University of Crete
Computer Science Department

Improving the Performance of Passive Network Monitoring Applications

Antonis Papadogiannakis

Master's Thesis

October 2007
Heraklion, Greece

University of Crete
Computer Science Department

Improving the Performance of Passive Network Monitoring Applications

Thesis submitted by

Antonis Papadogiannakis

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

THESIS APPROVAL

Author: _____
Antonis Papadogiannakis

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

_____
Mema Roussopoulos
Assistant Professor

_____
Vasilios A. Siris
Assistant Professor

Departmental approval: _____
Panos Trahanias
Professor, Chairman of Graduate Studies

Heraklio, October 2007

Abstract

Passive network monitoring is the basis for a multitude of systems that support the robust, efficient, and secure operation of modern computer networks. Traditional passive network monitoring approaches focus either on relatively simple network traffic measurement and analysis applications, or just for gathering packet traces that are analyzed off-line. However, these approaches are not adequate to support emerging monitoring applications such as intrusion detection systems, detection of Internet worm outbreaks and accurate traffic characterization. In addition, most of these applications would benefit from monitoring data gathered at multiple vantage points across the Internet. At the same time, the speed of modern network links increases, Internet traffic gets more complex, and applications more CPU and memory demanding due to more complex analysis operations. Thus, there is a growing demand for more efficient passive monitoring since the performance of such applications becomes a critical issue.

In this thesis we present the design, implementation and performance evaluation of DiMAPI, a flexible and expressive application programming interface for distributed passive network monitoring. A broad range of monitoring applications can benefit from DiMAPI to efficiently perform advanced monitoring tasks over a potentially large number of passive monitoring sensors.

Also, we present a novel approach for improving the performance of a large class of CPU and memory intensive passive network monitoring applications. Our approach, called *locality buffering*, reorders the captured packet stream, before it is delivered to the application, in a way that results to improved code and data locality, and consequently to an overall increase in the packet processing throughput and to a decrease in the packet loss rate. We have implemented locality buffering within the widely used `libpcap` packet capturing library, which allows existing monitoring applications to transparently benefit from the reordered packet stream without the need to change application code. Our experimental evaluation shows that locality buffering improves significantly the performance of popular applications.

Supervisor: Professor Evangelos Markatos

Περίληψη

Η παθητική εποπτεία της κίνησης ενός δικτύου υπολογιστών αποτελεί έναν σημαντικό παράγοντα για την εξασφάλιση της αποδοτικής και ασφαλής του λειτουργίας. Οι παραδοσιακές προσεγγίσεις εστιάζουν είτε σε απλές μετρήσεις και συλλογή στατιστικών, είτε στην πλήρη καταγραφή της κίνησης του δικτύου. Αυτές οι προσεγγίσεις όμως δεν είναι επαρκείς για να υποστηρίξουν τις νέες ανάγκες που έχουν προκύψει, όπως συστήματα για ανίχνευση επιθέσεων, ανίχνευση της ραγδαίας εξάπλωσης worms στο Διαδίκτυο και ακριβής ταξινόμηση της κίνησης του διαδικτύου ανάλογα με τις εφαρμογές που την παράγουν. Επιπλέον, η δυνατότητα ταυτόχρονης εποπτείας πολλών διαφορετικών δικτύων στο Διαδίκτυο θα οφελούσε αρκετά αυτές τις εφαρμογές. Την ίδια στιγμή, οι ταχύτητες των σύγχρονων δικτύων αυξάνουν, η κίνηση στο Διαδίκτυο γίνεται ολοένα και πιο περίπλοκη και οι εφαρμογές για παθητική εποπτεία δικτύων όλο και πιο απαιτικές σε υπολογιστική ισχύ. Για όλους τους παραπάνω λόγους υπάρχει αυξανόμενη ανάγκη για πιο αποδοτικές εφαρμογές εποπτείας δικτύων.

Σε αυτήν την εργασία παρουσιάζουμε την σχεδίαση, υλοποίηση και αξιολόγηση μιας προγραμματιστικής βιβλιοθήκης, που ονομάζεται DiMAPI, η οποία παρέχει δυνατότητες για την ανάπτυξη εφαρμογών για κατανεμημένη παθητική εποπτεία δικτύων. Η βιβλιοθήκη αυτή είναι ευέλικτη και εκφραστική, οπότε προσφέρει την δυνατότητα για ανάπτυξη αρκετών εφαρμογών χρησιμοποιώντας αποδοτικά ένα μεγάλο αριθμό εποπτευόμενων δικτύων.

Επίσης, παρουσιάζουμε μια καινοτόμα προσέγγιση για την βελτίωση της απόδοσης ενός μεγάλου εύρους από ήδη υπάρχουσες εφαρμογές που απαιτούν σημαντική υπολογιστική ισχύ. Αυτή η προσέγγιση, που ονομάζεται locality buffering, αναδιατάσσει την σειρά των πακέτων δικτύου, πριν τα δώσει στην εφαρμογή που θέλει να τα επεξεργαστεί, ομαδοποιώντας τα ¨παρόμοια' πακέτα έτσι ώστε να βελτιώνεται η πρόσβαση στην μνήμη του συστήματος και να επιταχύνεται η επεξεργασία των πακέτων απο την εφαρμογή λόγω μειωμένων cache misses. Υλοποιήσαμε αυτήν την προσέγγιση σε μια ευρέως διαδεδομένη βιβλιοθήκη για καταγραφή πακέτων δικτύου με τέτοιο τρόπο ώστε όσες εφαρμογές την χρησιμοποιούν να μπορούν να εποφεληθούν απο την τεχνική μας χωρίς να χρειαστεί να κάνουν καμμία αλλαγή στον κώδικά τους. Η πειραματική αξιολόγηση αυτής της τεχνικής, χρησιμοποιώντας τρεις δημοφιλής εφαρμογές, δείχνει οτι μπορεί να επιτύχει σημαντικές βελτιώσεις στην απόδοση τους.

Επόπτης: Καθηγητής Ευάγγελος Μαρκατος

Acknowledgments

I am deeply grateful to my supervisor, Professor Evangelos Markatos, for his valuable advice and guidance during all my studies. I am also grateful to Michalis Polychronakis for his constant support and excellent cooperation. It is truly a valuable experience to work with these people.

Many thanks to Alexandros Kapravelos, who implemented the packet loss estimation application, and Andreas Makridakis for his help and work in the development of DiMAPI. I would also like to thank Demetres Antoniades and George Vasiliades who contributed in several parts of this work. I greatly appreciate their help.

My best thanks to my friends and collegues Manos Athanatos, Christos Papachristos, Elias Athanasopoulos, Spiros Antonatos, Demetres Koukis and to all the past and current members of the Distributed Computing Systems Laboratory in ICS/FORTH for their support, usefull comments and feedback.

I would also like to thank my friends Nikos Dimaresis, Panos Tourlakis, Dimitris Zeginis, Manolis Kritsotakis, Maria Kalaitzaki and many others that I do not mention by name, for their support and for sharing with me these years of my life.

Finally, I would like to thank my parents, Charidimos and Aikaterinh, for their support, patience and encouragement during all these years.

Contents

List of Figures

List of Tables

1 Introduction

1.1 The Need for Effective Passive Network Monitoring

Over the last few years, we noticed a rapid evolution and growth of the Internet. Since the widespread development of DSL broadband home connections, metropolitan wireless networks and mobile devices with Internet connection, the number of users, hosts, domains, and enterprise networks that are 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 media streaming, Grid computing and very popular peer-to-peer systems for file sharing, demand for increased bandwidth. Moreover, the number of attacks against Internet connected systems continues to grow at alarming rates.

As networks grow larger and more complicated, effective network monitoring and measurement is becoming an essential function for understanding, managing and improving the performance and security of computer networks. Network traffic monitoring is getting increasingly important for a large set of Internet users and service providers, such as ISPs, NRNs, computer and telecommunication scientists, security administrators, and managers of high-performance computing infrastructures.

Passive traffic monitoring and capturing has been regarded as the main solution for advanced network monitoring and security systems that require fine-grained performance measurements, such as *deep packet inspection* [26]. For instance, calculating the distribution of traffic among different applications has become a difficult task. Several recent applications use dynamically allocated ports, and therefore, cannot be identified based on a well known port number. Instead, protocol parsing and several other heuristics are commonly used, like searching for an

application-specific string in the packets payload [9]. Also, recent intrusion detection systems, such as `snort` [45] and `bro` [41], need to be able to inspect and process network packets payload, in order to detect computer viruses and worms at times of emergency, based on attack "signatures" and using advanced pattern matching algorithms.

However, traditional passive network monitoring approaches are not adequate for fine-grained performance measurements nor for security applications. Traditional approaches to passive network monitoring focus either on collecting flow-level statistics [12], which makes them unsuitable for applications that perform fine-grained operations like deep packet inspection, or in full packet capture [24], which significantly increases their operational overhead. Such limitations, i.e., too little information provided by flow-level traffic summaries versus 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.

While passive monitoring has been traditionally used for relatively simple network traffic measurement and analysis applications, or just for gathering packet traces that are analyzed off-line, in recent years it has become vital for a wide class of more CPU and memory intensive applications, such as network intrusion detection systems (NIDS) [45], accurate traffic categorization [9], and NetFlow export probes [1] which need to inspect both the headers and the whole payloads of the captured packets, a process widely known as *deep packet inspection*. The complex analysis operations of such demanding applications are translated into an increased number of CPU cycles spent on each captured packet, which reduces the overall processing throughput that the application can sustain without dropping incoming packets. At the same time, as the speed of modern network links increases, there is a growing demand for more efficient packet processing using commodity hardware that can keep up with higher traffic loads.

Moreover, traditional passive network monitoring applications are most commonly based on data gathered at a single observation point. Such applications run locally on the monitoring sensor, which gathers the required information and processes the captured data. Several emerging applications would benefit from monitoring data gathered at multiple observation points 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. Recent research efforts [50,52,54] have demonstrated that a large-scale monitoring infrastructure of distributed cooperative monitors can be used for building Internet worm detection systems. Distributed Denial-of-Service attack detection applications would also benefit from multiple vantage points across the Internet. Also, wide-area application (e.g. peer-to-peer systems) debugging can be facilitated by a distributed monitoring infrastructure. Finally, user mobility necessitates distributed

monitoring due to nomadic users who change locations frequently across different networks.

So, it is clear that distributed network monitoring is becoming necessary for understanding the performance of modern networks and for protecting them against security breaches. The wide dissemination of a cooperative passive monitoring infrastructure across many geographically distributed and heterogeneous sensors necessitates a uniform access platform, which provides a common interface for applications to interact with the distributed monitoring sensors.

1.2 Contributions

In the above section we indicate the need for effective passive network monitoring. Thus, the motivation of this work is to develop new libraries, or extend existing ones, that will facilitate the development of passive network monitoring applications and improve their performance. Also, a complementary goal is to transparently improve the performance of existing monitoring applications without need to altering their code. For performance improvements, a passive monitoring library can build on top of specialized hardware (e.g. DAG cards [23] or network processors [28]) in a transparent way for the applications. However, we prefer a generic user-level technique, for easy deployment, that will significantly improve the packet processing performance of the monitoring application itself using commodity hardware.

Our key novel contributions in this thesis are the following:

- We present DiMAPI, a flexible and expressive programming framework for effective distributed passive network monitoring. DiMAPI enables users to clearly communicate their monitoring needs to remote passive monitoring platforms. Using existing solutions, like `rpcap` [32] or `WinPcap` [5], we would have to fetch all the packets from each remote monitoring sensor to the application's host in order to process them. On the other hand, using DiMAPI we push more functionality to the monitoring sensors side and only the necessary results are being transfered over the network, that is much more effective.

- Furthermore, DiMAPI exploits specialized hardware for improving performance without any change to the API, so that applications are able to run without modifying their code. Also, the monitoring infrastructure is efficiently shared among many users, providing better performance by grouping and optimizing their monitoring needs into a single monitoring daemon.

- We introduce a scalable and non-intrusive technique based on distributed passive network monitoring for estimating the real-time packet loss ratio between different measurement points.

- We present a novel technique, called *locality buffering*, that is able to significantly improve the performance of a wide class of CPU and memory intensive passive network monitoring applications, such as intrusion detection systems [45], accurate traffic classification applications [9] and NetFlow export probes [1]. The technique is based on adapting the packet stream by clustering packets with the same destination port, before they are delivered to the monitoring application, resulting to improved memory access locality and consequently to an overall performance improvement in the packet processing throughput. We implemented locality buffering within the widely used `libpcap` library, so existing applications can benefit transparently without any changes to their code, and we experimentally evaluated it using three popular passive monitoring tools. The results shows that, for instance, the Snort intrusion detection system exhibits a 40% increase in the packet processing throughput and a 60% improvement in packet loss rate.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 presents the design and implementation of DiMAPI, illustrates some simple examples of DiMAPI usage and discusses the main advantages that it offers. In Chapter 3 we describe in detail two real-world applications which take advantage of the use of DiMAPI. Chapter 4 outlines the overall approach of locality buffering and presents its detailed implementation within the `libpcap` packet capture library. Chapter 5 presents the experimental evaluation of DiMAPI network-level performance and the experimental evaluation of locality buffering using three popular passive monitoring tools. Finally, Chapter 6 summarizes related work, and Chapter 7 concludes the thesis.

2

DiMAPI: An API for 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 2.1 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 several remote monitoring sensors. DiMAPI [49] fulfils this requirement by facilitating the programming and coordination of a set of remote sensors from within a single monitoring application. DiMAPI enables users to efficiently configure and manage any set of remote or local passive monitoring sensors, acting as a middleware to homogeneously use a large distributed monitoring infrastructure.

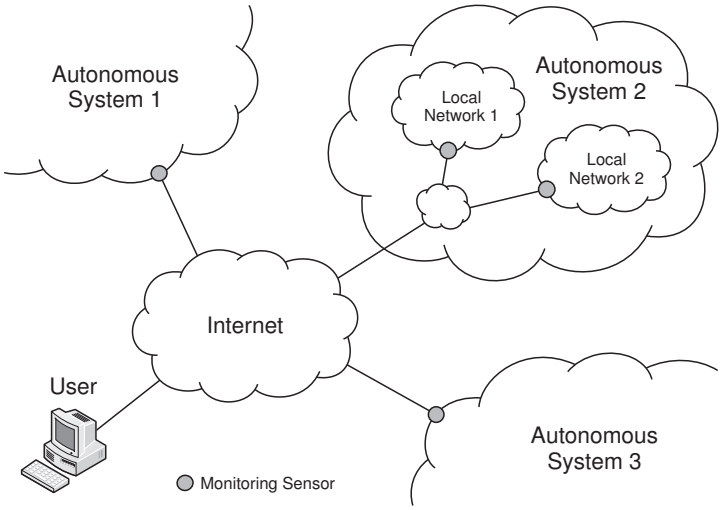

FIGURE 2.1: A high-level view of a distributed passive network monitoring infrastructure

2.1 Background: The Monitoring API

DiMAPI has been designed and realized by building on the Monitoring Application Programming Interface (MAPI) [42], an expressive and flexible API for passive network traffic monitoring over a single local monitoring sensor. MAPI builds on a generalized network flow abstraction and offers a standardized API, flexible and expressive enough to capture emerging application needs. Furthermore, MAPI applications are able to run with commodity network interfaces or specialized network monitoring hardware (e.g., DAG cards [23]) without the need to alter or re-compile their code.

In this section we introduce the main concepts of MAPI, briefly describe its most important operations and its implementation. A complete specification of MAPI is provided in the MAPI man pages [2].

2.1.1 Network Flow Abstraction

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 application specifications. A good API should also relieve the programmer from the complexities of the underlying monitoring platform, while making sure that any features of specialized hardware can be properly exploited.

