running locally or remotely is of no concern to the application. The details of the underlying communication mechanism are hidden from the end-user, making Remote MAPI completely transparent to the application.

4.3 DiMAPI Stub

4.3.1 Creating Network Flows using Multiple Remote Sensors

To support the scope functionality, the stub has to be extended for handling communication with many remote sensors concurrently. Consider for example the following call, which creates a network flow at three different remote sensors:

```
fd = mapi_create_flow("sensor.uninett.no:/dev/dag0,"
                      "monitor.cesnet.cz:/dev/scampi/0,"
                      "mon1.ics.forth.gr:eth0");
```

In order to implement this call, the DiMAPI stub has to communicate with the agents running at each of the three remote sensors. This is achieved by sending three control messages, one to each `commd`, through three different TCP sockets. Thus, the following calls will be made by the three agents:

```
Uninett:  fd_uninett = mapi_create_flow("/dev/dag0");
 Cesnet:  fd_cesnet = mapi_create_flow("/dev/scampi/0");
  FORTH:  fd_forth = mapi_create_flow("eth0");
```

In the above example, the creation of one *distributed* network flow from the user application resulted in the creation of three *local* network flows, one at each of the three remote sensors. Assuming that the three flows were created successfully, each `commd` will send back to the DiMAPI stub an acknowledgment message containing the flow descriptor of the flow that it created remotely (`fd_uninett`, `fd_cesnet`, and `fd_forth`, respectively). However, in the client side, the call to `mapi_create_flow` has to return back to the user a single unique identifier for the newly created distributed network flow (`fd` in the above example).

The DiMAPI stub is responsible for generating such a unique flow identifier, and internally storing the remote flow descriptors that it corresponds with. In the above example, the stub will store the mapping between `fd` and [`fd_uninett`, `fd_cesnet`, `fd_forth`]. For subsequent calls that manipulate `fd`, such as the following:

```
int fid = mapi_apply_function(fd, "PKT_COUNTER");
```

the DiMAPI stub will send to the agents of the three sensors the following corresponding messages:

```
Uninett:  [APPLY_FUNCTION, PKT_COUNTER, fd_uninett]
 Cesnet:  [APPLY_FUNCTION, PKT_COUNTER, fd_cesnet]
  FORTH:  [APPLY_FUNCTION, PKT_COUNTER, fd_forth]
```

Since the stub knows each of the remote flow descriptors that constitute `fd`, it can send targeted control messages with the appropriate flow descriptor for each `commd`. Note that the DiMAPI stub stores a similar mapping for the function identifier `fid`, and acts in a similar fashion whenever it is manipulated.

4.3.2 Support for Multi-threaded Applications

The mechanism of sending and receiving control messages has also been reconsidered in DiMAPI. Since the two endpoints are located on different hosts across the Internet, the time interval between the dispatch of a control message and the receipt of the corresponding reply is not constant, and may be several milliseconds long. This makes the exchange of control messages challenging for multi-threaded user applications, which may call several DiMAPI functions at the same time.

Consider for example a user application consisting of two threads, each of which at a given point of time reads the result of a `PKT_COUNTER` function applied at a different network flow. Both flows have been created on the same remote monitoring sensor, and each flow is private to each thread. The stub will send to the `commd` of the remote sensor two control messages, one for each thread, requesting the value of the counter. However, there is no guarantee that the replies from the `commd` will arrive in the same order that the corresponding requests were made, due to the arbitrary delays and routes of the network packets that carry the reply messages. Thus, a reply may be delivered to the wrong call that did not make the corresponding request. This introduces the need for demultiplexing of the incoming messages at the DiMAPI stub.

The receipt of incoming messages in DiMAPI is implemented using a separate "communication thread." This thread is responsible for receiving the replies of pending MAPI calls from remote sensors, and delivering them to the appropriate function. One such thread is created for each remote sensor used by the application (i.e., for each TCP socket created by the stub). Figure 4.4 illustrates a multi-threaded application that has created several network flows at two remote monitoring sensors. Specifically, thread `T1` has created the flows `fd_u1` and `fd_u2` at the sensor with name `sensor.uninett.no`, thread `T2` has created the flow `fd_u3`

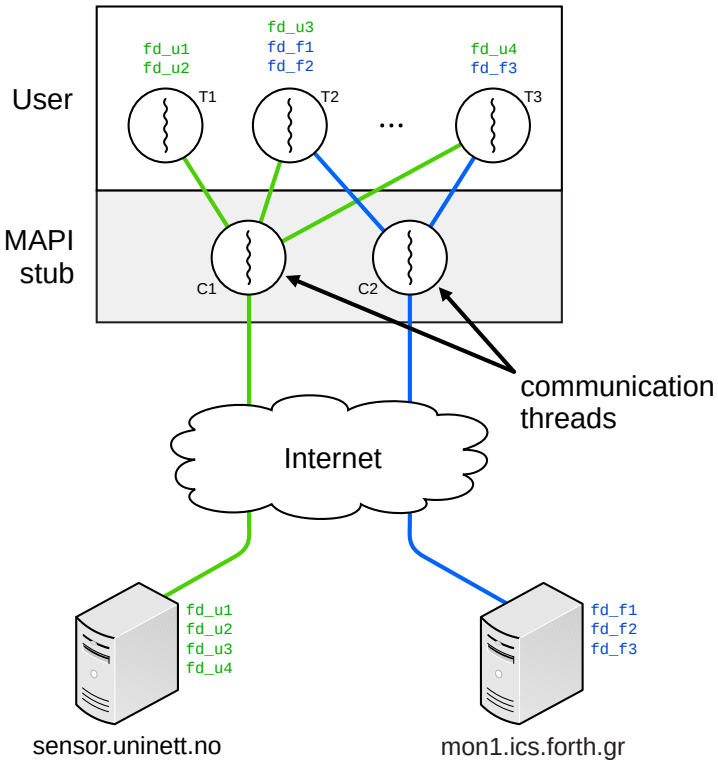

FIGURE 4.4: The role of the communication thread in a multi-threaded application. Each communication thread is responsible for the communication with one remote sensor.

at `sensor.uninett.no` and `fd_f1` and `fd_f2` at `mon1.ics.forth.gr`, and thread `T3` has created the flows `fd_u4` and `fd_f3`, both at different sensors. Messages related to the flows `fd_f1`, `fd_f2`, `fd_f3`, and `fd_f4` are handled by the communication thread `C1`, which delivers them to the appropriate thread (`T1`, `T2`, or `T3`). Correspondingly, messages for flows `fd_u1`, `fd_u2`, and `fd_u3` are handled by the communication thread `C2`, which delivers them to `T2` or `T3`, depending on the flow.

The pseudo-code of Figure 4.5 outlines the implementation of the generic IPC code for DiMAPI calls and the communication thread. A DiMAPI call—`mapi_apply_function` in our example—prepares and sends the control message, and then blocks by pushing down a semaphore variable, as a result of calling `sem_wait`. The communication thread waits infinitely in a loop for incoming replies from the `commd`. When such a reply message arrives, the communication thread looks up the flow for which it is destined, copies the result in a flow-specific buffer, and "wakes up" the blocked MAPI call by calling `sem_up`. When the execution of the blocked call resumes, it retrieves the result from the buffer and

```
int mapi_apply_function(int fd, char *funct, ...) {

    flow_t *flow = list_get(flowlist, fd);

    /* prepare control message */
    /* ... */

    send(host, msg);
    sem_wait(flow->sem);

    /* read reply */
    /* ... */
}

void comm_thread(void *host) {

    while(1) {

        recv(reply);
        flow = list_get(pendinglist, reply.fd);
        copy_result(fd->res, reply);
        sem_up(flow->sem);
    }
}
```

FIGURE 4.5: Pseudo-code for DiMAPI IPC.

processes it accordingly. This implementation guarantees that the incoming messages are always delivered to the call that sent the relevant request.