Towards these targets, MAPI builds on a simple, yet powerful, abstraction: the *network flow*. 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

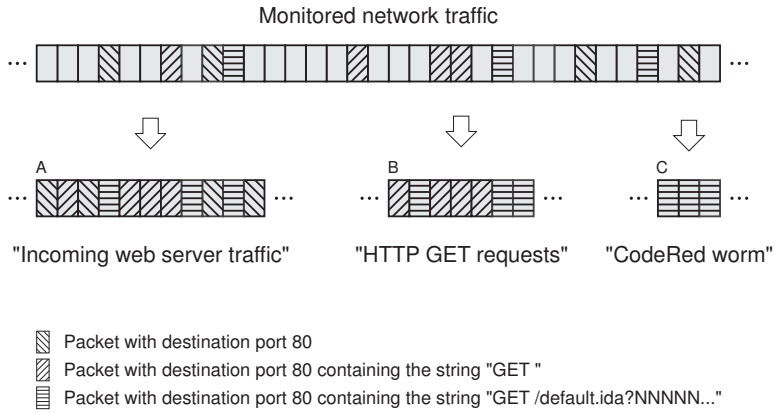

FIGURE 2.2: Network flow examples.

simple header-based filters to sophisticated protocol analysis and content inspection functions.

Figure 2.2 illustrates the concept of the network flow with some examples. On the top we see a portion of the monitored network traffic, and below three different network flows, each consisting of a subset of the monitored packets. 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 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], 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. Furthermore, MAPI gives the network flow a *first-class status*: flows are named entities that can be manipulated in similar ways to other programming abstractions, such as sockets, pipes, and files. In particular, users may create or destroy (close) flows, read, sample, or count the packets of a flow, apply functions to flows, and retrieve other statistics from a flow, etc. Using this generalized network flow abstraction, 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, such as NetFlow [12] and IPFIX [43], and related systems and proposals, do not have the means of providing the advanced functions required for this task. Going back to the example of Figure 2.2, 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`.

MAPI Function	Description
`mapi_create_flow`	Creates a new network flow
`mapi_apply_function`	Applies a function to all the packets of a flow
`mapi_connect`	Connects to a flow to start receiving results
`mapi_read_results`	Receives results computed by a function in the packets of a flow
`mapi_get_next_packet`	Reads the next packet of a flow
`mapi_loop`	Invokes a handler function for each of the packets of a flow
`mapi_close_flow`	Closes a flow

TABLE 2.1: Overview of the basic MAPI calls

2.1.2 Basic MAPI Operations

This section gives an overview of the basic MAPI function calls, summarized in Table 2.1. For a complete list of the available MAPI functions, along with their detailed descriptions, please refer to [2].

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, or -1 on error. By default, a newly created flow consists of all network packets that go through the monitoring interface `dev`. The packets of this flow can be further reduced to those which satisfy an appropriate filter or other condition, as will be described later.

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

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

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

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

Function Name	Description
BPF_FILTER	Filters the packets of a flow
PKT_COUNTER	Counts the number of packets seen by a network flow
BYTE_COUNTER	Counts the number of bytes seen by a network flow
STR_SEARCH	Searches for a string inside the packet payload
TO_BUFFER	Stores the packets of a flow for further reading
SAMPLE	Samples packets from a flow
HASHSAMP	Samples packets from a flow according to a hashing function
TO_FILE	Dumps the packets of a flow to a file
ETHEREAL	Filters packets using Ethereal display filters
HASH	Computes an additive hash over the packets of a flow
BUCKET	Divides packets into buckets based on their timestamps
THRESHOLD	Signals when a threshold is reached
BINOP	Adds or subtracts values from two other functions
DIST	Returns the distribution of results from another function
GAP	Returns the time delay between two consecutive packets in a flow
PKTINFO	Returns information about a packet
PROTINFO	Returns a specific protocol field
RES2FILE	Stores results from other functions to a file
STARTSTOP	Starts and/or stops measurements at a specific time
STATS	Returns statistical information about results from other functions
BURST	Returns the histogram of bursts

TABLE 2.2: Overview of MAPI `stdlib` functions

users to clearly communicate to the underlying monitoring system these different monitoring needs. To enable users to communicate these different requirements, MAPI allows the association of functions with network flows:

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

The above call 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`). 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 denoted by the flow descriptor `fd` to the TCP packets destined to port `80`, as in flow `A` of Figure 2.2. `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 2.2 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

Function Library	Description
`Standard MAPI Function Library (stdflib)`	Basic functionality for most frequently used monitoring needs
`Extra MAPI Function Library (extraflib)`	Set of functions for advanced monitoring needs (e.g. stream reassembly, regular expression pattern matching, flow data generation)
`Tracker MAPI Function Library (trackflib)`	Identify application-level traffic (e.g. FTP, Gnutella, BitTorrent, etc)
`MAPI Anonymization Function Library (anonflib)`	Anonymization functions for every protocol/field
`Endace DAG Function Library (dagflib)`	Functions intended for better use the capabilities of the DAG capturing hardware

TABLE 2.3: Overview of the MAPI function libraries

bytes of a flow, `SAMPLE`, which can be used to sample packets, `HASH`, for computing a digest of each packet, and `REGEXP`, for pattern matching using regular expressions.

Table 2.2 summarizes the functions of `stdlib` with a short description of each one. For a complete list of the available functions in `stdlib` and their description please refer to [2].

Except from MAPI standard library, several other function libraries are currently exist in MAPI. They offer capabilities like stream reassembly, traffic classification [9], data anonymization [31] and NetFlow-like data generation [12, 43]. Table 2.3 summarizes the function libraries currently implemented in MAPI. Moreover, MAPI users are able to add their own function libraries and new specialized functions for operating on packets.

After applying the desirable list of functions to a network flow, the user calls the function

```
int mapi_connect(int fd)
```

in order to connect to the flow with flow descriptor `fd` to start receiving results.

Retrieving Results from Applied Functions

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 for a data structure that contains the results. The functions that will be applied to the packets of the flow will write their results into this data structure. The user who is interested in retrieving the results will read the data structure using the following call:

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

The above call receives the results computed by the function denoted by the function descriptor `fid`, which has been applied to the network flow `fd`. It returns a pointer to the result's data structure:

```
typedef struct mapi_results {
  void* res;              //Pointer to result data
  unsigned long long ts;  //timestamp
  int size;               //size of the results
} mapi_results_t;
```

The *res* field of this data structure is a pointer to the actual function specific result data. The results are also provided with a 64-bit timestamp, that is the number of microseconds since 00:00:00 UTC, January 1, 1970 (the number of seconds is the upper 32 bits). The memory for the results of each function is allocated once, during the instantiation of the flow.

Reading Packets from a Network 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 packets, supporting this way efficient zero-copy packet capturing platforms [19, 23].

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

```
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`.

2.1.3 MAPI Implementation

Figure 2.3 shows the main modules of MAPI. 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`, is responsible for packet capturing and processing. It is

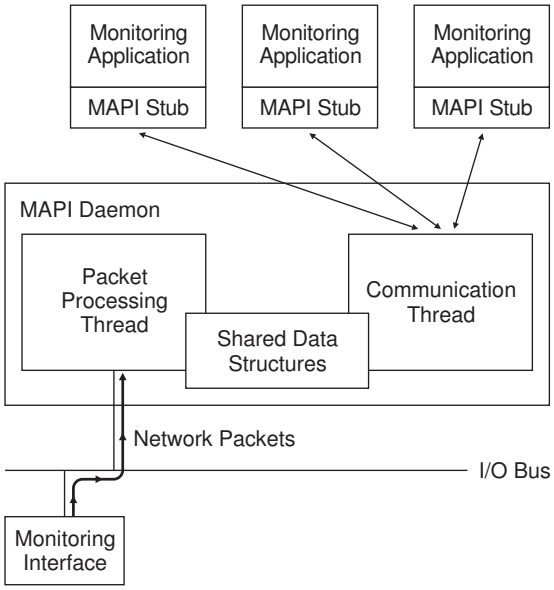

FIGURE 2.3: MAPI Daemon Architecture

implemented as a user-level process, instead of a library connected to an operating system module, because it lead to faster implementation and to a more robust system. `Mapid` is a single process that serve multiple monitoring applications in parallel, so it is possible to perform several performance optimizations and lead to better performance if compared with stand-alone monitoring applications that are not based in MAPI.

The daemon, which has exclusive access to the captured packets, consists of two threads: one data thread for packet processing, and one thread for the communication with the monitoring applications.

All active applications and their defined flows are internally stored in the daemon in a list. Each captured packet is checked by the main processing thread against the defined flow filters. Then, for every flow it belongs to, the packet passes from every function that have been applied in this flow. In that way, the appropriate actions are made for every packet: counters are incremented, sampling, substring search, or other functions that 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 [36], which applies a compiled BPF filter to captured packets in user level.

All communication between the daemon and the monitoring applications is handled by the "communication 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 and sends

them to the applications. Communication between the MAPI stub and `mapid` is performed through Unix sockets.

The MAPI stub is the part of the MAPI library that is transparent to the user. It holds some necessary data structures for the flows that have been created and configured and it is responsible to forward each MAPI call to `mapid` and return the results back to the user.

2.1.4 Example of MAPI usage: Link Utilization

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. The following application periodically reports the utilization of a network link. It uses two network flows to separate the incoming from the outgoing traffic, and demonstrates how to retrieve the results of an applied function.

```
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   int in_fid, out_fid;
12   mapi_results_t *result1, *result2;
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      result1 = mapi_read_results(in_fd, in_fid);
53      result2 = mapi_read_results(out_fd, out_fid);
54      in_cnt = result1->res;
55      out_cnt = result2->res;
56
57      printf("incoming: %.2f Mbit/s (%llu bytes)\n",
58          (*in_cnt-in_prev)*8/1000000.0, (*in_cnt-in_prev));
59      printf("outgoing: %.2f Mbit/s (%llu bytes)\n\n",
60          (*out_cnt-out_prev)*8/1000000.0, (*out_cnt-out_prev));
61
62      in_prev = *in_cnt;
63      out_prev = *out_cnt;
64    }
65
66    return 0;
67  }
68
69  void terminate() {
70    mapi_close_flow(in_fd);
71    mapi_close_flow(out_fd);
72    exit(EXIT_SUCCESS);
73  }
```

The flow of the code is as follows: 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 then we apply the filters that will differentiate the traffic captured by each flow (lines 29– 32). 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, we would define the filter `dst net 139.91.0.0` for the incoming traffic.

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.

After activating the flows (lines 39–46), we enter the main program loop, which periodically calls the `mapi_read_results()` for each flow (lines 52–53) and prints the incoming and outgoing traffic in Mbit/s, and the number of bytes seen in each one second interval (lines 57–60). In each iteration, the current value of each `BYTE_COUNTER` function result is retrieved by dereferencing `in_cnt` and `out_cnt`.

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

2.2 Network Flow Scope

In order to facilitate the concurrent programming and coordination of a large number of remote passive monitoring systems, we have extended MAPI to operate in a distributed monitoring environment. 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 attribute of network flows. In DiMAPI, each flow is associated with a scope that defines a set of monitoring interfaces which 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 MAPI, the input stream of packets 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.

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

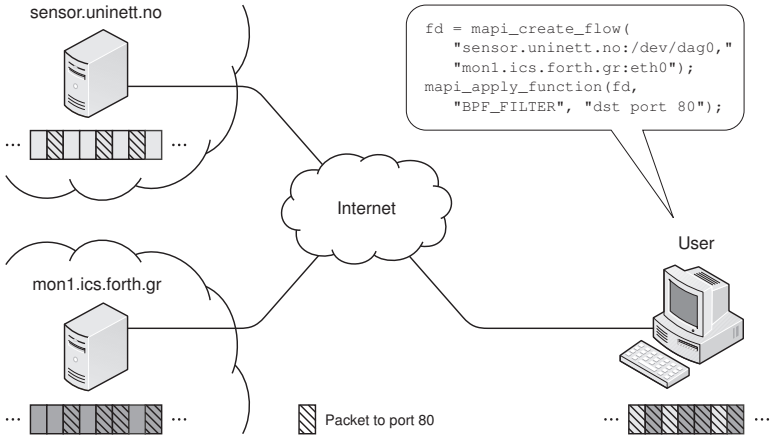

FIGURE 2.4: An example of a network flow scope with multiple sensors

name or IP address of the remote sensor and `interface` is the device name of the monitoring interface. The scope of a network flow 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("m1.forth.gr:/dev/dag0, 123.45.6.7:eth2");
```

In the example of Figure 2.4, 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 applies the `BPF_FILTER` function 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 packets with destination port 80 that are captured from both UNINETT's and FORTH's sensors.

The scope abstraction also allows the creation of flows associated with multiple interfaces 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 at the same monitoring sensor.

```
fd = mapi_create_flow("m1.abc.org:/dev/dag0, m1.abc.org:eth1");
```

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.

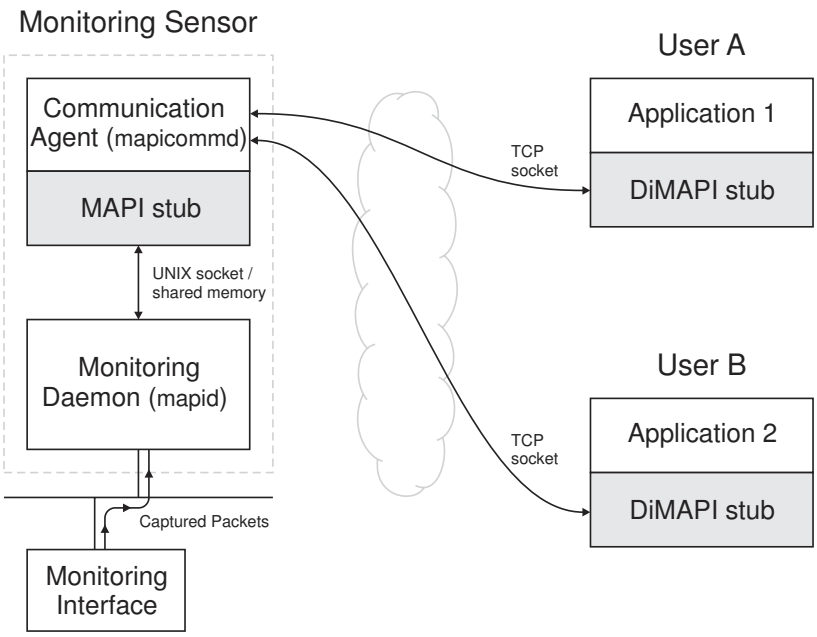

FIGURE 2.5: Architecture of a DiMAPI monitoring sensor

2.3 DiMAPI Implementation

Figure 2.5 illustrates the architecture of a monitoring sensor that supports DiMAPI. The overall architecture includes one or more monitoring interfaces for capturing traffic, a monitoring daemon, which provides optimized passive monitoring services, a DiMAPI stub, for writing monitoring applications, a communication agent, which facilitates communication with multiple remote monitoring applications, and finally, the actual monitoring applications.

The host of the monitoring sensor is equipped with one or more monitoring 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 packet capturing interfaces. Packets are captured and processed by `mapid`, as discussed in Section 2.1.3. `Mapid` is optimized to perform intensive monitoring tasks at high speeds, exploiting any features of the underlying hardware. Local monitoring applications communicate directly with `mapid` via a subset of the DiMAPI stub that is optimized for fast and efficient local access. This is achieved by performing all communication between local applications and `mapid` via shared memory and UNIX sockets [42].

Remote applications must be able to communicate their monitoring requirements to each sensor through the Internet. One possible 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. However, `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. 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 that avoids any modifications to `mapid`, as depicted in Figure 2.5. This is achieved by introducing an *intermediate* agent between `mapid` and the remote applications, for handling all remote communication. This *Communication Agent* (`mapicommd`), which runs on the same host as `mapid`, acts as a proxy for the remote applications, forwarding their monitoring requests to `mapid`, and sending back to them the computed results. The presence of `mapicommd` 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 `mapicommd`. Furthermore, the presence of `mapicommd` is also transparent to `mapid`, since `mapicommd` operates as a typical local monitoring application.

The DiMAPI stub is responsible to support the DiMAPI functionality in a monitoring application, running completely transparently for the user. At the monitoring sensor side, the DiMAPI functionality is solely implemented by `mapicommd`, which is built as a monitoring application that interacts locally with the `mapid`. This allows for a more *robust* system, as communication failures will not result in failure of the monitoring processes. Furthermore, in the 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 `mapicommd`.

In the following sections we look more closely into the operation and implementation of `mapicommd`, DiMAPI stub and their communication protocol. We also examine some privacy and security issues and an alternative implementation for getting results from the monitoring sensors that enhance performance.

2.3.1 Communication Agent

The communication agent runs on the same host with `mapid` and acts as an intermediary between remote monitoring 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 `mapicommd`. The communication agent is a simple user-level process implemented on top of MAPI, i.e., it looks like an ordinary MAPI-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.

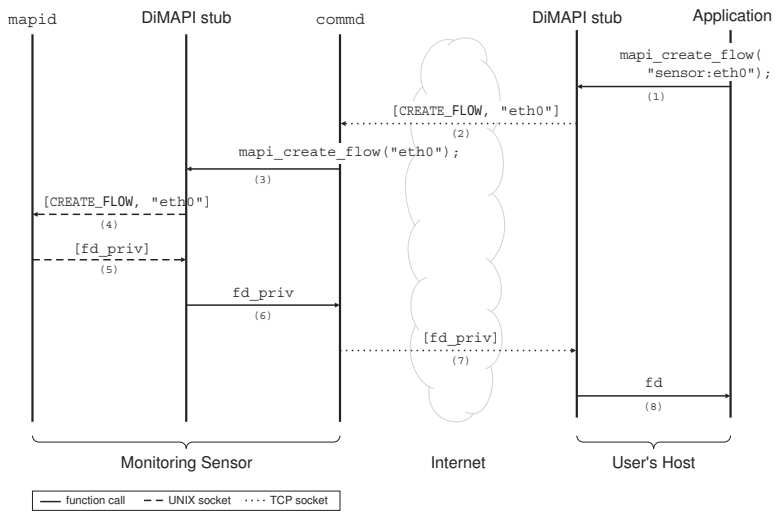

FIGURE 2.6: Control sequence diagram for the remote execution of the function `mapi_create_flow()`

The communication agent listens for monitoring requests from DiMAPI applications to a known predefined port. Then, it runs into an infinite loop, constantly waiting for connections from remote applications. A new thread is spawned for each new remote application, which thereafter handles all the communication between the monitoring application and `mapicommd`. The DiMAPI stub of the remote application sends a control message for each DiMAPI call invocation to `mapicommd`, which in turn repeats the call, though this time the MAPI stub of `mapicommd` will interact directly with the `mapid` running on the same host. `mapicommd` then returns the result to the stub of the remote application, which in turn returns it to the user.

The message sequence diagram in Figure 2.6 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()` call. 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 `mapicommd` running on that host through a TCP socket (step 2). The message contains the type of the DiMAPI call to be executed (`CREATE_FLOW`), along with the monitoring interface that will be used (`eth0`). Upon the receipt of the message, `mapicommd` repeats the call to `mapi_create_flow` (step 3) to the local `mapid`, thus the stub of `mapicommd` 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 `mapicommd` (step 5), which in turn finishes the execution of the `mapi_create_flow()` call by returning `fd_priv` to `mapicommd` (step 6). The communication agent constructs a cor-

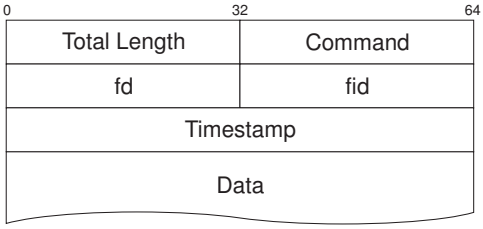

FIGURE 2.7: Format of the control messages exchanged between DiMAPI stub and `mapicommd`

responding reply message that contains the flow descriptor, and sends it back to the DiMAPI stub of the user application (step 7). In case that the network flow is associated with more than one monitoring sensors, steps 2–7 are repeated for each sensor of the network flow's scope and 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. Finally, the DiMAPI stub of the application generates and returns back to the user a new unique flow identifier (`fd`), and internally stores the mapping between the received flow descriptors (`fd_priv`) and the newly created identifier (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 Section 5.1.

2.3.2 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 for 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 through TCP sockets. DiMAPI stub library calls exchange control messages with `mapicommd` 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 the `mapicommd` is shown in Figure 2.7. Each message has variable length, denoted by the field `Total Length`. The `Command` field contains the operation type, sent by the stub to `mapicommd`, or the acknowledgement value for a request that `mapicommd` has processed. It takes values from an enumeration of all message types that can be exchanged between the stub and `mapicommd`. For example, for a call to `mapi_create_flow()`, the relevant message sent from the stub will have a `Command` value of `CREATE_FLOW`, for a call to `mapi_apply_function()` `Command` will be `APPLY_FUNCTION`, for a successful create flow the response will be `CREATE_FLOW_ACK` 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. For example, when the message is a reply from `mapicommd` to a call of `mapi_read_results()`, it contains the results of an applied function. If it is a reply of a `mapi_get_next_pkt` call then the `Data` filed contains a captured network packet. If the `Command` field contains a request sent from the DiMAPI stub, e.g., to apply some function to a network flow, it contains the arguments of the relevant function (e.g. the name of the function to be applied along with its arguments).

2.3.3 DiMAPI Stub

In this section we discuss some implementation issues regarding the DiMAPI stub on the monitoring application's side.

Creating and Configuring Network Flows using Multiple Remote Sensors

In order to support the scope functionality, DiMAPI stub has been 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,
    mon.cesnet.cz:eth0, mon1.ics.forth.gr:eth0");
```

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

```
sensor.uninett.no: fd_uninett = mapi_create_flow("/dev/dag0");
mon.cesnet.cz:     fd_cesnet = mapi_create_flow("eth0");
mon1.ics.forth.gr: 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 `mapicommd` will send back to DiMAPI stub an acknowledgement message containing the flow descriptor of the flow that it created remotely (`fd_uninett`, `fd_cesnet`, and `fd_forth`, respectively). The DiMAPI stub will generate a unique flow identifier (`fd`), and will internally store 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 communication agents of the three sensors the following corresponding messages:

```
sensor.uninett.no: [APPLY_FUNCTION, fd_uninett, PKT_COUNTER]
mon.cesnet.cz:     [APPLY_FUNCTION, fd_cesnet, PKT_COUNTER]
mon1.ics.forth.gr: [APPLY_FUNCTION, fd_forth, PKT_COUNTER]
```

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 `mapicommd`. DiMAPI stub stores a similar mapping for the function identifier `fid`, and acts in a similar fashion whenever it is manipulated.

Using Communication Threads

Since the monitoring sensors are distributed located on several 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. For this reason, it is not acceptable to send a control message and waiting for reply from each remote monitoring sensor one-by-one. Instead, the receipt of incoming messages in DiMAPI stub is implemented using a separate "communication thread" for each remote monitoring sensor used by the application (i.e., for each TCP socket created by the stub). Each communication thread is responsible for receiving the replies of pending MAPI calls from one remote sensor, and delivering them to the appropriate function.

A DiMAPI call prepares and sends a control message to each involving monitoring sensor, and then it blocks by pushing down a semaphore variable. The communication thread waits infinitely in a loop for incoming replies from the corresponding `mapicommd`. 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 pushing up the corresponding semaphore. When the execution of the blocked call resumes, it retrieves the result from the buffer and processes it accordingly. This implementation guarantees that the incoming messages are always delivered to the call that sent the relevant request.

Reading Results from Multiple Remote Sensors

While in local MAPI the `mapi_read_results()` function returns a single instance of `mapi_results_t` struct (see 2.1.2), in DiMAPI it returns a vector of `mapi_results_t` structs, one for each remote monitoring sensor (in the same order that these sensors had been declared in `mapi_create_flow()`.

In order to know the number of the remote monitoring sensors that our network scope consists of, and so the number of the `mapi_results_t` instances that `mapi_read_results()` will return, we use the *mapi_get_scope_size()* function:

```
int mapi_get_scope_size(int fd)
```

This function takes as a single argument the flow descriptor and returns the number of the corresponding monitoring sensors. In case of a local MAPI application, it returns 1. In this way we provide full compatibility between MAPI and DiMAPI applications.

Fetching Captured Packets

In DiMAPI, the *mapi_get_next_pkt()* returns packets from the monitoring sensors in a round-robin way, if it is possible. 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 (with size of one packet per each sensor). 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. Before returning the packet, a new `GET_NEXT_PKT` request is sent only to this sensor. In case of consequent `mapi_get_next_pkt()` requests, all slots are checked in a round-robin way, beginning from the slot that was emptied in the previous call. The next `GET_NEXT_PKT` request is sent to the sensor whose slot was emptied, before returning the packet to the application, which ensures that all slots will be always full, or at least have one pending request.

2.3.4 From Pull to Push Model

The current approach in `DiMAPI` functions is that the applications (transparently through the DiMAPI stub) should send a request to the remote monitoring sensor before receiving any results. This operation is similar to the definition of *pull model* in the Distributed Systems theory. Figure 2.8 depicts the operation and message exchanges that occur in the case of a `mapi_get_next_pkt()` call. The same operations are occurred in every MAPI call.

First, the MAPI application sends a request to the communication agent. Then, the communication agent calls locally the corresponding MAPI function, using the local `mapid` that runs in the same machine, and gets the results. Finally, the communication agent sends the results back to the MAPI application. So, it is clear

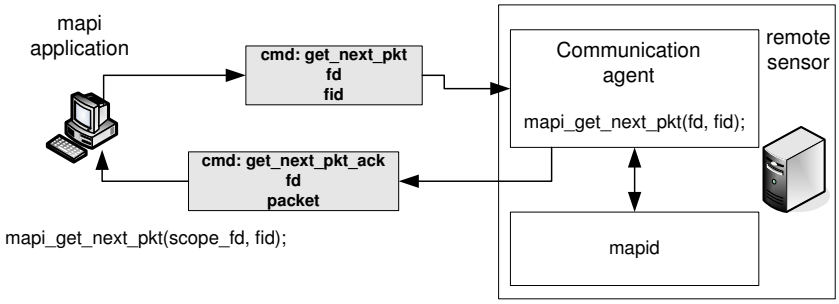

FIGURE 2.8: Pull model operation and message exchanges in DiMAPI

to see that the latency of these functions is equal to the real network round-trip time (RTT) between the host where the application runs and the remote sensor, because DiMAPI sends a request to the remote sensor and waits for the response.

The functions that DiMAPI provides for the creation and initialization of a network flow (`mapi_create_flow()`, `mapi_apply_function()` and `mapi_connect()`) are called only once per every network flow creation, so we focus on the functions used for retrieving data from the monitoring sensors: `mapi_read_results()` and mainly `mapi_get_next_pkt()`.

For `mapi_read_results()`, DiMAPI stub waits all the monitoring sensors to respond, so the latency and the throughput will be equal to the RTT of the slowest remote sensor. In `mapi_get_next_pkt()`, it returns the first packet that will arrive from any of the monitoring sensors. Furthermore, just before returning to the user a packet from a remote sensor, it sends immediately a request for a new packet to this sensor, as a prefetching technique, in order to have the buffer always full with one packet from every remote sensor and returns a packet immediately to the user in the next call, if possible. So, if the user of DiMAPI calls `mapi_get_next_pkt()` in a period larger than the fastest host's RTT, the latency will be just a few milliseconds in every call. However, if the user wants to call `mapi_get_next_pkt()` in a smaller period that this RTT, the latency will be equal to this network RTT. So, even if we can decrease latency in some cases due to the one-packet prefetching technique, the throughput of this function still depends on the network RTT of the fastest remote sensor. This is due to the usage of the pull model, which requires one request for receiving one packet.

In order to improve the performance of the DiMAPI functions in terms of latency and throughput, we implemented a second approach. In this approach the monitoring sensor sends results (or packets) back-to-back to the remote application, without waiting for requests (*push model*). The application itself is not aware of this operation, it is handled transparently by the DiMAPI stub instead. The stub sends a single request when the application is ready to receive results or packets. In case of `mapi_read_results()` the request contains the time interval that the application desires to receive new results. For `mapi_get_next_pkt()`, it

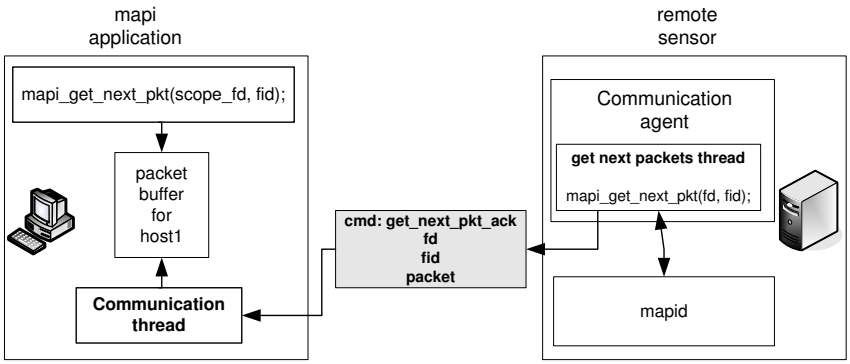

FIGURE 2.9: Push model operation and message exchanges in DiMAPI

contains the number of packets that the stub is willing to receive. The results are buffered in the stub and consecutive calls to these functions will be served immediately from the stub, since the results have been prefetched, without need for sending any requests to the sensor.

This mechanism for getting results from an applied function is activated at the first call of `mapi_read_results` for this function, by sending a single request to the `mapicommd` defining the time interval that it should ask and propagate new results to the stub. Then, `mapicommd` will create a new thread that will start to periodically call `mapi_read_results` using the local `mapid` and send the results back to the application. In the application's side, the results from the communication agent will be received from the respective communication thread that handles all the communication with this remote host and will be stored to a corresponding buffer. The next calls to `mapi_read_results()` from the application will result to return immediately the results from the buffer that they have been stored. Assuming that the time interval for results generation can be set in a suitable value, this approach seems very promising for significant improvement in terms of latency and throughput for the `mapi_read_results()` call.

Similarly for `mapi_get_next_pkt()`, the first call results to a request destined to the `mapicommd` for fetching a number of packets (defined in the request) to the application back-to-back. Upon this request arrives at `mapicommd`, a new thread is created asking for packets from `mapid` and immediately forwards them to the application's stub, till the number of requested packets is reached. Since the packets will arrive in batches, before the application will actually ask for them (*prefetching*), they must be saved in a buffer located inside the DiMAPI stub with size equal to the number of packets that were requested. Figure 2.9 depicts the operation of *push model* in *mapi_get_next_pkt()*. We also call this approach as *packet prefetching* since the stub transparently gets a number of packets before the application has actually requested these packets. In every subsequent call of `mapi_get_next_pkt()` the stub will return the next packet from the local buffer that the packets are stored. When the packet buffer is getting empty under a spec-

ified threshold (e.g. 10%), a new request will be sent to `mapicommd` for starting again to send a number of packets.

Since the monitoring sensor sends the captured packets immediately one after the other, we expect that the throughput will be dramatically increased and since the stub will return most of the packets to the application from the local buffer we also expect a significant improvement to the `mapi_get_next_pkt()` latency. In section 5.1.4 we present an experimental evaluation of the *push model* implementation in `DiMAPI` for `mapi_get_next_pkt()`. We examine the benefit that this approach can give for different rates of the monitoring traffic and while trying several sizes for the buffer that holds the packets (equal to how many packets will be prefetched to the application's stub).

2.3.5 Security and Privacy

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 that may contain sensitive information. For protection against such threats, any communication between the DiMAPI stub and a remote sensor can be 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.

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.

In a distributed monitoring infrastructure that promotes sharing of network packets and statistics between multiple 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. DiMAPI supports an advanced framework for creating and enforcing anonymization policies [31]. 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.

2.4 Examples of DiMAPI Usage

In this section we describe two simple monitoring applications built on top of DiMAPI. The first is a simple byte counter for web traffic, and the second is an application that detects covert traffic from a specific peer-to-peer file sharing client. Note that these are illustrative examples, two more complicated monitoring applications that exploit the power of DiMAPI are presented in chapter 3.