4.3.3 Fetching Captured Packets

When using MAPI, as presented in Section 2.1.3 a user can retrieve a captured packet by calling

`mapi_get_next_pkt(int fd, int fid)`

where `fd` is the flow descriptor and `fid` is the descriptor of the "TO_BUFFER" function applied on the flow. In DiMAPI, however, when requesting packets from multiple sensors, retrieving packets and delivering them to the application needs careful treatment. To ensure fairness among packets from different sensors, an internal mechanism for handling incoming packets is used as follows. Upon the first `mapi_get_next_pkt()` call, the request is forwarded to all the sensors of the scope. Each sensor is mapped to a slot in an internal buffer that stores incoming packets. Packets from the first sensor go to the first slot, packets from the second sensor go to the second slot, and so on. The first packet that arrives is delivered to the application, and the corresponding slot is emptied In case of consequent `mapi_get_next_pkt()` requests, all slots are checked in a round-robin way. The round-robin check begins from the slot that was emptied in the previous call in order to avoid slot starvation. The next packet request is sent to the sensor whose

slot was emptied in the previous call, which ensures that all slots will be always full, or at least have one pending request.

4.4 Security and Privacy Issues

A large-scale network monitoring infrastructure consisting of many sensors across the Internet is exposed to several threats that may disrupt its operation. Monitoring sensors may become targets of coordinated Denial of Service (DoS) attacks, aiming to prevent legitimate users from receiving a service with acceptable performance, or sophisticated intrusion attempts, aiming to compromise the monitoring hosts. Being exposed to the public Internet, monitoring sensors should have a rigorous security configuration in order to preserve the confidentiality of the monitored network, and resist to attacks that aim to compromise it.

To counter such threats, each sensor is equipped with a firewall, configured using a conservative policy that selectively allows inbound traffic only from the predefined IP addresses of legitimate users. Inbound traffic from any other source is dropped. Since our system is based on the Linux OS, such a policy can be easily implemented using `iptables` [54].

The administrator of each monitoring sensor is responsible for issuing credentials to users who want to access the monitoring sensor with DiMAPI. The credentials specify the usage policy applicable to that user. Whenever a user's monitoring application connects to some monitoring sensor and requests the creation of a network flow, it passes the user's credentials. The monitoring sensor performs *access control* based on the user's request and credentials. In this way, administrator delegates authority to use that sensor, using public key authentication. Access control in our system is based on the KeyNote [6] trust-management system, which allows direct authorization of security-critical actions.

Since all communication between user applications and the remote sensors will be made through public networks across the Internet, special measures must be taken in order to ensure the *confidentiality* of the transferred data. Data transfers through TCP are unprotected against eavesdropping from third-parties that have access to the transmitted packets, since they can reconstruct the TCP stream and recover the transferred data. This would allow an adversary to record DiMAPI's control messages, forge them, and replay them in order to access a monitoring sensor and impersonate a legitimate user. For protection against such threats, any communication between the DiMAPI stub and a remote sensor is encrypted using the Secure Sockets Layer protocol (SSL). For intra-organization applications, where an adversary cannot have access to the internal traffic, encrypted communication may not be necessary, depending on the policy of the organization, and could be replaced by plain TCP, for increased performance.

In a distributed monitoring infrastructure that promotes sharing of captured network packets and traffic statistics between different parties, exchanged data should be *anonymized* before made publicly available for security, privacy, and business

competition concerns that may arise due to the lack of trust between the collaborating parties. The DiMAPI architecture supports an advanced framework for creating and enforcing anonymization policies [59]. Since different users and applications may require different levels of anonymization, the anonymization framework offers increased flexibility by supporting the specification of user and flow specific policies. The anonymization framework, as well as the access control system, is out of the scope of this work.

Chapter 5

Experimental Evaluation

5.1 MAPI Performance 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.

5.1.1 Environment

The experimental environment is shown in Figure 5.1. It consists of three PCs connected to a Gigabit Ethernet switch (an SMC 8506T). The "Source" PC (equipped with a 1.1 GHz Pentium III processor) generates and transmits traffic to the "Destination" PC (equipped with a 2.5 GHz Pentium IV). Traffic consisting of 1460-byte UDP packets is generated at constant rate using `iperf` [64]. The traffic is mirrored by the switch and sent to the "Monitor" 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 GE network traffic monitoring card, and a Gigabit Ethernet Intel Pro 1000 MT network interface. The host operating system of the monitor PC is Debian Linux 3.0, kernel version 2.4.20.

The prototype MAPI implementation runs on top of two different network monitoring platforms: an Intel Pro 1000 MT (Gigabit Ethernet) desktop adapter, and a DAG 4.2 GE monitoring card for Gigabit Ethernet. The two platforms represent two widely different points in the hardware spectrum of available traffic capture adapters. Indeed, 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 operating system of the host computer. Thus, the packet capturing process using the Intel Pro 1000 MT is mostly based in operating system software and therefore adds noticeable overhead to the overall monitoring task.

On the other hand, the DAG 4.2 GE monitoring card has dedicated hardware which enables passive full packet capture at the speed of 1 Gbit/s. 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

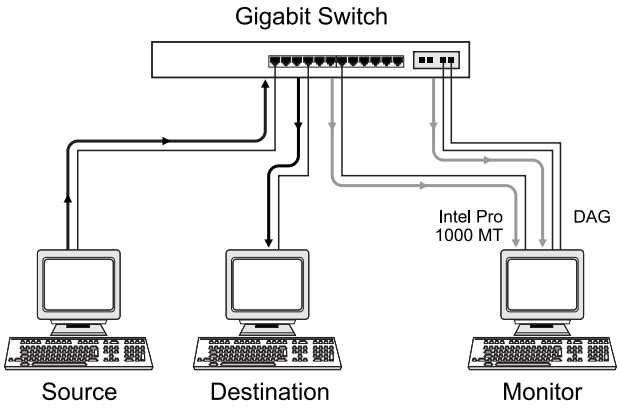

FIGURE 5.1: Experimental environment to test the performance of the MAPI implementation over the NIC and the DAG card.

	NIC		DAG	
	`libpcap`	MAPI	`libpcap`	MAPI
Cycles per Packet	11897	13082	453	235
Instructions per Packet	212	239	54	28

TABLE 5.1: Packet fetching performance for MAPI and `pcap`.

avoids costly interrupt and protocol 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.

Therefore, the two packet capture adapters represent two different design decisions in the search for balance between cost and speed: The Intel Pro 1000 MT is a low-cost adapter which adds noticeable overhead due to software intervention, while the DAG 4.2 GE card is significantly faster, but noticeably more expensive as well. We expect that these two different hardware platforms will stress different aspects of the MAPI design, will strain the MAPI implementation, and will hopefully demonstrate the portability of our MAPI approach.

5.1.2 Baseline 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 `libpcap` delegates these functions to user applications. Thus, the basic operating cost per received packet for MAPI is the cost to receive

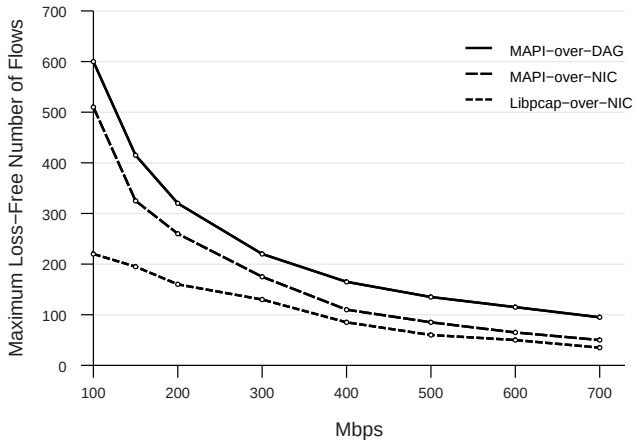

FIGURE 5.2: Maximum number of flows when gathering packet and byte statistics for each flow.