2.4.1 Web Traffic Byte Counter

The following code illustrates a simple DiMAPI application that counts the total bytes of the packets received by the web servers of multiple monitored networks within a predefined interval.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mapi.h>
4
5  int main() {
6    int fd;
7    int fid;
8    mapi_results_t *dres;
9    unsigned long long bytes, total_bytes=0;
10   int i, number_of_sensors;
11
12   /* create a flow using a scope of three monitoring sensors */
13   fd = mapi_create_flow("sensor.uninett.no:/dev/dag0,
14         mon1.ics.forth.gr:eth0, 123.45.6.7:eth2");
15   if (fd < 0) {
16     printf("Could not create flow\n");
17     exit(EXIT_FAILURE);
18   }
19
20   /* keep only packets directed to a web server */
21   mapi_apply_function(fd, "BPF_FILTER", "tcp and dst port 80");
22
23   /* and just count the bytes */
24   fid = mapi_apply_function(fd, "BYTE_COUNTER");
25
26   /* connect to the flow */
27   if (mapi_connect(fd) < 0) {
28     printf("Could not connect to flow %d\n", fd);
29     exit(EXIT_FAILURE);
30   }
31
32   /* get the number of the monitoring sensors */
33   number_of_sensors = mapi_get_scope_size(fd);
34
```

```
35    sleep(10);
36
37    /* get the vector with results from every sensor */
38    dres = mapi_read_results(fd, fid);
39
40    for (i=0; i<number_of_sensors; i++) {
41      bytes = *(unsigned long long*) dres[i].res;
42      printf("Web bytes in sensor %d: %llu\n",i, bytes);
43      total_bytes += bytes;
44    }
45
46    printf("Total bytes to web servers: %llu\n",total_bytes);
47
48    /* close the flow */
49    mapi_close_flow(fd);
50
51    return 0;
52  }
```

The above application operates as follows. We initially define a network flow with a scope of three remote monitoring sensors (line 15). Then, we restrict the packets of the flow to only those destined to some web server, by applying the `BPF_FILTER` function (line 23). After specifying the characteristics of the network flow, we instruct the monitoring system that we are interested in just counting the number of bytes of the flow, by applying the `BYTE_COUNTER` function (line 26). Finally, we activate the flow (line 29). After 10 seconds, the application reads the result by calling `mapi_read_results()` (line 40). The `mapi_get_scope_size()` function gives up the number of the monitoring hosts that should give results, one `mapi_results_t` instance per every monitoring sensor. The actual result of the `BYTE_COUNTER` function for the monitoring sensor *i* is retrieved from *dres[i].res* field. Using a loop we read the bytes of the web traffic from each monitoring sensor separately and we compute the total bytes by adding them (lines 42–46). Our work is done, so we close the network flow in order to free the resources allocated in every `mapid` (line 51).

2.4.2 Covert Peer-to-Peer Traffic Identification

The second example is an application that identifies covert traffic from Gnutella file sharing clients. Several Gnutella clients offer the capability to operate using HTTP traffic through port 80, thus hiding as normal web traffic, in order to bypass strict firewall configurations tha 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 <mapi.h>
```

```
4
5  int main() {
6    int fd;
7    int fid;
8    mapi_results_t *dres;
9    unsigned long long pkts, total_pkts;
10   int i, number_of_sensors;
11
12   /* create a flow using a scope of three monitoring sensors */
13   fd = mapi_create_flow("sensor.uninett.no:/dev/dag0,
14         mon1.ics.forth.gr:eth0, 123.45.6.7:eth2");
15   if (fd < 0) {
16     printf("Could not create flow\n");
17     exit(EXIT_FAILURE);
18   }
19
20   /* keep only web packets */
21   mapi_apply_function(fd, "BPF_FILTER", "tcp and port 80");
22
23   /* indicating Gnutella traffic */
24   mapi_apply_function(fd, "STR_SEARCH", "GNUTELLA CONNECT");
25
26   /* and just count them */
27   fid = mapi_apply_function(fd, "PKT_COUNTER");
28
29   /* connect to the flow */
30   if (mapi_connect(fd) < 0) {
31     printf("Could not connect to flow %d\n", fd);
32     exit(EXIT_FAILURE);
33   }
34
35   /* get the number of the monitoring sensors */
36   number_of_sensors = mapi_get_scope_size(fd);
37
38   /* forever, report the number of packets */
39   while(1) {
40     sleep(60);
41     dres = mapi_read_results(fd, fid);
42     total_pkts=0;
43     for (i=0; i<number_of_sensors; i++) {
44       pkts = *(unsigned long long*) dres[i].res;
45       printf("Gnutella packets in sensor %d: %llu\n",i, pkts);
46       total_pkts+= pkts;
47     }
48     printf("Total Gnutella packets: %llu\n",total_pkts);
49   }
50
51   return 0;
52 }
```

Similarly to the previous example, we initially create a network flow with a scope of three remote monitoring sensors (line 13), and apply the `BPF_FILTER` function to keep only the packets seemingly destined to, or coming from, a web server (line 21). 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 24). After specifying the characteristics of the network flow, we instruct the monitoring system that we are interested in just counting the number of packets, by applying the `PKT_COUNTER` function (line 27). Finally, we activate the flow (line 30). At this point, each monitoring sensor has started inspecting the monitored traffic for covert Gnutella traffic, and keeps a count of the matching packets. Then, the application periodically reads the result of the `PKT_COUNTER` function by calling `mapi_read_results()` in a infinite loop (lines 39–49).

2.5 Advantages of DiMAPI

Implementing similar distributed monitoring applications, like those presented in the previous section, using other existing tools and libraries except DiMAPI would have been a much more difficult process, resulting in longer code, higher overheads and overall reduced performance.

One alternative solution is to build these applications using solely `WinPcap` [5] or `rpcap` [32]. Both libraries extend `libpcap` [36] with remote packet capture capabilities, allowing captured packets at a remote host to be transferred to a local host for further processing. For example, in order to count the covert Gnutella packets using one of these libraries, the application has to first transfer locally *all* the captured web packets, separately from each remote sensor, then identify locally the Gnutella packets, count them, and finally drop them. The pattern matching operation has to be performed locally since `libpcap` does not offer any pattern matching operation. However, transferring all the web packets from each remote sensor to the local application incurs a significant network overhead. In case of many remote interfaces, the scalability of this approach renders it practically infeasible. In contrast, DiMAPI enables traffic processing at each remote sensor, which allows for sending back only the computed results. In this case, only the *count* of Gnutella packets is transferred through the network, which incurs substantially less network overhead.

An other approach would be to use tools like `snort` [45] or `ngrep` [44], which allow for pattern matching in the packet payload, for capturing the Gnutella packets at each remote sensor. At the end-host, we should have to use some scripts for starting and stopping the remote monitoring applications and for retrieving and collectively reporting the results, through some remote shell such as `ssh`. How-

ever, 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. The monitoring daemon on each sensor groups and optimizes the monitoring operations requested by the users of the system, providing the same or even better performance compared to `libpcap` [42]. Consequently, using DiMAPI we can achieve more effective distributed passive network monitoring than with any other existing tool or library.

3

Applications Based on DiMAPI

Traditionally, passive monitoring tools operate at a selected vantage point in the network that offers a broad view of the traffic, such as the access link that connects an Autonomous System to the Internet. Besides monitoring a single link, emerging applications can benefit from monitoring data gathered at multiple observation points across a network [15, 24, 27]. Such a distributed monitoring infrastructure can be extended outside the border of a single organization and span multiple administrative domains across the Internet [49]. In this environment, the processing and correlation of the data gathered at each sensor can give a broader perspective of the state of the monitored network.

Several applications can benefit from such a distributed passive monitoring infrastructure by deriving useful network metrics regarding the network conditions between different domains. These metrics include Round-Trip Time [29], per-application throughput, packet retransmissions [20], packet reordering [37], one-way delay and jitter, and packet loss ratio [40]. In this thesis, we focus on the passive estimation of the packet loss ratio between different domains, which is a typical application that takes advantage of the use of DiMAPI. In the remaining of this chapter we discuss the advantages of a passive packet loss measurement approach and we describe in detail the design and implementation of this technique. Moreover, we present the design of a Grid network monitoring element for passive monitoring Grid network infrastructures using DiMAPI. This is also a typical application that can significantly benefit from DiMAPI.

3.1 Passive End-to-End Packet Loss Estimation

Accurate monitoring of network characteristics, such as delay, packet loss rate, and available bandwidth, is critical for the efficient management and operation of modern computer networks. One of the most important network performance metrics is the packet loss ratio. Packet loss occurs when correctly transmitted packets from a source never arrive at the intended destination. Packets are usually lost due to congestion, e.g., at the queue of some router, routing problems, or poor network conditions that may result to datagram damages. Packet loss affects significantly the data transfer throughput and the overall end-to-end connection quality. Consequently, it is desirable to have accurate packet loss measurements for the network paths that several services use, in order to timely identify network inefficiencies.

Most of the existing techniques for packet loss estimation are based on *active* network monitoring, which usually involves the injection of a certain number of packets into the network for measuring how many of them are lost [6,46,47]. Such active monitoring tools incur an unavoidable network overhead due to the injected probe packets, which compete with the real user traffic.

In contrast to above approaches, in this application we present a novel real-time, end-to-end packet loss estimation method based on distributed *passive* network monitoring, based on DiMAPI. Our approach does not add any overhead to the network since it passively monitors the network traffic without injecting any probe packets. At the same time, it estimates almost in real-time the *actual* packet loss faced by the active applications. Moreover, it offers the capability for measuring the loss rates of particular services, allowing for fine-grained per-application packet loss estimation, which is important in case different applications on the same path face different degrees of packet loss. The design, implementation and experimental evaluation of the passive packet loss estimation application is presented in more detail in [40].

3.1.1 Existing Tools

Previous work on packet loss estimation can be broadly categorized into approaches based on passive and active network monitoring, with the latter having a significantly larger literature body.

One of the most popular tools for inferring the basic network characteristics, such as round-trip time and packet loss, is `ping`. `Ping` uses the ICMP protocol to send probe packets to a target host at fixed intervals, and reports loss when the response packets are not received within a specified time period. However, ICMP packets are often rate limited, or blocked, by routers and firewalls. An other active tool is `zing` [6], which estimates the end-to-end packet loss in one direction between two cooperative end hosts, by sending UDP packets at Poisson modulated intervals with a fixed mean rate. `Badabing` [47] also measures the one-way packet loss by sending fixed-size packets at specific intervals. `Sting` is an active monitoring tool that measures the loss rate in both forward and reverse directions

from a single host to any TCP-based server, by exploiting TCP's loss recovery algorithms [46]. Finally, network tomography using unicast probes has been used for inferring loss rates on end-to-end paths [21].

Besides active tools, there also exist methods that use passive network monitoring for measuring the TCP packet loss, based on the TCP retransmission mechanism [11]. However, there are several applications, such as `tftp`, which use UDP instead of TCP. Techniques for estimating the loss rate based on the TCP protocol are also presented in [7], however they work only in individual clients and they cannot be used by other external applications, e.g., for improving routing or selecting a replicated server with the best network conditions.

3.1.2 Passive Packet Loss Measurement Characteristics

An inherent property of passive network monitoring is that it does not disrupt the existing traffic conditions. This non-intrusive nature of passive measurements makes them completely invisible on the network. Moreover, our passive packet loss estimation method exhibits several other advantages over active packet loss measurement techniques, which we discuss in the following.

Real-time measurement of the *actual* packet loss ratio. The proposed technique measures the actual packet loss faced by the traffic of an active application in real-time, as it passes through the passive monitors. In contrast, active monitoring approaches unavoidably disrupt the current traffic due to the probe packets. Thus, they can measure potential temporary side effects that may be caused by the injected traffic.

Scalability. In a large-scale network monitoring infrastructure, it is desirable to measure the end-to-end packet loss between many different resources or domains. In a system with N resources, the number of required end-to-end measurements grows with $O(N^2)$, since, as a general rule, each resource has a distinct path to any other resource. For active monitoring, it is clear that as the number of resource pairs increases, the injected traffic incurs a significant disruption in the network, so usually such measurements are performed sequentially, measuring one or a few paths at a time. In contrast, a passive monitoring approach can provide an instant estimation of the packet loss ratio across different paths, independently of their number.

Per-application measurement. Using appropriate filters, the proposed approach can measure the packet loss faced only by the traffic of a particular service. This capability is of particular importance for cases in which different services may exhibit different packet loss ratio in the same path. This can happen due to the use of differentiated services, rate-limiting devices, or load-balancing configurations.

IP-level measurement. In contrast to techniques that passively estimate the loss ratio based on properties of the TCP protocol [7, 11], our approach measures

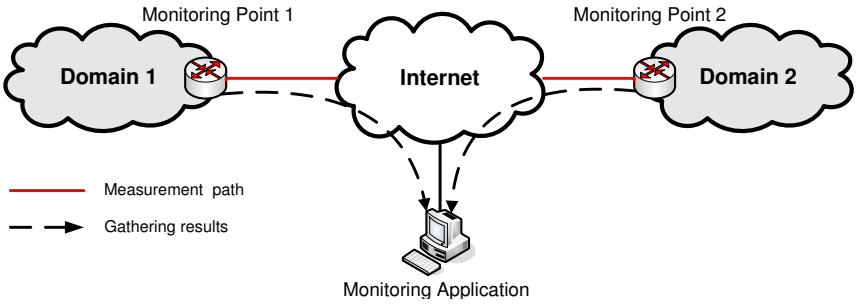

FIGURE 3.1: End-to-end architecture for passive packet loss estimation.

the packet loss at the IP layer, so it can also work for UDP or any other Transport Layer protocol.

Besides the above positive properties, our approach has also certain limitations. A necessary operational requirement is the presence of two passive monitors at the ends of the measured path. If passive traffic monitoring is not feasible in some domain, then we should rely on active monitoring tools. Furthermore, the presence of real traffic in the measured path is mandatory for the operation of our approach, since it measures the packet loss faced by the existing traffic. It is clear from the above that our approach is complementary to existing active probing techniques, and both approaches can perfectly coexist.

3.1.3 Approach

We adopt an end-to-end approach for estimating the packet loss ratio using two passive monitors at the two ends. The overall approach is shown in Figure 3.1. The two monitoring points gather information about the packets passing through them. Periodically, this information is sent to a central application which correlates these results and estimates the packet loss ratio.

A naive packet loss algorithm in this environment would just count at both ends the number of packets in each direction between the two domains, and then periodically subtract the number of packets received at the destination from the number of packets that were actually sent, and vise versa. However, this simple method has a major drawback: we cannot accurately synchronize the two monitoring points to count the same window of packets. Suppose that both passive monitors start and stop counting packets at exactly the same time. When they start counting, some packets are already in transit. These packets were not counted at the sender side, but they will be counted at the receiver, so the packet loss ratio will be underestimated. Similarly, when the measurement stops, the in transit packets will have been counted by the sender, but will be missed by the receiver, so the packet loss ratio will be overestimated. A possible solution would be to start and stop the measurement in the receiver's monitoring point after a delay close to the network's one-way delay. However, this solution is still inaccurate due to the network delay

variability. Even the loss of a single packet can be significant, e.g., in long haul tcp connections.

In order to solve the above problem, we take a different approach by measuring the packet loss of each *flow* separately. For the TCP and UDP protocols, a *flow* is defined as a set of IP packets with the same protocol, source and destination IP address, and source and destination port (also known as a 5-tuple). For protocols that do not support ports, a flow is defined only by the protocol and source and destination IP address. A flow is considered *expired* if no packet has arrived for that particular flow within a specified timeout (60 sec in our experiments). In case of TCP, a flow can also be considered expired if the connection is explicitly closed, i.e., when an RST of FIN packet is seen.

Each of the two monitoring sensors classifies the IP packets into flows, according to the above definitions. In periodic time intervals, both sensors send statistical information about the identified *expired flows* to the monitoring application. Since expired flows are well defined, the monitoring application can correlate the statistics gathered at both sensors regarding the *same* expired flow. Thus, for each pair of statistics regarding the same expired flow, the application computes the packet loss for that flow based on the difference of the number of packet that each expired flow reports. This gives an accurate measurement of the *actual* packet loss faced by the particular traffic flow.

3.1.4 Implementation

Distributed Passive Monitoring Platform

In each measurement point we need a passive traffic monitoring platform for the identification and collection of the expired flows. We have implemented our prototype using MAPI [42], a flexible passive monitoring API. A communication agent, part of the distributed MAPI version [49], is responsible for accepting monitoring requests from remote applications and sending back the corresponding results. Using this distributed monitoring API (DiMAPI), we are able to manipulate multiple monitoring sensors from the same application.