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 [36]. Table 5.1 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 5.1 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.

5.1.3 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 5.2 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 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 5.2 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 5.2 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.

5.1.4 Supporting multiple sampling 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).[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 5.3. The metric of interest here is CPU idle time as a

[1] 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.

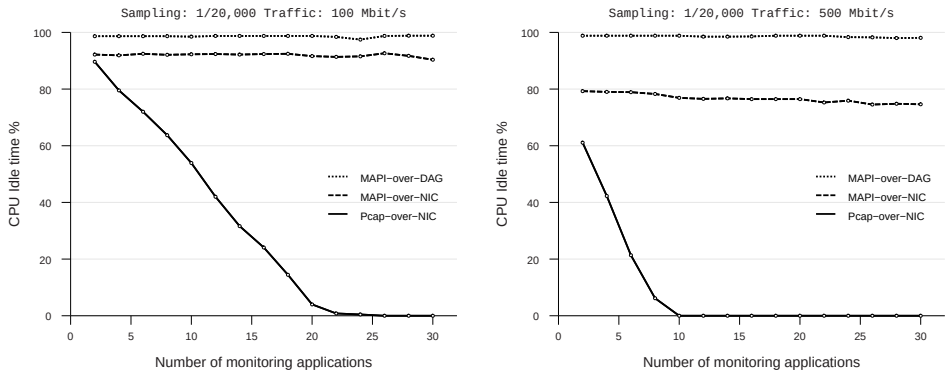

FIGURE 5.3: Performance of multiple packet sampling applications.

function of the number of monitoring applications. Figure 5.3 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 5.4 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 5.3 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 5.3 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.

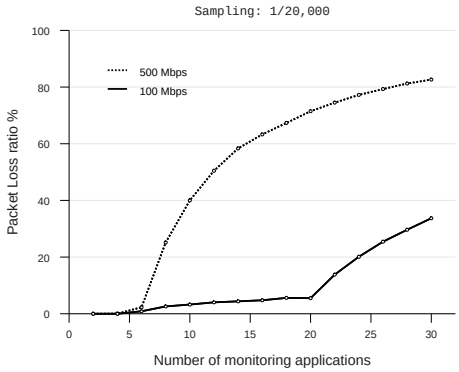

FIGURE 5.4: Packet Loss ratio of `libpcap` implementation for multiple packet sampling applications. Note that the packet loss ratio for both MAPI-over-DAG and MAPI-over-NIC was under 1%.

5.1.5 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 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 [8].

Figure 5.5 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.

5.2 DiMAPI Network-Level Performance

In this section we experimentally evaluate several performance aspects of DiMAPI. Our analysis consists of network overhead and response latency measurements,

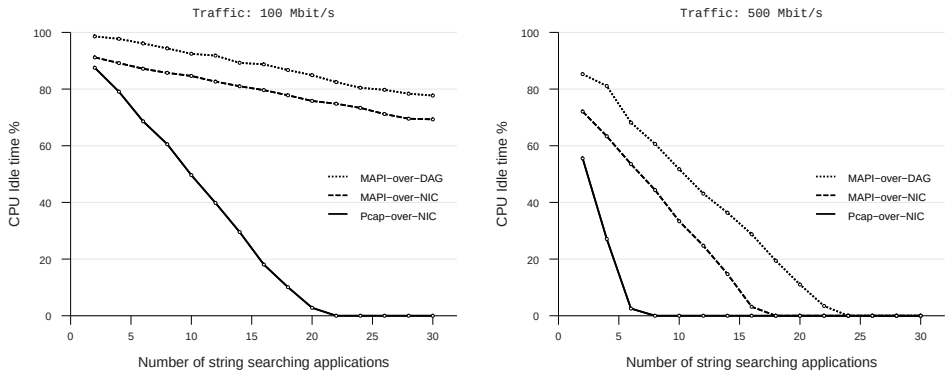

FIGURE 5.5: Performance of multiple string searching applications.

and how these metrics scale as the number of the participating monitoring sensors increases.

5.2.1 Experimental Environment

For the experimental evaluation of DiMAPI we used two different monitoring sensor deployments. The first system consists of 15 monitoring sensors distributed across our (FORTH) network. All nodes are interconnected through 100 Mbps Ethernet, for the sensor control interface. Each sensor is equipped with a second Ethernet interface for the actual passive network monitoring. The monitored traffic is generated using `iperf` [64] and `tcpreplay` [65]. The second monitoring sensor deployment consists of four monitoring sensors located at four different ASes across the Internet. One is located at FORTH, one at the University of Crete (UoC), one at the Venizelio Hospital at Heraklion (VH), and one at the University of Pennsylvania (UP). In this deployment, each sensor monitors live traffic passing through the monitored links of the corresponding organization. The operating system of the sensors is Linux (various distributions).

5.2.2 Network Overhead

As discussed in Chapter 4, whenever a monitoring application that utilizes remote sensors calls a DiMAPI library function, this results to a message exchange between the DiMAPI stub and the `commd` running on each sensor. This procedure poses questions about the overhead and the scalability of this approach. In this set of experiments, we set out to quantify the network overhead that DiMAPI incurs when used for building distributed monitoring applications.

For the experiments of this section, we implemented a simple test monitoring application that creates a network flow, configures it by applying several functions, and then periodically reports some result according to the applied functions. This application operates in a similar fashion to the examples presented in Section 3.2.

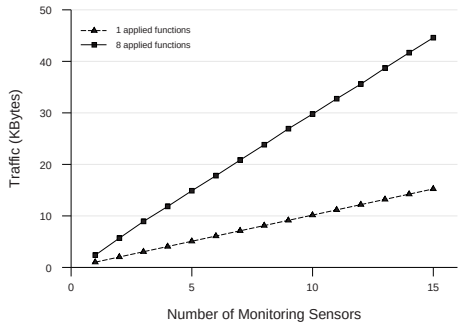

FIGURE 5.6: Total network traffic exchanged during the initialization phase, i.e., creation, configuration, and instantiation of a network flow, when applying 1 and 8 functions.

The measurements were performed in the 15-sensor FORTH network, while the test application was running on a separate host. Our goal is to measure the network overhead generated by DiMAPI, when using different monitoring granularity. The generated network traffic was measured using a second local DiMAPI application running on the same host with the test application. This local application reports the amount of DiMAPI control traffic by creating a network flow that captures all packets to and from the DiMAPI control port. Since it is a local monitoring application, it incurs no network traffic. We validated our results by capturing the control traffic using `tcpdump`.

In the first experiment, we measured the network overhead incurred during the initialization of a network flow, as a function of the number of remote monitoring sensors constituting the scope of the flow. The initialization overhead includes the traffic generated from both the DiMAPI stub and `commd` during the creation, configuration, and instantiation of a network flow. For example, in the example of Section 3.2, the initialization phase includes lines 1–14, and comprises five DiMAPI function calls.

Figure 5.6 shows the amount of traffic generated during the initialization phase for two variations of the test application. In the first variation, corresponding to the dashed line, the network flow is configured by applying only one function, which results to a total of three DiMAPI library function calls for the initialization phase. In the second variation, the network flow is configured by applying 8 functions, a rather extreme case, resulting to a total of 11 DiMAPI library function calls. The incurred traffic grows linearly with the number of monitoring sensors, and, for 15 sensors, reaches about 15 KBytes for the first variation and 45 KBytes for the second. In both cases, the network overhead remains low, and can be easily amortized during the lifetime of the application.

In the next experiment we measured the rate of the network traffic generated during the lifetime of the application due to the periodic results retrieval. After the initialization phase, the test application constantly reads the new value of the result

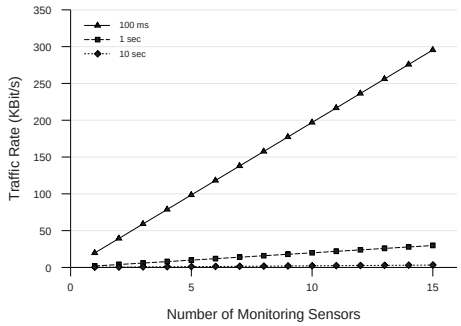

FIGURE 5.7: Network overhead incurred with a DiMAPI monitoring application that uses function BYTE_COUNTER, with polling periods 0.1, 1, and 10 seconds.

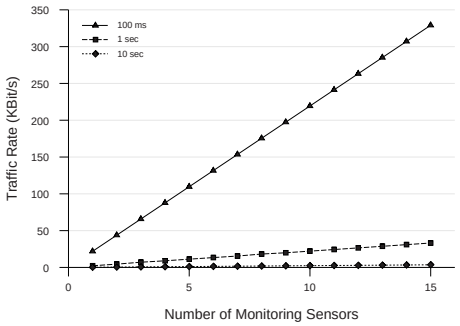

FIGURE 5.8: Network overhead incurred with a DiMAPI monitoring application that uses function HASHSAMP, with polling periods 0.1, 1, and 10 seconds.

by periodically calling `mapi_read_results()` at a predefined time interval. The measured traffic includes both the control messages of DiMAPI and the data transferred, across all monitoring sensors. We modified the test application to read the number of bytes of a network flow in three different periodic intervals, and plotted the mean rate of the generated traffic for one hour in Figure 5.7.

In the case that the application reads the result every 0.1 sec intervals, which is orders of magnitude lower than the minimum polling cycle allowed by most implementations of the Simple Network Management Protocol (SNMP), the generated traffic reaches 295 Kbit/s, when using a network flow with a scope of 15 sensors. However, for periodic intervals of one second or more, the generated traffic is negligible.

The result of the BYTE_COUNTER function is an unsigned 8-byte integer. In order to see the effects of larger result structures, we repeated the experiment by reading the results of the HASHSAMP function. HASHSAMP is used to perform hash-based sampling on the packets of a network flow, and its results format is a 36-byte structure. The traffic rate when reading the results of HASHSAMP is shown in Figure 5.8. We see that there is only a slight increase in the traffic rate due to the larger size of the produced results.

In all of our experiments the CPU utilization at the end-host was negligible, constantly lower than 1%.

5.2.3 Response Latency

In this set of experiments we set out to explore the delay between the call of a DiMAPI function and the return from the function. Since the call of a DiMAPI function results to a message exchange with each of the remote sensors within the flow's scope, the return from the function is highly dependent on the round trip

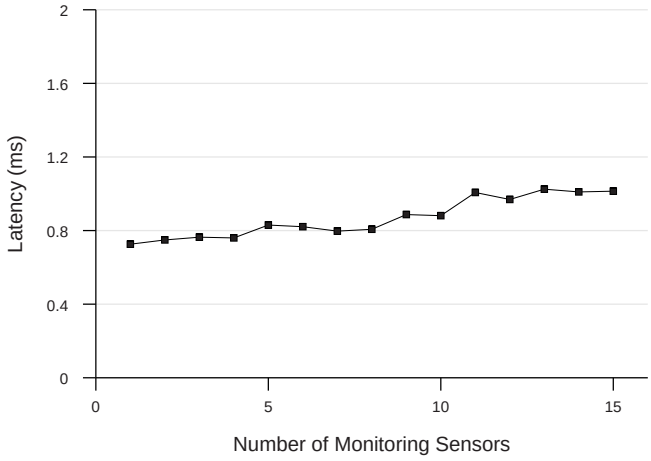

FIGURE 5.9: Completion time for `mapi_read_results()`. This includes the processing within the DiMAPI stub, the processing within each of the remote monitoring sensors, and the network round-trip time.

time (RTT) of the network path between the host on which the application runs and the remote monitoring sensors. Ideally, the latency introduced by DiMAPI should be negligible, and thus the overall latency should be close to the maximum RTT to the sensors within the flow's scope.

We measured the time it takes for `mapi_read_results()` to retrieve a result using the same test application we used for the experiments of Section 5.2.2 in the FORTH network. The time was measured by generating two timestamps using the `gettimeofday()` function from within the monitoring application, right before and after the call to `mapi_read_results()`. In this way, the measured time includes both the processing time of the DiMAPI stub and that of the remote sensor, as well as the network latency.

Figure 5.9 shows the completion time of the `mapi_read_results()` call as a function of the number of monitoring sensors constituting the network flow scope. As the number of sensors increases, there is a slight increase in the delay for retrieving the result. Since all the sensors are located within the FORTH LAN, the network latency for each monitoring sensor is almost constant and remains very low. Thus, the delay for retrieving the result from 15 sensors also remains very low, below 1 ms.

In order to explore how the network latency affects the delay of DiMAPI calls under more realistic conditions, we repeated the experiment using the second sensor deployment. As described in Section 5.2.1, this network comprises monitoring hosts located in four different ASes across the Internet, thus the RTT between the end host where the application runs and each of the monitoring sensors varies considerably.

Network flow scope	**`mapi_read_results()` delay (ms)**	**Network RTT (ms)**
VH	170.58	160.69
UoC	3.26	3.24
FORTH	0.68	0.67
UP	283.65	279.22
VH, UoC, FORTH, UP	285.496	-

TABLE 5.2: Comparison between the completion time of a DiMAPI call and the network round trip time.

We report our findings in Table 5.2. The third column shows the actual RTTs for each sensor, as measured from the end host using `ping`. We measured the delay of `mapi_read_results()` for reading a result from each monitoring sensor. The results of Table 5.2 suggest that for each sensor, the delay is slightly higher, but comparable, to the corresponding RTT. Furthermore, when using a network flow with a scope that includes all the monitoring sensors, the delay is roughly equal to the delay of the slowest sensor.

Chapter 6

Applications

The network flow abstraction of MAPI allows users to rapidly develop simple monitoring applications within a few lines of code. Its expressive power can also be used for building advanced applications that need to perform complex monitoring tasks. In the following sections we present two relatively simple monitoring applications, which demonstrate the ease of use of MAPI and DiMAPI, as well as a more complex distributed intrusion detection application.

6.1 Usage Examples

The following sections present two example programs that demonstrate the ease of programming with MAPI for building applications that perform complex monitoring operations. Both applications may operate using either one or many monitoring sensors, depending on the scope of the relevant network flows.

6.1.1 Link Utilization

The following listing presents an application that periodically reports the utilization of a network link. It uses two network flows to separate the incoming from the outgoing traffic.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <signal.h>
5  #include "mapi.h"
6
7  static void terminate();
8  int in_fd, out_fd;
9
10 int main() {
11
12     int in_fid, out_fid;
13     unsigned long long *in_cnt, *out_cnt;
14     unsigned long long in_prev=0, out_prev=0;
15
```