Identification of Expired Flows

Every packet is associated with exactly one flow. At each sensor, the monitoring daemon keeps a record for each active flow in a hashtable for fast lookup. In addition to the 5-tuple, a flow record holds the timestamps of the first and last packet of the flow. The arrival time of the last packet of the flow is necessary for deciding whether the flow has expired or not. Finally, the record holds the number of packets and bytes of the flow, from which we compute the packet and byte loss ratios.

For every new packet, the monitoring daemon looks up the corresponding flow record in the hashtable, increases the packet counter, and adds the packet size to

the existing byte counter value. Also, the timestamp of the last packet is renewed. In case a flow record is not found, a new one is created.

In order to identify immediately the expired flows, the monitoring daemon maintains a linked list that contains pointers to the flow records in a temporal order. For every new packet, the timestamp of the last packet in the corresponding flow record is renewed, and that flow comes first in the linked list. A separate thread in the monitoring daemon runs periodically (e.g., every one second) and finds the expired flows in the end of that list. Starting from the last entry of the list, it checks whether the timestamp of the last packet of that flow is older than the specified timeout, and if so, it removes it from the list and puts it in the expired flow list. The same process is continued until a non-expired flow is found. Finally, the monitoring daemon sends the list with the expired flows to the monitoring application.

Distributed Monitoring Sensor Management

The last component of the architecture is the monitoring application. The application collects periodically the expired flows from the distributed monitoring sensors, using the DiMAPI functionality, correlates them, and reports the packet loss ratio for every pair of sensors. The application uses a hashtable, similar to the one described earlier, for identifying pairs of statistics from different sensors for the same expired flow. For every matched pair, it subtracts the number of packets that they measured to compute the packet loss for this flow. Finally, the application reports the total packet loss ratio between pairs of measurement points and also the packet loss per every individual flow. It reports the byte loss ratio as well, which can be also an interesting metric for some applications.

3.2 Grid Network Monitoring Element

Accurate monitoring of network characteristics, such as delay, packet loss rate, and available bandwidth, is critical for the efficient operation of modern Grid systems, which are usually composed of many resources interconnected by local area networks or, more often, through the Internet. Network monitoring can be used for Grid performance debugging, since Grid-enabled applications are highly dependent on network characteristics, and for performing complete diagnosis when the applications are not working as expected. Using network monitoring we can usually find the source of the problem. Also, network monitoring can be used in Grid systems for resource allocation and scheduling decisions.

Active and passive monitoring can be combined in a Grid Network Monitoring Element. The main benefit of passive monitoring, compared to active monitoring, is its non-intrusive nature. Active monitoring tools, such as the ubiquitous `ping`, incur an unavoidable network overhead due to the injected probe packets, which compete with the real user traffic. In contrast, passive monitoring techniques observe the existing traffic of the monitored link passively, without introducing any

additional network traffic. Also, passive techniques measures at real time the actual performance, while active techniques may measure temporary side effects. Moreover, using passive monitoring we are able to perform per application measurement or even measurements in the IP level. Active monitoring tools usually rely on a specific protocol and furthermore they are often blocked by firewalls or rate limited. On the other hand, active tools are usually easy to deploy, while passive monitoring require the installation of passive monitors. Also, active tools use the desired traffic patterns and inject packets at any time. Passive measurements requires the presence of real traffic in the measurement path. Consequently, the best approach seems to combine both active and passive monitoring in a Grid network monitoring element. Each approach provides different tools and probably different metrics. We should try to use passive monitoring whenever it is applicable, due to the advantages we discussed above, else we should use active probes to generate our own measurement traffic.

Using passive monitoring, we can infer several network characteristics: perform Grid traffic categorization and accounting (e.g. find what percentage of the traffic is GridFTP, or find which subnet generates the most outgoing traffic), bandwidth estimations, performance debugging of individual applications and security applications (Denial-of-Service attack detection, Internet epidemics and intrusion detection). So, in a Grid environment, passive monitoring can play an important role for assessing the status of the Grid infrastructure connectivity and for taking effective balancing decisions. Grid applications can also benefit from a distributed passive monitoring infrastructure [49] by using it to derive useful network metrics between different domains. Such metrics include among others the network Round-Trip Time [29], application-level RTT [25], per-application throughput, packet retransmissions [20], packet reordering [37], one-way delay and jitter, and packet loss ratio [40].

In this section we define how passive network measurements are configured inside a grid-wide Network Monitoring Service. This service is based on a Network Monitoring Element (NMElement), which is a Grid element that concentrates the Network Monitoring functionalities of a Grid: it offers an interface for measurement requests coming from applications, and a plug-in based interface for publishing measurements. It has access to a database that contains the description of the domain partitioning of Grid resources, and the persistent attributes of other NMElements. Finally, the DiMAPI daemons for monitoring and communication (`mapid` and `mapicommd`) run inside the NMElement. A detailed description of its functionalities can be found in [14].

The definition of a Network Monitoring session [13] aiming to passive network measurements is composed of the following elements: the identifiers of the source and destination domains, the description of the type of service for which the passive measurement is requested, and the time period of the measurement: this can be historical, most recent, one-shot, or periodic. Certain combinations of these attributes are also allowed.

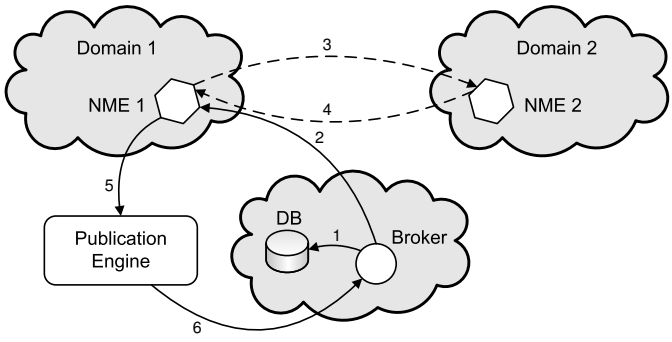

FIGURE 3.2: Embedding passive network measurements in a Grid Network Monitoring Service

In principle, a measurement is not targeted to a flow between two specific hosts: the domain partitioning should guarantee the significance of the measurement for any pair of hosts in the two domains.

Figure 3.2 illustrates the message exchange between the agents that participate to the measurement, as described in the following. Messages are represented by arrows, in which the attached numbers are referenced in the following text. In order to implement this distributed architecture for passive monitoring and measurements we employ DiMAPI.

The application that needs the measurement will send a measurement request (**2**) to one of the the NM services in charge of monitoring the request between the two domains. This can be either the source or the destination of the flow which we are interested to passively monitor. The information regarding the identity of the NMElement, necessary in order to handle the request, is first retrieved with a query (**1**) to the NM Database attached to a NMElement in the domain of the requesting application.

When the Network Monitoring service receives such request, it first checks the availability of the module in charge of managing the measurement. The information is retrieved from an internal registry of available modules. The next step consists of verifying the availability of resources dynamically allocated to monitoring tasks: this information is retrieved by inspecting the current system state (using ps/netstat like commands).

In case any of the above steps fail, a “resource not available” reply is returned to the calling application. This indicates that the measurement was not performed, but does not imply anything about the availability of the inspected resource. The application will redirect the request to another NMElement, or will repeat it using less demanding parameters. If the measurement is feasible, the successive step consists of locating a peer NMElement: the selection is carried out using the local NM Database, by querying for the peer NMElement, which is identified by the (source domain, destination domain) pair.

The measurement can be either extracted from a local cache of available results, or actually come from a new measurement. In the former case, the historical result is found as indexed by the Network Element, complemented by measurement attributes indicated in the request of the application. Otherwise, a request for the activation of the peer module for a passive measurement is delivered to the peer (**3**) using DiMAPI. In case of a negative reply, this is bounced back to the requesting application. Otherwise, the measurement will proceed normally. The peer module will send back the measured data for the passive measurement (**4**), through DiMAPI.

The result of the measurement is finally streamed outside the NMElement, either to the GIS, or to any other publication media (**5**), according to the available plugin in the NM Service module. The final step is the delivery of the result to the requesting application (**6**).

4

Improving the Performance of Packet Processing using Locality Buffering

Passive network monitoring is the basis for a multitude of systems that support the robust, efficient, and secure operation of modern computer networks. While passive monitoring has been traditionally used for relatively simple network traffic measurement and analysis applications, or just for gathering packet traces that are analyzed off-line, in recent years it has also become vital for a wide class of more CPU and memory intensive applications, such as Network Intrusion Detection Systems (NIDS) [45], accurate traffic categorization [9], and NetFlow export probes [1]. The complex analysis operations of such demanding applications are translated into an increased number of CPU cycles spent on each captured packet, which reduces the overall processing throughput that the application can sustain without dropping incoming packets. At the same time, as the speed of modern network links increases, there is a growing demand for more efficient packet processing using commodity hardware that can keep up with higher traffic loads.

A common characteristic that is often found in such monitoring applications is that they usually perform different operations to different types of packets. For example, a NIDS applies a certain subset of attack signatures to packets with destination port 80, i.e., it applies the web-attack signatures to packets destined to web servers, it applies a different set of signatures to packets destined to database servers, and so on. Furthermore, NetFlow probes [1], traffic categorization, as well as TCP stream reassembly, which has become a mandatory function of modern NIDS, all need to maintain a large data structure that holds the active network flows found in the monitored traffic at any given time. Thus, for packets belonging to the same network flow, the process accesses the same part of the data structure that corresponds to the particular flow.

In all above cases, we can identify a *locality* of executed instructions and data references for packets of the same type. In this work, we present a novel technique for improving packet processing performance by taking advantage of this locality property found in many passive monitoring applications. In practice, the captured packet stream is a mix of interleaved packets corresponding to hundreds or thousands of different packet types, depending on the monitored link. Our approach, called *locality buffering*, is based on adapting the packet stream in a way that enhances the locality of the application's code and memory access, and thus accelerating overall packet processing performance. Specifically, captured packets are not sent directly to the monitoring application, but instead are grouped in buffers according to their flow, and are sent in "batches"

We have implemented locality buffering in `libpcap` [36], the most widely used packet capturing library, which allows for improving the performance of a wide range of passive monitoring applications written on top of `libpcap` in a transparent way, without the need to alter their code. The experimental evaluation of our prototype implementation with real-world applications shows that locality buffering can significantly improve packet processing throughput and reduce the packet loss rate. For instance, the popular Snort IDS exhibited a 40% increase in the packet processing throughput, and a 60% improvement in packet loss rate. The design, implementation and experimental evaluation of the locality buffering technique are also described in [39].

4.1 Our Approach: Locality Buffering

The starting point of our work is the observation that several widely used passive network monitoring applications, such as intrusion detection systems, perform almost identical operations for a certain class of packets, while different packet classes result to the execution of different code paths and to data accesses to different memory locations. Such packet classes include the packets of a particular network flow, i.e., packets with the same protocol, source and destination IP addresses, and source and destination port numbers, or even wider classes such as all packets of the same application-level protocol, e.g., all HTTP, FTP, or BitTorrent packets.

Consider for example a NIDS like Snort [45]. Each arriving packet is first decoded according to its Layer 2–4 protocols, then it passes through several *preprocessors*, which perform various types of processing according to the packet type, and finally it is delivered to the main inspection engine, which checks the packet protocol headers and payload against a set of attack signatures. According to the packet type, different preprocessors may be triggered. For instance, IP packets go through the IP defragmentation preprocessor, which merges fragmented IP packets, TCP packets go through the TCP stream reassembly preprocessor, which reconstructs the bi-directional application level network stream, while HTTP packets go through the HTTP preprocessor, which decodes and normalizes the HTTP

FIGURE 4.1: The effect of locality buffering on the incoming packet stream.

protocol fields. Similarly, the inspection engine will check each packet only against a subset of the available attack signatures, according to its type. Thus, packets destined to a Web server will be checked against the subset of signatures tailored to Web attacks, FTP packets will be checked against FTP attack signatures, and so on.

When checking a newly arrived packet, the corresponding preprocessor(s) code, signature subset, and data structures will be fetched into the CPU cache. Since packets of many different types will likely be highly interleaved in the monitored traffic mix, different data structures and code will be constantly alternating in the cache, resulting to cache misses and reduced performance. The same effect occurs in other monitoring applications, such as NetFlow collectors or traffic classification applications, in which arriving packets are classified according to the network flow in which they belong to, which results to updates in a corresponding entry of a hash table. If many concurrent flows are active in the monitored link, their packets will arrive interleaved, and thus different portions of the hash table will be constantly being transferred in and out of the cache, resulting to poor performance.

The above observations motivated us to explore whether changing the order in which packets are delivered from the OS to the monitoring application improves packet processing performance. Specifically, we speculated that rearranging the captured traffic stream in such a way that packets of the same class are delivered to the application in "batches" would improve the locality of memory accesses, and thus reduce the overall cache miss ratio. This rearrangement can be conceptually achieved by buffering arriving packets into separate "buckets," one for each packet class, and emptying each bucket at once, either whenever it gets full, or after some predefined timeout since the arrival of the first packet of the bucket. For instance, if we assume that packets with the same destination port number correspond to the same class, then any interleaved packets destined to different network services will be rearranged so that packets to the same service are delivered back-to-back to the monitoring application, as depicted in Figure 4.1.

Choosing the destination port number as a class identifier strikes a good balance between the number of required buckets and the achieved locality. Indeed, choosing a more fine-grained classification scheme, such as a combination of the destination IP address and port number, would require a tremendous amount of buckets, and would probably just add overhead, since most of the applications of interest to this work perform (5-tuple) flow-based classification anyway. At the same time, packets destined to the same port usually correspond to the same application-level

Performance metric	Original trace	Sorted trace
Throughput (Mbit/sec)	188.39	286.18
Cache Misses (per packet)	18.86	2.79
Clock Cycles (per packet)	48,978.76	30,846.89

TABLE 4.1: Snort's performance using a sorted trace

protocol, so they will trigger the same Snort signatures and preprocessors, or will belong to the same or "neighbouring" entries in a network flow hash table.

4.2 Estimation of Feasibility

To get an estimation of the feasibility and the magnitude of improvement that locality buffering can offer, we performed a preliminary experiment whereby we sorted off-line the packets of a network trace based on the destination port number, and fed it to a passive monitoring application. This corresponds to applying locality buffering using buckets of infinite size. Details about the trace and the experimental environment are discussed in Section 5.2. We ran Snort [45] using both the sorted, as well as the original trace, and measured the processing throughput (trace size divided by the measured user plus system time), L2 cache misses, and CPU cycles of the application. Snort was configured with all the default preprocessors and signature sets enabled (2833 rules and 11 preprocessors). The L2 cache misses and CPU clock cycles were measured using the PAPI library [3], which utilizes the hardware performance counters.

Table 4.1 summarizes the results (each measurement was repeated 100 times, and we report the average values). We see that sorting results to a significant improvement of more than 50% in Snort's packet processing throughput, L2 cache misses are reduced by more than 6 times, and 40% less CPU cycles are consumed.

From the above experiment, we see that there is a significant potential of improvement in packet processing throughput using locality buffering. However, in practice, rearranging the packets of a continuous packet stream can only be done in short intervals, since we cannot indefinitely wait to gather an arbitrarily large number of packets of the same class before delivering them to the monitoring application—the captured packets have to be eventually delivered to the application within a short time interval (in our implementation, in the orders of milliseconds). Note that slightly relaxing the in-order delivery of the captured packets results to a delay between capturing the packet, and actually delivering it to the monitoring application. However, such a sub-second delay does not actually affect the correct operation of the monitoring applications that we consider in this work (delivering an alert or reporting a flow record a few milliseconds later is totally acceptable). Furthermore, packet timestamps are computed *before* locality buffering, and are not altered in any way, so any inter-packet time dependencies remain intact.

4.3 Implementation within libpcap

We have chosen to implement locality buffering within libpcap, the most widely used packet capturing library, which is the basis for a multitude of passive monitoring applications. Typically, applications read the captured packets through a call such as pcap_next, one at a time, in the same order as they arrive to the network interface. By incorporating locality buffering withing libpcap, monitoring applications continue to operate as before, taking advantage of locality buffering in a transparent way, without the need to alter their code or linking them with extra libraries. Indeed, the only difference is that consecutive calls to pcap_next or similar functions will most of the time return packets with the same destination port number, depending on the availability and the time constraints, instead of highly interleaved packets with different destination port numbers.