```
16      signal(SIGINT, terminate);
17      signal(SIGQUIT, terminate);
18      signal(SIGTERM, terminate);
19
20      /* create two flows, one for each traffic direction */
21      in_fd = mapi_create_flow("eth0");
22      out_fd = mapi_create_flow("eth0");
23      if ((in_fd < 0) || (out_fd < 0)) {
24          printf("Could not create flow\n");
25          exit(EXIT_FAILURE);
26      }
27
28      /* separate incoming from outgoing packets */
29      mapi_apply_function(in_fd, "BPF_FILTER",
30                          "dst host 139.91.145.84");
31      mapi_apply_function(out_fd, "BPF_FILTER",
32                          "src host 139.91.145.84");
33
34      /* count the bytes of each flow */
35      in_fid = mapi_apply_function(in_fd, "BYTE_COUNTER");
36      out_fid = mapi_apply_function(out_fd, "BYTE_COUNTER");
37
38      /* connect to the flows */
39      if(mapi_connect(in_fd) < 0) {
40          printf("Could not connect to flow %d\n", in_fd);
41          exit(EXIT_FAILURE);
42      }
43      if(mapi_connect(out_fd) < 0) {
44          printf("Could not connect to flow %d\n", out_fd);
45          exit(EXIT_FAILURE);
46      }
47
48      while(1) {      /* forever, report the load */
49
50          sleep(1);
51
52          in_cnt = (unsigned long long *)mapi_read_results(in_fd, in_fid);
53          out_cnt = (unsigned long long *)mapi_read_results(out_fd, out_fid);
54
55          printf("incoming: %.2f Mbit/s (%llu bytes)\n",
56                  (*in_cnt-in_prev)*8/1000000.0, (*in_cnt-in_prev));
57          printf("outgoing: %.2f Mbit/s (%llu bytes)\n\n",
58                  (*out_cnt-out_prev)*8/1000000.0, (*out_cnt-out_prev));
59
60          in_prev = *in_cnt;
61          out_prev = *out_cnt;
62      }
63      return 0;
64  }
65
66  void terminate() {
67      mapi_close_flow(in_fd);
68      mapi_close_flow(out_fd);
69      exit(EXIT_SUCCESS);
70  }
```

We begin by creating two network flows with flow descriptors `in_fd` and `out_fd` (lines 21 and 22) for the incoming and outgoing traffic, respectively, and we then apply the filters that will differentiate the traffic captured by each

flow (lines 29 and 31). In our case, we monitor the link that connects the host 139.91.145.84 to the Internet. All incoming packets will then have 139.91.145.84 as destination address, while all outgoing packets will have this IP as source address. In case that we would monitor a link that connects a whole subnet to the Internet, the host in the filtering conditions should be replaced by that subnet. For instance, for the subnet 139.91/16, the filter for the incoming traffic would be`dst net 139.91.0.0`. Since we are interested in counting the amount of traffic passing through the monitored link, we apply the `BYTE_COUNTER` function to both flows (lines 35 and 36), and save the relevant function descriptors in `in_fid` and `out_fid` for future reference.

Finally, the flow of control enters the main program loop, which periodically prints the incoming and outgoing traffic in Mbit/s, and the number of bytes seen in each one second interval (lines 56–59). In each iteration, the current value of each `BYTE_COUNTER` function result is retrieved by dereferencing `in_cnt` and `out_cnt`.

For ensuring a graceful termination of the program, we have initially registered the signals `SIGINT`, `SIGTERM`, and `SIGQUIT` with the function `terminate()` (lines 16–18), which closes the two flows and terminates the process.

6.1.2 Covert Peer-to-Peer Traffic Identification

This example demonstrates how DiMAPI can be used for the identification of covert traffic from Gnutella file sharing clients—a rather complicated task that requires deep packet inspection. Several Gnutella clients offer the capability to operate using HTTP traffic through port 80, thus masquerading as normal web traffic, in order to bypass strict firewall configurations that aim to block P2P traffic. The following code illustrates how DiMAPI can be used for writing a simple monitoring application that identifies file sharing clients joining the Gnutella network using covert web traffic.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <signal.h>
5  #include "mapi.h"
6
7  static void terminate();
8  void print_IP_pkt(struct mapipkt *pkt);
9
10 int fd;
11
12 int main() {
13
14     int fid;
15     struct mapipkt *pkt;
16
17     signal(SIGINT, terminate);
18     signal(SIGQUIT, terminate);
19     signal(SIGTERM, terminate);
20
```

```
21      /* create a flow using the eth0 interface */
22      fd = mapi_create_flow("sensor.uninett.no:/dev/dag0,"
23                            "mon1.ics.forth.gr:eth0");
24      if (fd < 0) {
25          printf("Could not create flow\n");
26          exit(EXIT_FAILURE);
27      }
28
29      /* keep only web packets */
30      mapi_apply_function(fd, "BPF_FILTER", "tcp and port 80");
31
32      /* indicating Gnutella traffic */
33      mapi_apply_function(fd, "STR_SEARCH", "GNUTELLA CONNECT");
34
35      /* must use TO_BUFFER in order to read full packet records */
36      fid = mapi_apply_function(fd, "TO_BUFFER");
37
38      /* connect to the flow */
39      if(mapi_connect(fd) < 0) {
40          printf("Could not connect to flow %d\n", fd);
41          exit(EXIT_FAILURE);
42      }
43
44      while(1) {      /* forever, wait for matching packets */
45
46          pkt = mapi_get_next_pkt(fd, fid);
47          printf("\nGnutella packet!\n");
48          print_IP_pkt(pkt);
49      }
50
51      return 0;
52  }
53
54  void terminate() {
55      mapi_close_flow(fd);
56      exit(EXIT_SUCCESS);
57  }
```

In the same fashion as with the previous example, the program starts by creating a network flow for capturing the Gnutella packets. However, in this example, the scope of the flow denotes that it will receive packets from two remote monitoring sensors (line 22). We then configure the network flow to match Gnutella control packets. We first apply the function `BPF_FILTER` to keep only the packets seemingly destined to, or coming from, a web server (line 30). Once a file sharing client that wants to connect to the Gnutella network obtains the address of another servant on the network, it sends a connection request containing the string "`GNUTELLA CONNECT`." Thus, we use the function `STR_SEARCH` to further restrict the packets of the flow to those containing this characteristic string (line 33).

We are interested in printing detailed information about each captured packet, so we need to receive the full records of the matching packets to the address space of the application. This is accomplished by applying the function `TO_BUFFER` (line 36), which instructs `mapid` to store the captured packets that match the conditions of the flow. The application can then retrieve the stored records using `mapi_get_next_pkt()`.

Finally, we activate the flow (line 39). At this point, each monitoring sensor has started inspecting the monitored traffic for covert Gnutella traffic. In the main execution loop, the application blocks into `mapi_get_next_pkt()` (line 46) until a matching packet is available. When such a packet is captured, the application prints its source and destination MAC and IP addresses by calling `print_IP_pkt()`. Since the application has access to the full packet record, `print_IP_pkt()` can be altered as needed to print any other part of the packet, even the entire packet payload.

6.2 Distributed Intrusion Detection

In this section we describe our experience with building a distributed Network Intrusion Detection System (NIDS) using DiMAPI. NIDSes are an important part of any modern network security management architecture, providing an additional layer of protection against cyber-attacks. A NIDS monitors the network traffic, trying to detect attacks or suspicious activity by matching packet data against well-defined patterns. Such patterns, also known as *signatures*, identify attacks by matching fields in the header and the payload of packets. For example, a packet directed to port `80` containing the string `/bin/perl.exe` in its payload is probably an indication of a malicious user attacking a web server. This attack can be detected by a signature which checks the destination port number, and defines a string search for `/bin/perl.exe` in the packet payload.

6.2.1 Benefits of Distributed Intrusion Detection

As a response to the growing number of cyberattacks, system administrators are increasingly deploying intrusion detection sensors at diverse locations across their administrative domains, seeking a more complete coverage against potential threats. As described earlier, intrusion detection systems are based on a predefined set of signatures, with each signature crafted to match a specific pattern of a known, malicious event. Upon the detection of suspicious activity, the IDS issues an alert to warn the system operator about the emerging threat. Since the signature that was triggered is known and the malicious packet(s) that triggered this particular signature have been logged, the operator has accurate information regarding the source and the type of the attack. The composition and selection of the signatures, which constitute the heart of an IDS, is a difficult task which must be accomplished carefully. Complex signatures can precisely describe advanced attacks, but can easily miss even their slight variations. Simplistic signatures tend to trigger too often and result to declare benign traffic as malicious. Besides the inherent inability of NIDSes to detect novel attacks, the major drawbacks of current NIDSes are the high rate and verbosity of the alerts, and the increased number of false positives, i.e. events that look like attacks, but in reality are completely legitimate traffic.