4.3.1 Periodic Packet Stream Sorting

In libpcap, whenever the application attempts to read a new packet, e.g., through a call to pcap_next, the library reads a packet from the kernel through a recv call, and delivers it to the application. That is, the packet is copied from kernel space to user space, in a small buffer equal to the maximum packet size, and then pcap_next returns a pointer to the beginning of the new packet.

So far, we have conceptually described locality buffering as a set of buckets, with packets with the same destination port ending up into the same bucket. One straightforward implementation of this approach would be to actually maintain a separate buffer for each bucket, and copy each arriving packet to its corresponding buffer. However, this has the drawback that an extra copy is required for storing each packet to the corresponding bucket, right after it has been fetched from the kernel through recv.

In order to avoid extra packet copy operations, which incur significant overhead, we have chosen an alternative approach. We distinguish between two different phases: the packet *gathering* phase, and the packet *delivery* phase. We have modified the single-packet-sized buffer of libpcap to hold a large number of packets, instead of just one. During the packet gathering phase, newly arrived packets are written sequentially into the buffer, by increasing the buffer offset in the recv call, until the buffer is full, or a certain timeout has expired.

Instead of arranging the packets into different buckets, which requires an extra copy operation for each packet, we maintain an indexing structure that specifies the order in which the packets in the buffer will be delivered to the application during the delivering phase. This indexing structure is illustrated in Figure 4.2. The index consists of a table with 64K entries, one for each port number. Each entry of the table points to the beginning of a linked list that holds references to all packets within the buffer with the particular destination port. In the packet delivery phase, traversing each list sequentially, starting from the first non-empty port number entry, allows for delivering the packets of the buffer ordered according

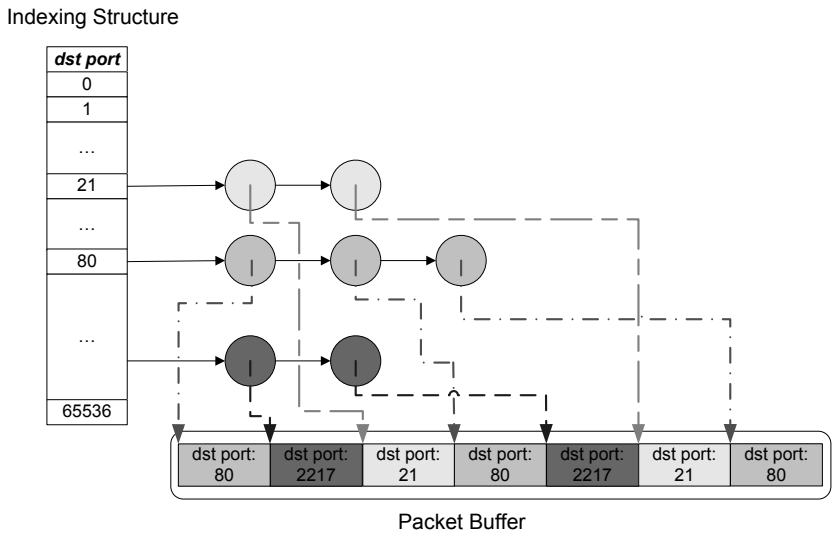

FIGURE 4.2: Using an indexing table with a linked list for each port, the packets are delivered to the application sorted by their destination port.

to their destination port. In this way we achieve the desired packet sorting, while, at the same time, all packets remain in place, in the initial memory location where they had been written by `recv`, avoiding extra costly copy operations. In the following, we discuss the two phases in more detail.

In the beginning of each packet gathering phase the indexing table is zeroed using `memset`. For each arriving packet, we perform a simple protocol decoding for determining whether it is a TCP or UDP packet, and consequently extract its destination port number. Then, a new reference for the packet is added to the corresponding linked list. For non-TCP or non-UDP packets, a reference is added into a separate list. The information that we keep for every packet in each node of the linked lists includes the packet's length, the precise timestamp of the time when the packet was captured, and a pointer to the actual packet data in the buffer.

Instead of dynamically allocating memory for new nodes in the linked lists, which would be an overkill, we pre-allocate a large enough number of spare nodes, equal to the maximum number of packets that can be stored in the buffer. Whenever a new reference has to be added to a linked list, a spare node is picked. Also, for fast insertion of new nodes at the end of the linked list, we keep a table with 64K pointers to the tail of each list.

The system continues to gather packets until the buffer becomes full, or a certain timeout has elapsed. The timeout ensures that if packets arrive with a low rate, the application will not wait too long for receiving the next batch of packets. We use 100ms as the default timeout in our prototype implementation, but both the timeout and the buffer size can be defined by the user. The buffer size and the time-

out are two significant parameters of our approach, since they influence the number of sorted packets that can be delivered to the application in each batch. Depending on how intensive each application is, this number of packets determines the benefit in its performance. In Section 5.2 we examine the effect that the number of packets in each batch has on overall performance using three different passive monitoring applications.

Upon the end of the packet gathering phase, packets can be delivered to the application following the order imposed from the indexing structure. For that purpose, we keep a pointer to the list node of the most recently delivered packet. Starting from the beginning of the index table, whenever the application requests a new packet, e.g., through `pcap_next`, we return the packet pointed either by the next node in the list, or, if we have reached the end of the list, by the first node of the next non-empty list. The latter happens when all the packets of the same destination port have been delivered (i.e., the bucket has been emptied), so conceptually the system continues with the next non-empty group.

4.3.2 Using a Separate Thread for Packet Gathering

A drawback of the above implementation is that during the packet gathering phase, the CPU remains idle most of the time, since no packets are delivered to the application for processing in the meanwhile. Reversely, during the processing of the packets that were captured in the previous packet gathering period, no packets are stored in the buffer. In case that the kernel's socket buffer is small and the processing time for the current batch of packets is increased, it is possible that a significant number of packets may get lost by the application, in case of high traffic load.

Although in practice this effect does not degrade performance due to the very short timeouts used (e.g. 100ms), as we show in Section 5.2, we can improve further the performance of locality buffering by employing a separate thread for the packet gathering phase, combined with the usage of two buffers instead of a single one. The separate packet gathering thread receives the packets from the kernel and stores them to the *write buffer*, and also updates its index. In parallel, the application receives packets for processing from the main thread of `libpcap`, which returns the already sorted packets of the second *read buffer*. Each buffer has its own indexing table.

Upon the completion of both the packet gathering phase, i.e., after the timeout expires or when the write buffer becomes full, and the parallel packet delivery phase, the two buffers are swapped. The write buffer, which now is full of packets, turns to a read buffer, while the now empty read buffer becomes a write buffer. The whole swapping process is as simple as swapping two pointers, while semaphore operations ensure the thread-safe exchange of the two buffers.

5 Experimental Evaluation

5.1 DiMAPI Network-Level Performance

In this section we experimentally evaluate several performance aspects of DiMAPI. Our analysis consists of measurements regarding the network overhead and response latency, and how these metrics scale as the number of the participating monitoring sensors increases.

5.1.1 Experimental Environment

For the experimental evaluation of DiMAPI we used two different monitoring sensor deployments. The first deployment consists of 15 monitoring sensors distributed inside the internal network of FORTH. All sensors are interconnected through 100 Mbit/sec Ethernet for the control interface. Each sensor is equipped with a second Ethernet interface for the actual passive network monitoring. The monitored test traffic is generated by replaying a real network traffic trace using `tcpreplay` [4]. The second deployment consists of four monitoring sensors located at four different ASes across the Internet: FORTH, the University of Crete (UoC), the Venizelio Hospital at Heraklion (VHosp), and the University of Pennsylvania (UPenn). In this deployment, each sensor monitors live traffic passing through the monitored links of the corresponding organization.

5.1.2 Network Overhead

As discussed in Section 2.3, whenever a monitoring application that utilizes remote sensors calls a DiMAPI function, this results to a message exchange between the DiMAPI stub and the `mapicommd` running on each sensor. This procedure poses

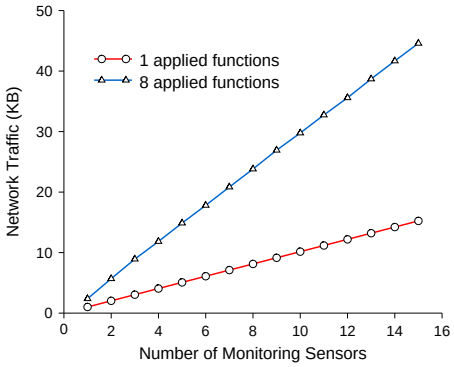

FIGURE 5.1: Total network traffic exchanged during the initialization phase, when applying 1 and 8 functions

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 test monitoring application that creates a network flow, configures it by applying several functions, and then periodically reports some results according to the applied functions. The measurements were performed in the 15-sensor FORTH network, while the test application was running on a separate host. Our target is to measure the network overhead generated by DiMAPI, when using different monitoring granularity. The generated network traffic was measured using a second local MAPI 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.

In the first experiment, we measured the network overhead for 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 incurred by both the DiMAPI stub and `mapicommd` during the creation, configuration, and instantiation of a network flow. Figure 5.1 shows the amount of traffic generated during the initialization phase for two variations of the test application, the first applying only one function, which results to a total of three DiMAPI library function calls for the initialization phase, and the second 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 experiments we measured the rate of the network traffic incurred during the lifetime of the application due to the periodic results retrieval. After

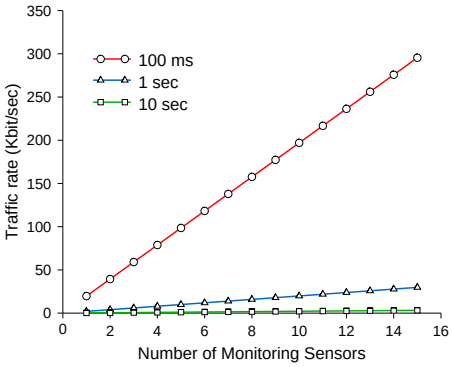

FIGURE 5.2: Network overhead incurred using the function BYTE_COUNTER with polling periods 0.1, 1, and 10 seconds

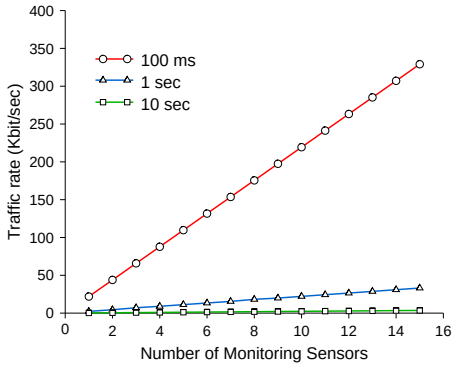

FIGURE 5.3: Network overhead incurred using the function HASHSAMP with polling periods 0.1, 1, and 10 seconds

the initialization phase, the test application constantly reads the new value of the result 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. Figure 5.2 shows the results when applying the BYTE_COUNTER function that returns an unsigned 8-byte integer, while Figure 5.3 shows the results when applying the HASHSAMP function, that returns a significantly larger data structure. HASHSAMP is used to perform hash-based sampling on the packets of a network flow, and its results format is a 36-byte data structure.

In case that the application with BYTE_COUNTER reads the result in 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/sec, 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. When reading the results of HASHSAMP, we see 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%.

5.1.3 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

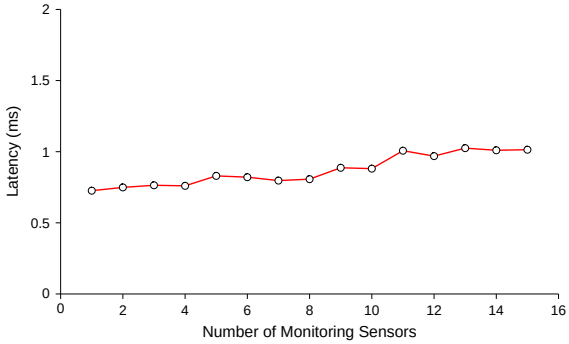

FIGURE 5.4: Completion time for `mapi_read_results()`

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 stub should be negligible, and thus the overall latency should be close to the maximum RTT of the sensors within the flow's scope.

We measured the time for retrieving results by calling `mapi_read_results()` call, using a test application which applies the `BYTE_COUNTER` function and running it in the FORTH network. The time was measured by generating two timestamps 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 5.4 presents the completion time for the execution of a `mapi_read_results()` call as a function of the number of monitoring sensors in the network flow scope. As the number of sensors increases, there is a very 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. 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 monitoring sensor varies considerably.

We report our findings in Table 5.1. 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 results from each monitoring sensor. The results of Table 5.1 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.

Network Flow Scope	`mapi_read_results()` delay (ms)	Network RTT (ms)
VHosp	170.58	160.69
UoC	3.26	3.24
FORTH	0.68	0.67
UPenn	283.65	279.22
VHosp, UoC, FORTH, UPenn	285.496	-

TABLE 5.1: Comparison between the completion time of `mapi_read_results()` and the network Round Trip Time

In order to achieve even lower response latency in getting the monitoring results, that will not depend on network's RTT, we can use the push model, as described in section 2.3.4. The experiments of the next section are focused to examine improvements using the push model in fetching packets to the application. Similar improvements can be achieved in the latency of `mapi_read_results` when using this technique.

5.1.4 Evaluation of Packet Prefetching

In the next experiments we examine the improvement that the packet prefetching approach achieves in DiMAPI, using the push model as described in section 2.3.4. Instead of requesting from a remote `mapicommd` one packet each time the application calls the `mapi_get_next_pkt()`, the DiMAPI stub transparently receives a number of packets that `mapicommd` sends back-to-back, stores them in a buffer and returns them from this buffer to the application in each `mapi_get_next_pkt()` call.

For our experiments we used three different computers. The first one is the passive monitoring sensor where `mapid` and `mapicommd` daemons run. Another computer is used for generating traffic using the `tcpreplay` [4] tool, by sending several times and at different rates a network packet trace of 1 GB size with real network traffic captured from a passive monitor located at the School Network of Crete. Finally, a third machine is used for running DiMAPI test applications. These computers are all interconnected through a local 1 Gbit/sec switch.

The test applications call the `mapi_get_next_pkt()` function using both pull and push model implementations. In order to compare their performance, we measure the completion time of a `mapi_get_next_pkt()` call by placing `gettimeofday()` calls before and after `mapi_get_next_pkt()`. Moreover, we count the number of packets and the total bytes that we are able to deliver in the application calling `mapi_get_next_pkt()` within one second interval using pull and push models. We run the test applications for 10 minutes and report the average throughput in Mbit/sec as computed from the packet's size in bytes.

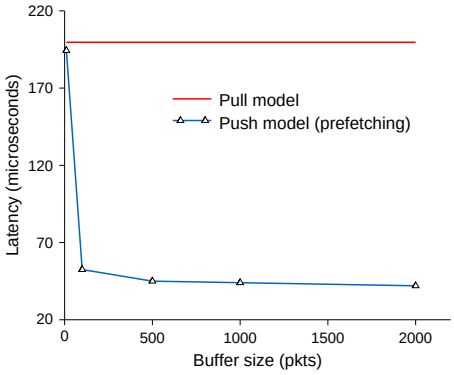

FIGURE 5.5: Completion time for *mapi_get_next_pkt()* with pull and push models for different buffer sizes while replaying at 100 Mbit/sec

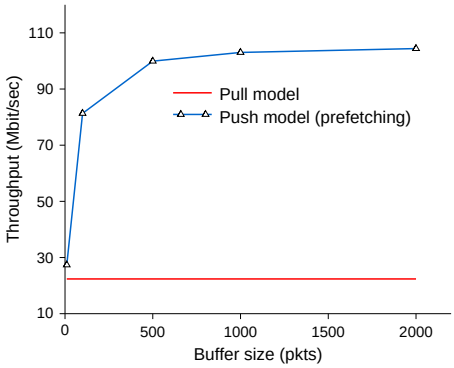

FIGURE 5.6: Throughput in Mbit/sec for *mapi_get_next_pkt()* with pull and push models for different buffer sizes while replaying at 100 Mbit/sec

Examine the Effect of Buffer Size

Firstly we examine the effect that the size of the buffer which holds the packets has on the performance. This size is equal to how many packets will be prefetched to the application's stub in one batch. Figures 5.5 and 5.6 present the latency and throughput respectively for `mapi_get_next_pkt()` while replaying the trace at the constant rate of 100 Mbit/sec and varying the size of the buffer from 10 to 2000 packets.

The results show that prefetching significantly reduces the latency of a `mapi_get_next_pkt()` call and increases the number of packets that can be delivered to the application in one second. The delay drops from 200 to 52 microseconds in case of 100 packets buffer size and to 42 microseconds when using 2000 packets buffer, that comprises an improvement of 3.85 and 4.76 times respectively. Throughput increases from 22.4 to 81.3 and 99.9 Mbit/sec in case of buffer sizes of 100 and 500 packets respectively. This means that when using buffer larger that 500 packets in size, in our setup, we can achieve to forward all the network packets from `mapicommd` to the application with the same rate that they reach at the monitoring interface.

When increasing the buffer size from 500 to 2000 packets we observe only a slight improvement in latency and throughput. So, we consider 500 packets as a good enough buffer size. If compared to, e.g., buffer size of 2000 packets, the application's stub will need to perform one request every 500 packets instead every 2000 packets that means just 4 more requests every 2000 packets.

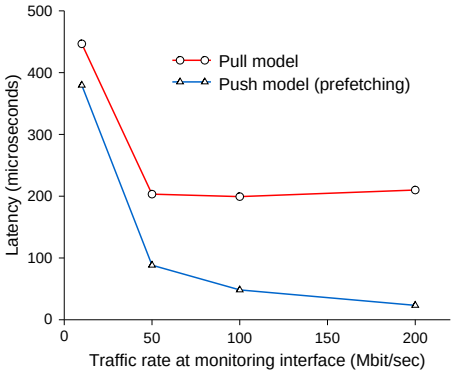

FIGURE 5.7: Completion time for `mapi_get_next_pkt()` with pull and push models while replaying a trace from 10 to 200 Mbit/sec

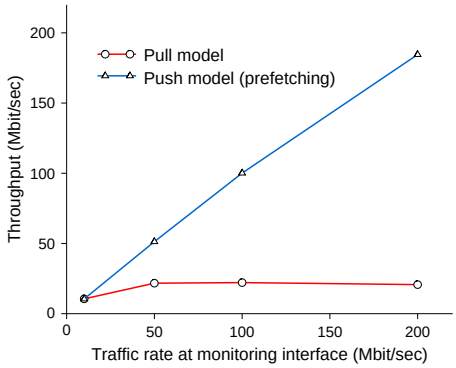

FIGURE 5.8: Throughput in Mbit/sec for `mapi_get_next_pkt()` with pull and push models while replaying a trace from 10 to 200 Mbit/sec

Examine the Effect of Traffic Rate

Next, we vary the traffic generation rate from 10 to 200 Mbit/sec while using 500 packets for the size of the buffer that packets are stored in the push model case. Figures 5.7 and 5.8 presents the results.

We observe that increasing the rate of the generated traffic results to a reduction on the average latency for a `mapi_get_next_pkt()` call and to an increment of the throughput for both pull and push models. This is reasonable since `mapid` and `mapicommd` daemons do not block waiting for new packets to arrive in the monitoring device, which is more possible to happen at low traffic rates. However, in the pull model we can see that this effect is visible only when the rate is increased from 10 to 50 Mbit/sec. After 50 Mbit/sec, the delay and throughput of `mapi_read_results()` remains always constant at about 200 microseconds and 22 Mbit/sec respectively. The latency is limited to 200 microseconds due to the network's RTT, while the maximum throughput that can be achieved for pull model is 22 Mbit/sec. So, at higher rates than 22 Mbit/sec a significant amount of packets will be dropped from the buffer that `mapid` saves them and will be lost.

On the other hand, using the push model we are not limited from the network's RTT since the packets are sent back-to-back. The throughput that we can achieve using the push model for fetching packets approaches the traffic rate on the monitoring interface. For example, for 100 Mbit/sec traffic rate at the monitoring interface, the `mapicommd` sends packets to the application with 100 Mbit/sec also, while for 200 Mbit/sec traffic rate `mapicommd` achieves throughput of 185 Mbit/sec. Comparing the two different approaches, the push model is 4 times faster at 100 Mbit/sec and 9 times faster at 200 Mbit/sec from the pull model. Using the push model, `mapicommd` can send up to 185 Mbit/sec without loosing any packet, while the pull model can transfer packets with rate only up to 22 Mbit/sec.

5.2 Locality Buffering Performance Evaluation

In this section, we present the experimental evaluation of our prototype implementation of locality buffering. We deploy the modified versions of `libpcap` to three popular passive monitoring applications: `Snort` intrusion detection system, `Appmon` application for accurate traffic classification and Fprobe flow export tool. Then we compare their performance using the original `libpcap` and our locality buffering implementations.

5.2.1 Experimental Environment

Our experimental environment consists of two PCs interconnected through a Gigabit switch. The first PC is used for traffic generation, which is achieved by replaying real network traffic traces at different rates using `tcpreplay` [4]. We used a full payload trace captured at the access link that connects an educational network with thousands of hosts to the Internet. The trace contains 1,698,902 packets, corresponding to 64,628 different network flows, totalling more than 1 GB in size.

By rewriting the source and destination MAC addresses in all packets, the generated traffic can be sent to the second PC, the passive monitoring sensor, which captures the traffic and processes it using different monitoring applications. The passive monitoring sensor is equipped with an Intel Xeon 2.40 GHz processor with 512 KB L2 cache and 512 MB RAM running Debian Linux (kernel version 2.6.18). The kernel socket buffer size was set to 16 MB, in order to minimize packet loss due to packet bursts.

We tested the performance of the monitoring applications on top of three different versions of `libpcap`: the original version, our modified version that employs locality buffering, and a third version with the optimized locality buffering approach that uses a separate thread for storing incoming packets. For each setting, we measured the application's user and system time using the UNIX `time` utility. Also, the idle CPU time is computed from the average percentage of the CPU usage that the application's process has taken. Furthermore, we measured the L2 cache misses and the CPU clock cycles by reading the CPU performance counters through the PAPI library [3]. Finally, an important metric that was measured is the percentage of packets being dropped by `libpcap`, which usually happens when replaying the traffic in high rates, due to high CPU utilization.

Traffic generation begins after the application has been initiated. The application is terminated immediately after capturing the last packet of the replayed trace. All measurements were repeated 10 times, and we report the average values. We focus mostly on the discussion of our experiments using Snort, which is the most resource-intensive among the tested applications. However, we also briefly report our experiences with Fprobe and Appmon.

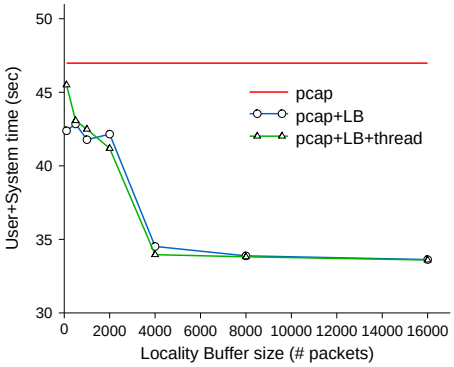

FIGURE 5.9: Snort's user plus system time as a function of the buffer size for 100 Mbit/s traffic.

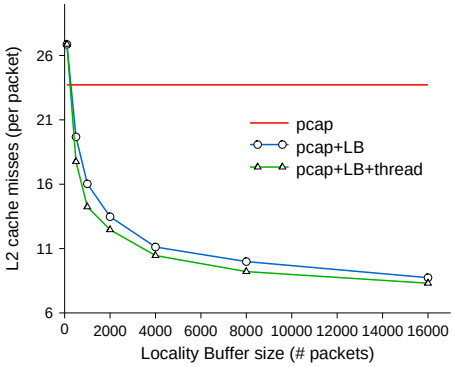

FIGURE 5.10: Snort's L2 cache misses as a function of the buffer size for 100 Mbit/s traffic.

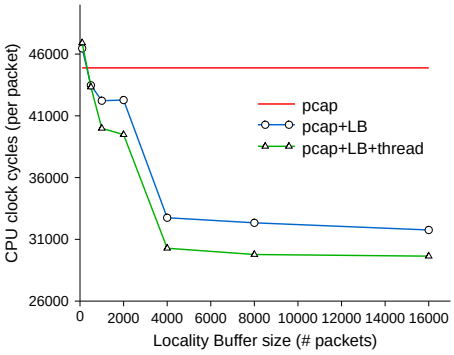

FIGURE 5.11: Snort's CPU cycles as a function of the buffer size for 100 Mbit/s traffic.

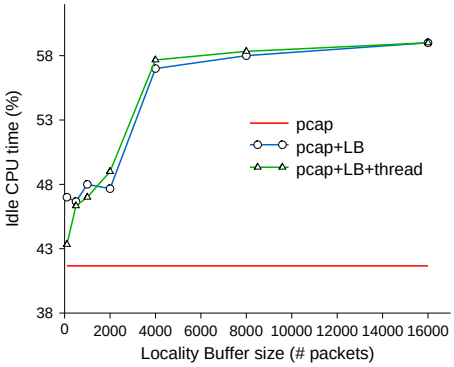

FIGURE 5.12: Idle CPU time as a function of the buffer size for 100 Mbit/s traffic.

5.2.2 Results from Snort

We ran Snort using its default configuration, in which almost all of the available rule sets and preprocessors are enabled. Snort loaded 2833 rules, while 11 preprocessors were active.

Initially, we examine the effect that the size of the buffer in which the packets are sorted has on the overall application performance. We vary the size of the buffer from 100 to 16000 packets while replaying the network trace at a constant rate of 100 Mbit/sec. Using a 100 Mbit/sec rate, no packets were dropped. We do not use any timeout in these experiments for packet gathering. As long as we send traffic at constant rate, the buffer size determines how long the packet gathering phase will last. Figure 5.9 shows the user plus system time of Snort for processing the replayed traffic using the different `libpcap` versions. Figures 5.10

and 5.11 present the per-packet L2 cache misses and clock cycles respectively, while Figure 5.12 presents the average idle CPU time while Snort was running.

We observe that increasing the size of the buffer results to lower user time, fewer cache misses and clock cycles, and generally to an overall performance improvement. This is because using a larger packet buffer offers better possibilities for effective packet sorting, and thus to better memory locality. However, increasing the size from 4000 to 16000 packets gives only a slight improvement. Based on this result, we consider 4000 packets as optimum buffer size in our experiments. For a rate of 100 Mbit/sec, 4000 packets roughly correspond to an 160 millisecond period at average.

We can also notice that using locality buffering we achieve a significant reduction on the L2 cache misses from 23.7 per packet to 10.5, when using a 4000 packets buffer, which is an improvement of 2.26 times against Snort with the original `libpcap` library. Also, Snort's user time and clock cycles are significantly reduced, making it faster by more than 40%. Due to the improved memory accessing locality, the CPU remains idle for a significantly larger percentage of time.

Comparing our two different implementations, they result to similar performance in all the metrics measured. The modified version of `libpcap` that uses a separate thread for storing packets to the buffer seems to perform slightly better than the simple implementation.

We replayed the trace in different rates, from 10 to 300 Mbit/sec, trying different buffer sizes as before for each rate and we concluded to the same findings. In all rates, 4000 packets was found as the optimum buffer size. Using this optimum buffer size, locality buffering results in all rates to a significant reduction on Snort's cache misses and user time, similar to the improvement observed in 100 Mbit/sec against the original `libpcap`. The two implementations have almost equal performance in all cases, with the one using a thread performing a little better.

Another important metric for evaluating the improvement of our technique is the percentage of the packets that are being dropped in high rates by the kernel because Snort is not able to process all of the them in these rates. In Figure 5.13 we plot the average percentage of packets that are being dropped while replaying the trace with speeds ranging from 10 to 300 Mbit/sec. We used 4000 packets size for the locality buffer, which was found to be the optimal size for Snort when replaying this traffic at any rate.

Using the unmodified `libpcap`, Snort cannot process all packets in rates higher than 125 Mbit/sec, so a significant percentage of packets is being lost. On the other hand, using locality buffering, the packet processing time is accelerated and the system is able to process more packets in the same time interval. As shown in Figure 5.13, when deploying our locality buffering implementations in Snort, it becomes much more resistant in packet loss. It begins to loose packets at 200 Mbit/sec instead of 125 Mbit/sec, which is a 60% improvement. Also, at 250 Mbit/sec, our implementation drops 2.6 times less packets than the original `libpcap`. The two different implementations of the locality buffering technique

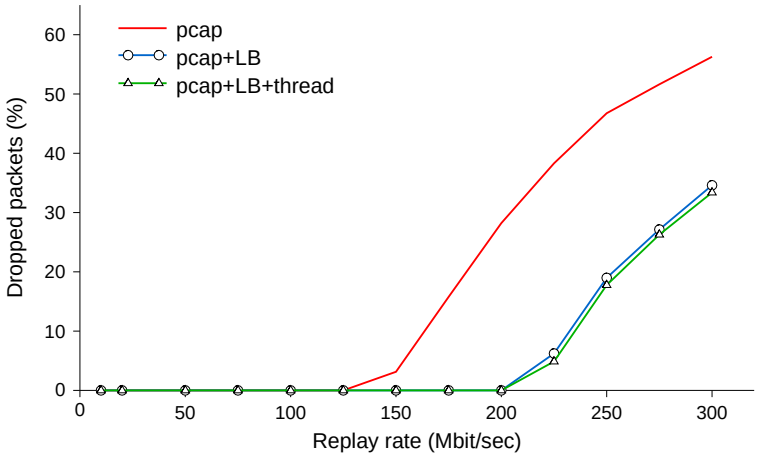

FIGURE 5.13: Packet loss ratio of the passive monitoring sensor when running Snort, as a function of the traffic speed.

achieve almost the same performance, with the thread-based implementation having slightly less dropped packets.

We do not observe any significant improvement with the thread-based implementation, compared to the simple locality buffering implementation, because the major benefit of our technique is the acceleration of packet processing due to improving memory access locality. Moreover, in the constant and high traffic rates that we generated in our experiments, the CPU time was not idle during the packet gathering phase, since packets were continuously arriving. In case of bursty traffic, however, the separate thread would be more resistant to dropping packets.

5.2.3 Results from Appmon

`Appmon` [9] is a passive network monitoring tool for accurate per-application traffic identification and categorization. It uses deep-packet inspection and packet filtering for attributing flows to the applications that generate them. We ran Appmon on top of our modified versions of `libpcap` and examined the improvement that they can offer using different buffer sizes that vary from 100 to 16000 packets. Figure 5.14 presents the Appmon's user plus system time and Figure 5.15 the per-packet L2 cache misses measured while replaying the trace at a constant rate of 100 Mbit/sec.

The results show that the Appmon's performance can be improved using the locality buffering implementations. Its cache misses are reduced from 8.4 to 7.1

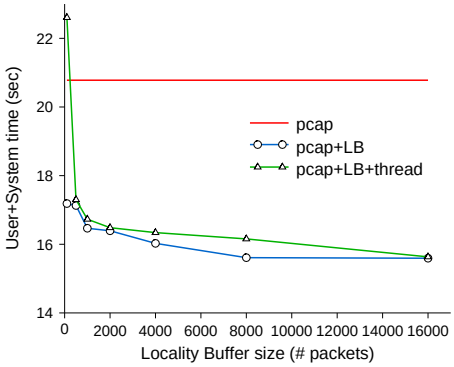

FIGURE 5.14: Appmon's user plus system time as a function of the buffer size for 100 Mbit/s traffic.

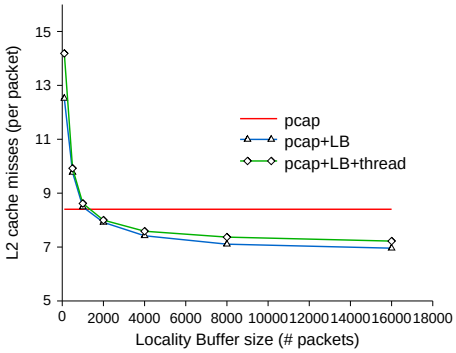

FIGURE 5.15: Appmon's L2 cache misses as a function of the buffer size for 100 Mbit/s traffic.

misses per packet, when used buffer size of 8000 packets, that is a 18% improvement. Thus, the user plus system time is reduced by more than 30% compared width the original `libpcap`. The optimum buffer size in the case of Appmon, based on the these results, seems to be around 8000 packets. Our different implementations resulted again to very close performance, with the first one giving a little better results this time.