The increasing use of intrusion detection systems results to vast amounts of logs and alerts that are difficult, if not impossible, to manage. Indeed, it is hard to separate the false alarms from the real attacks among the hundreds of reported alerts, and to bring out the truly critical threats. In addition, it seems very difficult to identify sophisticated new types of attacks which involve many different hosts, such as Distributed Denial of Service (DDos) attacks, polymorphic worms, and distributed port scans. A security system based only on the sensors deployed at the network of a single autonomous system, has still a restricted point of view, which limits its ability to detect such coordinated attacks.

The shortcomings of existing network security applications give rise for more extended and scalable solutions through the use of *distributed* intrusion detection systems. This approach involves the *cooperation* of many intrusion detection sensors, distributed over a large network or several collaborating autonomous systems. The analysis and correlation of the data gathered at each sensor gives a broader perspective in which related incidents become apparent. The major benefits of distributed security applications include:

Reduced false alarms. A distributed infrastructure can cover a larger and more diverse monitoring space than traditional single-node security applications. Events seen by each sensor can be correlated with the information collected from the rest of the sensors in order to increase the confidence of the reported incidents [66]. Studies have shown that cooperative detection is more effective in suppressing "false alarms" that may be caused, for example, by mechanisms with attack-like characteristics like popular downloads, flash crowds and peer-to-peer applications [3]. For example, a sudden increase of identical packets at different sensors may imply the outbreak of a new worm. The same effect seen by a single node could be due to legitimate requests for a popular page that just went on-line. Therefore, distributed intrusion detection systems may have considerably reduced false alarm rate.

Enhanced detection capability. With the deployment of numerous monitoring sensors, there is an increased possibility that distributed and coordinated attacks will cross several of the cooperating sensors which may be able to identify them. An individual sensor may not be able to observe such kinds of attacks, since it will get only a small portion of the malicious traffic, while a cooperative security system exploits the complementary coverage from several monitoring nodes. A recent study [71] suggests that a network of 40 distributed sensors is sufficient for detecting "top-offenders." As the number of sensors increases, the system can detect attacks at an earlier stage and can react more effectively. The correlation of data from multiple sources enhances the detection accuracy of current heuristics-based detection systems, enabling them to perform more accurate detection of sophisticated new types of attacks which intrusion detection systems cannot cope with, such as polymorphic worms or slow port scanners.

Reduced number of alerts. Current intrusion detection systems tend to be verbose and produce vast amounts of alerts which are difficult to manage and interpret. In a distributed intrusion detection system, alerts from various sensors are aggregated and may be prioritized according to the number of different sensors that have reported a particular alert. Hence, they provide condensed alerts and logs which enables the operator to make informed decisions about the emerging threats.

Resilience and Scalability. A distributed infrastructure makes it harder for attackers to evade detection by spotting and blacklisting specific monitoring hosts or portions of the address space that are being monitored for random attacks. Additionally, such a system comprises numerous hosts, and thus, withstands denial of service attacks targeted to harm its availability. At the same time, a distributed infrastructure is easy to expand and reduces the cost per-participant, considering the related economies of scale.

6.2.2 Implementation

Implementing a distributed NIDS is a rather complicated task. Several basic operations like packet classification, TCP/IP stream reconstruction, and pattern matching, must be crafted together to form a fully functional system. Each one of these operations alone requires deliberate decisions for its design and considerable programming effort for its implementation. Furthermore, the resulting system is usually targeted to a specific hardware platform. For instance, the majority of current NIDSes are built on top of `libpcap` [40] packet capture library using commodity network interfaces set in promiscuous mode. As a result, given that `libpcap` provides only basic packet delivery and filtering capabilities, the programmer has to provide considerable amount of code to implement the large and diverse space of operations and algorithms required by a NIDS.

In contrast, DiMAPI inherently supports the majority of the above operations in the form of predefined functions which can be applied to network flows, and thus, can be effectively used for the development of a simple NIDS. Consequently, a great burden is released from the programmer who has now a considerably easier task. Furthermore, a NIDS based on DiMAPI is not restricted to a specific hardware platform. DiMAPI operates on top of a diverse range of monitoring hardware, including sophisticated lower level components [14], and thus, can further optimize overall system performance, considering that certain functions can be pushed to hardware.

We have developed a distributed NIDS using DiMAPI. Based on the observation that a signature which describes a known intrusion threat can be represented by a corresponding network flow, the overall implementation is straightforward. As an example, consider the following `snort` [53] signatures. The first detects malicious activity from a host infected with a backdoor, while the second detects packets of the Witty worm [37, 56].

```
alert tcp any 146 -> any 1000:1300
    (msg:"BACKDOOR Infector 1.6";
    content: "|57 48 41 54 49 53 49 54|";
    depth: 100; flags:A+; sid:120;)

alert udp any 4000 -> any any
    (msg:"ISS PAM/Witty Worm Shellcode";
    content: "|65 74 51 68 73 6f 63 6b 54 53|";
    depth:246; classtype:misc-attack;
    reference:url, www.lurhq.com/witty.html; sid:1000078;)
```

The packets that match the above rules can be returned by two corresponding network flows, after the application of the appropriate DiMAPI functions:

```
fd_backdoor = mapi_create_flow(dev);
mapi_apply_function(fd_backdoor, "BPF_FILTER",
    tcp and (src port 146) and ((tcp[2:2]>=1000 "
    "and tcp[2:2]<=1300)) and tcp[13:1]&16>0);
mapi_apply_function(fd_backdoor, "STR_SEARCH",
    "|57 48 41 54 49 53 49 54|", 0, 100);

fd_witty = mapi_create_flow(dev);
mapi_apply_function(fd_witty, BPF_FILTER,
    udp and src port 4000);
mapi_apply_function(fd_witty, STR_SEARCH,
    |65 74 51 68 73 6f 63 6b 54 53|, 0, 246);
```

Our DiMAPI-based NIDS operates as follows: during program start-up, the files that contain the set of rules are parsed, and for each rule, a corresponding network flow is created. Rules are written in the same description language used by `snort`. The rest of the functionality is left to DiMAPI, which will optimize the functional components of all the defined rules and deliver any matching packets. Whenever an attack packet is received from any of the network flows, SIDS generates a corresponding alert and, depending on the user's choice, prints the attack packet on the console, dumps it on disk, or logs it on a database. The database export format of the alerts is also compatible with `snort`, which allows the use of a wide variety of visualization tools, such as `ACID` [17].

Our implementation takes no more than 3000 lines of code, while the core functionality of other popular NIDSes, such as `snort`, consists of roughly 30.000 lines of code[1]. For example, `libpcap` does not provide any string searching facility, and thus the programmer would have to provide a significant chunk of code for the implementation of the string searching algorithm. Instead of forcing the programmer to provide all this mundane code, MAPI already provides this frequently used functionality.

[1] Although the functionality of the two systems is not identical, it is clearly depicted a difference in code length of at least one order of magnitude.

Additionally, the functionality of the monitoring daemon at each sensor can be shared by multiple concurrently running applications. For example, along with the intrusion detection application, one can develop a firewall application in the same fashion (i.e., in a few lines of code), adding this way extra capabilities to the overall system. Again, instead of providing code for the whole firewall operations, the programmer can use DiMAPI to reduce the development effort, and to effectively share resources by pushing the core firewall functionality into the MAPI daemon.

We should note that DiMAPI is not designed to fully replace fully-blown NIDS platforms. Our experience with intrusion detection shows that DiMAPI provides reasonably good support and performance for basic intrusion detection functionality, especially in a distributed monitoring environment. This is useful for providing the defense capabilities needed to respond to coordinated large-scale attacks. A key advantage of our DiMAPI-based NIDS is that the it can rely on more than one vantage points for attack detection. The scope of the network flows that correspond to the signatures is user-defined, thus the system is able to observe and correlate malicious activity from multiple sources. This functionality is crucial for building distributed early-warning systems for large-scale attacks like Internet worms [72].

Chapter 7

Related Work

Although MAPI presents a novel approach to passive network monitoring, it shares some functionality with existing network monitoring systems. The most widely used library for packet capture is `libpcap` [40], which provides a flexible and portable API for user-level packet capture. The `libpcap` interface supports a filtering mechanism based on the BSD Packet Filter [39], which allows for selective packet capture based on header fields. The Linux Socket Filter [31] offers similar functionality with BPF for the Linux OS, while xPF [32] and FFPF [7] provide a richer programming environment for network monitoring at the packet filter level. FLAME [4] is an architecture that allows users to directly install custom modules on the monitoring system, similarly in principle to Management-by-Delegation models [28]. Windmill [38] is an extensible network probe environment which allows loading of "experiments" on the probe, for analyzing protocol performance.

CoralReef provides a set of tools and support functions for capturing and analyzing network traces [34]. `libcoral` provides an API for monitoring applications that is independent of the underlying monitoring hardware. Nprobe [42] is a monitoring tool for network protocol analysis. 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 is more expressive than existing tools. One of the main differences is that MAPI can be used to filter packets based on payload data. Furthermore, 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 5.1. By being expressive, MAPI enables the underlying monitoring system to choose the most appropriate implementation that matches the application's needs. DiMAPI leverages current passive network monitoring/capturing approaches that are tied to a single monitoring host, into a distributed environment. Indeed, DiMAPI is implemented on top of various monitoring architectures, in-

cluding `libpcap`-based interfaces and DAG cards [23], and provides a flexible interface on top of them for building distributed monitoring applications.

The `pcap` library has been widely used in several passive monitoring applications such as packet capturing [52, 62], network statistics monitoring [19], and intrusion detection systems [53]. `WinPcap` [2] and `rpcap` [35] extend `libpcap` with remote packet capture capabilities. Both allow captured packets at a remote host to be transferred at a local host for further processing. DiMAPI offers the same functionality but for *multiple* distributed monitoring sensors. Also, DiMAPI enables traffic processing at each remote monitoring sensor and sending back only the computed results. In this way, the considerable network overhead of above approaches due to the transfer of the captured packets is avoided.

Mmdump [67] is a specialized tool for tracing multimedia applications. Mmdump extends `tcpdump` by parsing messages from RTSP, H.323 multimedia session control protocols to set up and tear down packet filters as needed. It parses the control messages to extract the dynamically assigned port numbers. Then the packet filter is changed to capture the packets of that stream. Mmdump is a custom solution for tracking applications that use dynamic ports. MAPI generalizes this idea and can be used for a much broader spectrum of applications (including those targeted by Mmdump).

Except from packet capture oriented systems, there has been significant activity in the design of systems providing flow-based measurements. Cisco IOS NetFlow technology [12] collects and measures traffic data on a per-flow basis. In this context, a flow is usually defined as all packets that share a common protocol, source and destination IP addresses, and port numbers. In contrast to packet capture systems, NetFlow only extracts and maintains flow-level records, from which various traffic statistics can be derived. NeTraMet [9], much like NetFlow, can collect traffic data on a per-flow basis focusing only on flows that match a specific rule. FlowScan [48] 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 build a visualization application with the same capabilities but for a broader range of network and application characteristics.

A drawback of such tools is that they are usually accessible only by network administrators who have access rights to network equipment like routers. Open source probes like `nProbe` [18] offer NetFlow record generation by capturing packets using commodity hardware. Although DiMAPI 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, DiMAPI is able to provide sophisticated traffic statistics, e.g., for applications that use dynamically allocated ports [49].

There is also significant activity within the IETF for defining network traffic monitoring standards, from working groups such as RMON [68], PSAMP [21], and IPFIX [50]. The Simple Network Management Protocol (SNMP) [10] facilitates the retrieval of traffic statistics from network devices. Though SNMP is useful it is limited in capabilities, since we cannot perform fine-grained monitoring.

RMON offers significantly more capabilities than SNMP, however its complexity and overhead prohibit wide deployment [29].

As network traffic monitoring is becoming increasingly important for the operation of modern networks, several passive monitoring infrastructures have been proposed. Gigascope [16] is a stream database for storing captured network data in a central repository for further analysis using the GSQL query language. A similar approach is followed by the CoMo project [30], which allows users to query network data gathered from multiple administrative domains, and Sprint's passive monitoring system [26], which also collects data from different monitoring points into a central repository for analysis. Arlos et al. [5] propose a distributed passive measurement infrastructure that supports various monitoring equipment within the same administrative domain.

Chapter 8

Conclusion

We have presented a novel programming abstraction for distributed network traffic monitoring. MAPI strikes a balance between performance and flexibility in network traffic monitoring, by providing a flexible and expressive interface that allows users to specify complex monitoring needs, Indeed, current approaches to network monitoring provide either too much information, such as `libpcap`, or too little information, such as flow-level traffic summaries. On the contrary, MAPI provides a rich and expressive interface that enables users to precisely define their traffic monitoring needs, receive only the amount of information they are interested in, and therefore pay the overhead of only the information they receive. This is achieved by building an API around a generalized network flow abstraction, which 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.

Furthermore, the Distributed version of MAPI facilitates the programming and coordination of several geographically distributed monitoring sensors from within a single monitoring application. DiMAPI introduces the network flow scope, an attribute that enables the creation and manipulation of network flows over several local and remote passive monitoring sensors. The design of DiMAPI mainly focuses on minimizing performance overheads, while it provides extensive functionality for a broad range of distributed monitoring applications.

We have implemented a prototype of MAPI on top of a commodity Gigabit Ethernet network interface (Intel Pro 1000 MT), as well as on top of a DAG 4.2 GE special-purpose adapter 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 outperforms competitive network traffic monitoring approaches. As Section 5.1 shows, MAPI improves application performance compared to `libpcap`. This is because MAPI enables applications to receive only the traffic they are interested in, while `libpcap` forces applications *first* to receive all network traffic, and *then* select the packets they are interested in.

We have also evaluated the network-level performance of DiMAPI using a number of monitoring applications operating over large monitoring sensor sets, as well as highly distributed environments. Our results show that DiMAPI has low overhead, while the response latency in retrieving monitoring results is very close to the actual round trip time between the monitoring application and the monitoring sensors within the scope.

As networks continue to get faster and emerging Internet applications become more complex, we expect that the impact of MAPI on efficiency and ease of programming is expected to increase. Furthermore, as intelligent hardware becomes more prevalent, MAPI will allow programmers to precisely express their monitoring needs to the underlying system, which will optimize the monitoring process in the most effective way. 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 and programmable monitoring cards, is likely to improve performance further, considering that certain functions can be pushed to the hardware. Currently, our efforts are focused on the deployment of DiMAPI-enabled monitoring sensors across several autonomous systems, aiming to create a large-scale passive monitoring infrastructure. Finally, we are exploring the possibility of extending the functionality of the communication agent, in order to allow the monitoring sensors to communicate with each other, for supporting monitoring applications that require such functionality.

Bibliography

[1] MAPI Public Release. http://mapi.uninett.no/download/mapi_1.0b1.tgz.

[2] WinPcap Remote Capture. http://www.winpcap.org/docs/docs31beta4/html/group__remote.html.

[3] K. G. Anagnostakis, M. B. Greenwald, S. Ioannidis, A. D. Keromytis, and D. Li. A Cooperative Immunization System for an Untrusting Internet. In *Proceedings of the 11th IEEE Internation Conference on Networking (ICON)*, September/October 2003.

[4] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, M. 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.

[5] P. Arlos, M. Fiedler, and A. A. Nilsson. A distributed passive measurement infrastructure. In *Proceedings of the 6th International Passive and Active Network Measurement Workshop (PAM'05)*, pages 215–227, 2005.

[6] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust-Management System version 2. *IETF, Network Working Group, Informational RFC 2704*, Sept. 1999.