We were also running Appmon when replaying traffic in rates varying from 10 to 300 Mbit/sec, observing always similar results. Since Appmon does significantly less processing than snort, no packets were dropped in these rates. The output of Appmon remains identical in all cases, which means that the periodic packet stream sorting does not affect the correct operation of Appmon's classification process.

5.2.4 Results from Fprobe

`Fprobe` [1] is a passive monitoring application that collects traffic statistics for each active flow and exports corresponding NetFlow records. We ran Fprobe with our modified versions of `libpcap` and performed the same measurements as with Appmon. Figure 5.16 plots the user plus system time of the Fprobe variants per buffer sizes from 100 up to 16000 packets, while replaying the trace at 100 Mbit/sec rate.

We notice a speedup of about 30% when locality buffering is enabled. The buffer size that optimizes overall performance is again around 8000 packets. We notice that in Appmon and Fprobe tools the optimum buffer size is about 8000 packets, while in Snort 4000 packets size is enough to optimize the performance. This happens because Appmon and Fprobe are not so CPU-intensive as Snort, so they require a larger amount of packets to be sorted in order to achieve a significant performance improvement. Finally, we observe that the version of `libpcap` that

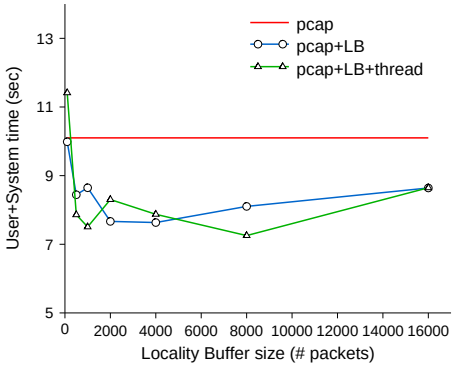

FIGURE 5.16: Fprobe's user plus system time as a function of the buffer size for 100 Mbit/s traffic.

uses a separate thread for storing packets gives better performance in Fprobe for some of the buffer sizes, but it is not clear which of these two versions is preferable in this case. Similar results were observed in all rates of the replayed traffic.

6
Related Work

6.1 Passive Network Monitoring Tools and Libraries

There are several techniques and tools currently available for passive network monitoring, which can be broadly categorized into three categories [26]: passive packet capturing, flow-level measurements, and aggregate traffic statistics. These categories are with decreasing order regarding the offered functionality and complexity. For example, flow-level measurements and aggregate traffic statistics can be provided by packet capturing systems. DiMAPI belongs to the first category, since it is capable to perform distributed packet capture and manage remote monitoring sensors, but can also offer the latter functionalities by applying the appropriate functions to the network flows.

The most widely used library for packet capturing is `libpcap` [36], which provides a portable API for user-level packet capture. The `libpcap` interface supports a filtering mechanism based on the BSD Packet Filter [35], which allows for selective packet capture based on packet header fields. The `libpcap` library has been widely used in several passive monitoring applications such as packet capturing [44, 48], network statistics monitoring [17], flow export [1, 16] and intrusion detection systems [45]. Also, DiMAPI is implemented on top of `libpcap` for commodity network interfaces. Thus, performance optimizations like our locality buffering technique, which has been implemented within `libpcap`, can be beneficial for all the above tools and libraries.

`WinPcap` [5] and `rpcap` [32] extend `libpcap` with remote packet capturing capabilities. Both allow the transfer of captured packets at a single remote host to a local host for further processing. DiMAPI offers the same and more functionality through the scope abstraction for *multiple* distributed monitoring sensors, being also much more expressive. Furthermore, by enabling traffic processing at

each remote sensor, DiMAPI avoids the considerable network overhead of above approaches since it sends back only the computed results.

CoralReef provides a set of tools and support functions for capturing and analyzing network traces [30]. `libcoral` provides an API for monitoring applications that is independent of the underlying monitoring hardware. Nprobe [38] 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

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. 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` [16] offer NetFlow record generation by capturing packets using commodity hardware. DiMAPI shares some goals with the above flow-based monitoring systems, but it has significantly more functionality.

6.2 Distributed Passive Network Monitoring Infrastructures

As network traffic monitoring is becoming increasingly important for the operation of modern networks, several passive monitoring infrastructures have been proposed.

CoMo [27] is a passive monitoring infrastructure which allows users to query network data gathered from multiple administrative domains, by providing a number of generic query mechanisms. It is based on a number of distributed monitoring nodes, consisting of the CoMo core processes and a number of user defined plug-in modules. Each one of these nodes is able to answer queries based on the modules that are plugged-in.

A similar approach is followed by Gigascope [15] that is a stream database for storing captured network data in a central repository for further analysis using the GSQL query language. Users are able to implement special query operators by following a specific API. Gigascope is able to satisfy fast simple network monitoring needs by serving user"s SQL-like queries, from a database that is created from a single monitoring sensor.

Sprint's passive monitoring system [24] was installed within the Sprint IP backbone network and it was collecting data from different monitoring points into a central repository for further analysis. However, it could not support many different monitoring applications, and it is not a scalable approach as this system was installed effectively just within the Sprint"s backbone network.

All the above infrastructures are mainly based on databases with predefined custom schemas which collect data from distributed sensors and accept queries using SQL-like languages from monitoring applications. In order to implement

new functionality, new plugins must be written and embedded in the monitoring sensors. If compared to DiMAPI, they do not support any of the concepts of network flow or network scope, and none of these systems provides any API which will aid the developer to create novel distributed monitoring applications. DiMAPI adopts a different approach, providing an API for distributed passive monitoring applications development instead of supplying with a database for data queries.

Arlos et al. [10] propose DPMI, a distributed passive measurement infrastructure that supports various monitoring equipment within the same administrative domain. DPMI defines the means of creating a testbed that will provide passive monitoring capabilities from a number of predefined measurement points to data consumers.

Finally, a lot of work is being done in the area of monitoring of high performance computing systems, such as clusters and Grids. Ganglia [34] is a distributed monitoring system based on a hierarchical design targeted at federations of clusters. GridICE [8] is a distributed monitoring tool integrated with local monitoring systems with a standard interface for publishing monitoring data. These systems could utilize at lower levels the functionality offered by DiMAPI.

6.3 Locality Buffering

The concept of locality buffering for improving passive network monitoring applications, and, in particular, intrusion detection and prevention systems, was first introduced by Xinidis et al. [53], as part of a load balancing traffic splitter for multiple network intrusion detection sensors that operate in parallel. In this work, the load balancer splits the traffic to multiple intrusion detection sensors, so that similar packets (e.g. packets destined to the same port) are processed by the same sensor. However, in this approach the splitter uses a limited number of locality buffers and copies each packet to the appropriate buffer based on hashing on its destination port number. Our approach differs in two major aspects. First, we have implemented locality buffering within a packet capturing library, instead of a separate network element. To the best of our knowledge, our prototype implementation within the libpcap library is the first attempt for providing memory locality enhancements for accelerating packet processing in a generic and transparent way for existing passive monitoring applications. Second, the major improvement of our approach is that packets are not actually copied into separate locality buffers. Instead, we maintain a separate index which allows for scaling the number of locality buffers up to 64K.

Locality enhancing techniques for improving server performance have been widely studied. For instance, Markatos et al. [33] present techniques for improving request locality on a Web cache, which results to significant improvements in the file system performance.

6.4 Improving the Performance of Packet Capturing

Several research efforts [18, 19, 51] have focused on improving the performance of packet capturing through kernel and library modifications which reduce the number of memory copies required for delivering a packet to the application. In contrast, our approach with locality buffering technique aims to improve the packet processing performance of the monitoring application itself, by exploiting the inherent locality of the in-memory workload of the application.

7 Conclusion

In this thesis, we presented the design, implementation and performance evaluation of DiMAPI, a flexible and expressive API for building distributed passive network monitoring applications. One of the main novelties of DiMAPI is the introduction of the network flow *scope*, a new attribute of network flows which enables the creation and manipulation of flows over a set of local and remote passive monitoring sensors. The design of DiMAPI mainly focuses on minimizing performance overheads, while providing extensive functionality for a broad range of distributed monitoring applications.

We have evaluated the performance of DiMAPI using a number of monitoring applications operating over large monitoring sensor sets, as well as highly distributed environments. Our results showed that DiMAPI has low network 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. Furthermore, using result and packet prefetching (push model) we can achieve even lower response times, since we are not limited from the network's round trip time. For instance, we showed that when sending batches of 500 captured network packets back-to-back from a monitoring sensor to a monitoring application, it can continue sending captured packets up to 185 Mbit/sec, in a Gigabit network, without loosing any packet. On the other hand, the first implementation (pull model) can transfer packets with rate only up to 22 Mbit/sec.

We also presented a novel distributed passive monitoring technique for real time packet loss estimation between different domains. The technique is based on tracking the *expired flows* at each monitoring sensor. Using DiMAPI as distributing monitoring infrastructure, a central monitoring application correlates the results from the monitoring sensors and computes the actual packet loss ratio. Our passive monitoring approach for packet loss estimation is accurate and reliable, while at

the same time exhibits inherent advantages such as scalability and a non-intrusive nature. This is a typical application relying on the basic DiMAPI functionality.

Moreover, we introduced locality buffering, a technique for improving the performance of packet processing in a wide class of passive network monitoring applications by enhancing the locality of memory access. Our approach is based on reordering the captured packets before delivering them to the monitoring application, by grouping together packets with the same destination port. This results to improved locality for code and data accesses, and consequently to an increase in the packet processing throughput and to a decrease in the packet loss rate.

We described in detail the design and the implementation of locality buffering within the widely used `libpcap` library, and presented our experimental evaluation using three representative CPU-intensive passive monitoring applications. The evaluation results showed that all applications gain a significant performance improvement, while the system can keep up with higher traffic speeds without dropping packets. Specifically, locality buffering resulted to a 40% increase in the processing throughput of the Snort IDS, while the packet loss rate was decreased by 60%. Using the original `libpcap` implementation, the Snort sensor begins loosing packets when the monitored traffic speed reaches 125 Mbit/sec, while using locality buffering, packet loss is exhibited when exceeding 200 Mbit/sec. Fprobe, a NetFlow export probe, and Appmon, an accurate traffic classification application, also exhibited a significant throughput improvement, up to 30%, even though they do not perform as CPU-intensive processing as Snort.

Overall, we believe that implementing locality buffering within `libpcap` is an attractive performance optimization, since it offers significant performance improvements to a wide range of passive monitoring applications, while at the same time its operation is completely transparent, without needing to modify existing applications. DiMAPI implementation for commodity network interfaces is also based on `libpcap`, so DiMAPI based applications can benefit indirectly from locality buffering.

Bibliography

[1] fprobe: Netflow probes. http://fprobe.sourceforge.net/.

[2] MAPI Public Release. `http://mapi.uninett.no`.

[3] Performance application programming interface. http://icl.cs.utk.edu/papi/.

[4] Tcpreplay. http://tcpreplay.synfin.net/trac/.

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

[6] A. Adamns, J. Mahdavi, M. Mathis, and V. Paxson. Creating a scalable architecture for internet measurement. *IEEE Network*, 1998.

[7] M. Allman, W. M. Eddy, and S. Ostermann. Estimating loss rates with tcp. *ACM Performance Evaluation Review*, 31(3), December 2003.

[8] S. Andreozzi, N. D. Bortoli, S. Fantinel, A. Ghiselli, G. Rubini, G. Tortone, and M. Vistoli. GridICE: a Monitoring Service for Grid Systems. *Future Generation Computer Systems Journal*, 21(4):559–571, Apr. 2005.

[9] D. Antoniades, M. Polychronakis, S. Antonatos, E. P. Markatos, S. Ubik, and A. Oslebo. Appmon: An application for accurate per application traffic characterization. In *Proceedings of IST Broadband Europe 2006 Conference*, December 2006.

[10] 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.

[11] P. Benko and A. Veres. A passive method for estimating end-to-end tcp packet loss. In *Proceedings of IEEE Globecom*, 2002.

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

[13] A. Ciuffoletti, A. Papadogiannakis, and M. Polychronakis. Network monitoring session description. In *Proceedings of the CoreGRID Workshop on Grid Middleware (in conjunction with ISC '07)*, June 2007.

[14] A. Ciuffoletti and M. Polychronakis. Architecture of a network monitoring element. Technical Report TR-0033, CoreGRID Project, February 2006.

[15] 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.

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

[17] L. Deri. ntop. `http://www.ntop.org/`.

[18] L. Deri. Improving passive packet capture:beyond device polling. In *Proceedings of SANE*, 2004.

[19] 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.

[20] N. G. Duffield and M. Grossglauser. Trajectory Sampling for Direct Traffic Observation. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 271–282, 2000.

[21] N. G. Duffield, F. L. Presti, V. Paxson, and D. F. Towsley. Inferring link loss using striped unicast probes. In *INFOCOM*, pages 915–923, 2001.

[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. `http://www.endace.com/`.

[24] 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.

[25] Y. Fu, L. Cherkasova, W. Tang, and A. Vahdat. EtE: Passive end-to-end Internet service performance monitoring. In *Proceedings of the USENIX Annual Technical Conference*, pages 115–130, 2002.

[26] M. Grossglauser and J. Rexford. Passive traffic measurement for IP operations. In *The Internet as a Large-Scale Complex System*, pages 91–120. 2005.

[27] 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`.

[28] Intel Corporation. Intel IXP1200 Network Processor white paper, 2000.

[29] H. Jiang and C. Dovrolis. Passive estimation of tcp round-trip times. *SIGCOMM Comput. Commun. Rev.*, 32(3):75–88, 2002.

[30] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, and K. Claffy. The architecture of CoralReef: an Internet traffic monitoring software suite. In *Proceedings of the 2nd International Passive and Active Network Measurement Workshop*, Apr. 2001.

[31] D. Koukis, S. Antonatos, D. Antoniades, E. P. Markatos, and P. Trimintzios. A generic anonymization framework for network traffic. In *Proceedings of the IEEE International Conference on Communications (ICC)*, volume 5, June 2006.

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

[33] E. P. Markatos, D. N. Pnevmatikatos, M. D. Flouris, and M. G. H. Katevenis. Web-conscious storage management for web proxies. *IEEE/ACM Trans. Netw.*, 10(6):735–748, 2002.

[34] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7), July 2004.

[35] 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.

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

[37] L. Michael and G. Lior. The effect of packet reordering in a backbone link on application throughput. *Network, IEEE*, 16(5):28–36, 2002.

[38] A. Moore, J. Hall, E. Harris, C. Kreibich, and I. Pratt. Architecture of a network monitor. In *Proceedings of the 4th International Passive and Active Network Measurement Workshop*, April 2003.

[39] A. Papadogiannakis, D. Antoniades, M. Polychronakis, and E. P. Markatos. Improving the performance of passive network monitoring applications using locality buffering. In *Proceedings of the 15th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, October 2007.

[40] A. Papadogiannakis, A. Kapravelos, M. Polychronakis, E. P. Markatos, and A. Ciuffoletti. Passive end-to-end packet loss estimation for grid traffic monitoring. In *Proceedings of the CoreGRID Integration Workshop*, 2006.

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

[42] 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 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 483–496, April 2004.

[43] 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`.

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

[45] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, November 1999.

[46] S. Savage. Sting: A tcp-based network measurement tool. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, 1999.

[47] J. Sommers, P. Barford, N. Duffield, and A. Ron. Improving accuracy in end-to-end packet loss measurement. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 157–168, 2005.

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

[49] P. Trimintzios, M. Polychronakis, A. Papadogiannakis, M. Foukarakis, E. P. Markatos, and A. Øslebø. DiMAPI: An application programming interface for distributed network monitoring. In *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, April 2006.

[50] 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.

[51] P. Wood. libpcap-mmap. http://public.lanl.gov/cpw/.

[52] 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.

[53] K. Xinidis, I. Charitakis, S. Antonatos, K. G. Anagnostakis, and E. P. Markatos. An active splitter architecture for intrusion detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 03(1):31–44, 2006.

[54] 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.