[7] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proceedings of OSDI'04*, 2004.

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

[9] N. Brownlee. Traffic flow measurement: Experiences with NeTraMet. RFC2123, http://www.rfc-editor.org/, March 1997.

[10] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP), May 1990. RFC1157. http://www.ietf.org/rfc/rfc1157.txt.

[11] CERT. CERT Advisory CA-2003-04 MS-SQL Server Worm. http://www.cert.org/advisories/CA-2003-04.html.

[12] Cisco Systems. Cisco IOS Netflow. http://www.cisco.com/warp/public/732/netflow/.

[13] CNN.com. Computer worm grounds flights, blocks ATMs, Jan. 2003. http://www.cnn.com/2003/TECH/internet/01/25/internet.attack/.

[14] J. Coppens, S. V. den Berghe, H. Bos, E. Markatos, F. D. Turck, A. Øslebø, and S. Ubik. SCAMPI: A Scalable and Programmable Architecture for Monitoring Gigabit Networks. In *Proceedings of the E2EMON Workshop*, 2003.

[15] J. Coppens, E. P. Markatos, J. Novotny, M. Polychronakis, V. Smotlacha, and S. Ubik. SCAMPI - a scaleable monitoring platform for the internet. In *Proceedings of the 2nd International Workshop on Inter-Domain Performance and Simulation (IPS)*, Mar. 2004.

[16] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 647–651, 2003.

[17] R. Danyliw. Analysis Console for Intrusion Databases (ACID). http://acidlab.sourceforge.net/.

[18] L. Deri. nProbe. http://www.ntop.org/nProbe.html.

[19] L. Deri. ntop. http://www.ntop.org/.

[20] L. Deri. nCap: Wire-speed packet capture and transmission. In *Proceedings of the IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services (E2EMON)*, 2005.

[21] N. Duffield. A framework for packet selection and reporting, 2005. Internet Draft. http://www.ietf.org/internet-drafts/draft-ietf-psamp-framework-10.txt.

[22] eEye Digital Security. .ida “Code Red” Worm. http://www.eeye.com/html/Research/Advisories/AL20010717.html.

[23] Endace measurement systems. *DAG 4.3GE dual-port gigabit ethernet network monitoring card*, 2002.

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

[25] R. Farrow. Reverse-engineering New Exploits. *CMP Network Magazine*, Mar. 2004.

[26] C. Fraleigh, C. Diot, B. Lyles, S. Moon, P. Owezarski, D. Papagiannaki, and F. Tobagi. Design and Deployment of a Passive Monitoring Infrastructure. In *Proceedings of the Passive and Active Measurement Workshop*, Apr. 2001.

[27] C. Fraleigh, S. Moon, C. Diot, B. Lyles, and F. ad Tobagi. Architecture of a passive monitoring system for ip networks. Technical Report TR00-ATL-101801, Sprint, 2000.

[28] G. Goldszmidt and Y. Yemini. Distributed management by delegation. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS)*, pages 333–340, 1995.

[29] M. Grossglauser and J. Rexford. Passive traffic measurement for IP operations. In K. Park and W. Willinger, editors, *The Internet as a Large-Scale Complex System*, pages 91–120. Oxford University Press, 2005.

[30] G. Iannaccone, C. Diot, D. McAuley, A. Moore, I. Pratt, and L. Rizzo. The CoMo White Paper, 2004. `http://como.intel-research.net/pubs/como.whitepaper.pdf`.

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

[32] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis. xPF: packet filtering for low-cost network monitoring. In *Proceedings of the IEEE Workshop on High-Performance Switching and Routing (HPSR)*, pages 121–126, May 2002.

[33] S. Kandula, D. Katabi, M. Jacob, and A. W. Berger. Botz-4-Sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[34] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, and K. Claffy. The architecture of CoralReef: an Internet traffic monitoring software suite. In *PAM2001 — A workshop on Passive and Active Measurements*, Apr. 2001.

[35] S. Krishnan. rpcap. `http://rpcap.sourceforge.net/`.

[36] 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.

[37] LURHQ Threat Intelligence Group. Witty Worm Analysis. `http://www.lurhq.com/witty.html`.

[38] G. R. Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 215–227, 1998.

[39] S. McCanne and V. 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.

[40] S. McCanne, C. Leres, and V. Jacobson. libpcap. Lawrence Berkeley Laboratory, Berkeley, CA. (software available from http://www.tcpdump.org/).

[41] J. Mirkovic and P. Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, 2004.

[42] A. Moore, J. Hall, E. Harris, C. Kreibich, and I. Pratt. Architecture of a network monitor. April 2003.

[43] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[44] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The Spread of the Sapphire/Slammer Worm. Technical report, CAIDA, ICSI, Silicon Defense, UC Berkeley EECS and UC San Diego CSE, 2003.

[45] D. Moore, C. Shannon, and J. Brown. Code-Red: a case study on the spread and victims of an internet worm. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, 2002.

[46] M. Overton. Bots and botnets—risks, issues and prevention. In *Proceedings of the 15th Virus Bulletin Conference*, Oct. 2005.

[47] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

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

[49] M. Polychronakis, K. G. Anagnostakis, E. P. Markatos, and A. Øslebø. Design of an Application Programming Interface for IP Network Monitoring. In *Proceedings of the 9th IFIP/IEEE Network Operations and Management Symposium (NOMS'04)*, pages 483–496, Apr. 2004.

[50] J. Quittek, T. Zseby, B. Claise, and S. Zander. Requirements for IP Flow Information Export, Oct. 2004. RFC3917. `http://www.ietf.org/rfc/rfc3917.txt`.

[51] E. Rescorla. Security holes... Who cares? In *Proceedings of the 12th USENIX Security Symposium*, pages 75–90, Aug. 2003.

[52] J. Ritter. ngrep – Network grep. http://ngrep.sourceforge.net/.

[53] M. 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/).

[54] Rusty Russell. Linux 2.4 Packet Filtering HOWTO, 2002.

[55] SecurityFocus. Slammer worm crashed Ohio nuke plant network, Aug. 2003. http://www.securityfocus.com/news/6767.

[56] C. Shannon and D. Moore. The Spread of the Witty Worm, 2004. http://www.caida.org/analysis/security/witty/.

[57] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The top speed of flash worms. In *Proceedings of the Second ACM workshop on Rapid malcode (WORM)*, pages 33–42, 2004.

[58] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002.

[59] The LOBSTER Project. Deliverable D1.1a: Anonymization Framework Definition, 2005. http://www.ist-lobster.org/deliverables/Deliverable_D1.1a.pdf.

[60] The Register. Phishers tapping botnets to automate attacks, Nov. 2004. http://www.theregister.com/2004/11/26/anti-phishing_report/.

[61] The SCAMPI Consortium. A Tutorial Introduction to MAPI, Apr. 2005. http://mapi.uninett.no/doc/mapitutor.pdf.

[62] The Tcpdump Group. tcpdump. http://www.tcpdump.org/.

[63] R. Thomas. Monitoring dos attacks with the vip console and netflow v1.0.

[64] A. Tirumala, J. Ferguson, J. Dugan, F. Qin, and K. Gibbs. Iperf. http://dast.nlanr.net/Projects/Iperf/.

[65] A. Turner. tcpreplay. http://tcpreplay.sourceforge.net/.

[66] A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*, pages 54–68. Springer-Verlag, 2001.

[67] 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.

[68] S. Waldbusser. Remote network monitoring management information base. RFC2819. `http://www.ietf.org/rfc/rfc2819.txt`.

[69] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous payload-based worm detection and signature generation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[70] J. Wu, S. Vangala, L. Gao, and K. Kwiat. An effective architecture and algorithm for detecting worms with various scan techniques. In *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS)*, 2004.

[71] V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the DOMINO overlay system. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2004.

[72] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and early warning for internet worms. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS)*, pages 190–199, 2003.

[73] C. C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 138–147, 2002.