DiMAPI: An Application Programming Interface for Distributed Network Monitoring

Panos Trimintzios,[†] Michalis Polychronakis,[*] Antonis Papadogiannakis,[*] Michalis Foukarakis,[*]
Evangelos P. Markatos,[*] Arne Øslebø[‡]

[†]European Network and Information Security Agency
Panagiotis.Trimintzios@enisa.eu.int

[‡]UNINETT
Arne.Oslebo@uninett.no

[*]Institute of Computer Science, Foundation for Research & Technology – Hellas
{mikepo,papadog,foukas,markatos}@ics.forth.gr

***Abstract*— Network monitoring and measurement is commonly regarded as an essential function for understanding, managing and improving the performance and security of network infrastructures. Traditional passive network monitoring approaches are not adequate for fine-grained performance measurements nor for security applications. In addition, many applications would benefit from monitoring data gathered at multiple vantage points within a network infrastructure.**

This paper presents the design and implementation of DiMAPI, an application programming interface for distributed passive network monitoring. DiMAPI extends the notion of the *network flow* with the *scope* attribute, which enables flow creation and manipulation over a set of local and remote monitoring sensors. Experiments with a number of applications on top of DiMAPI show that it has reasonable performance, while the response latency is very close to the actual round trip time between the monitoring application and the monitoring sensors. 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.

I. Introduction

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. As networks get faster and as network-centric applications get more complex, our understanding of the Internet continues to diminish. The world is often surprised with security breaches, such as Internet worms and Denial of Service (DoS) attacks, and the extent of unclassified traffic due to applications that use dynamic ports.

Current monitoring standards often force administrators to trade-off functionality for interoperability [1]. Passive traffic monitoring and capture 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 [2]. Current passive network monitoring applications are commonly based on data gathered at a single observation point. For instance, Network Intrusion Detection Systems (NIDS) usually run on a single monitoring host that captures the network packets and processes them by performing tasks such as filtering, TCP stream reassembly, and pattern matching.

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

Recent research efforts [3]–[5] have demonstrated that a large-scale monitoring infrastructure of distributed cooperative monitors can be used for building Internet worm detection systems. Distributed DoS attack detection applications would also benefit from multiple vantage points across the Internet. Finally, user mobility necessitates distributed monitoring due to nomadic users who change locations frequently across different networks.

It is clear from the above 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.

In this paper we present the design, implementation, and performance evaluation of DiMAPI, an programming interface for distributed passive network monitoring. DiMAPI builds on top of MAPI [6], an API for local passive monitoring applications. MAPI enables users to express complex monitoring needs, choose only the amount of information they are interested in, and therefore balance the overhead they pay with the amount of information they receive. The main contribution of DiMAPI is that it elevates the generalized flow abstraction of MAPI into a distributed world, facilitating the programming and coordination of several distributed monitoring sensors in a flexible and efficient way.

The remainder of this paper is organized as follows. Section II outlines the main concepts of MAPI [6] and its basic

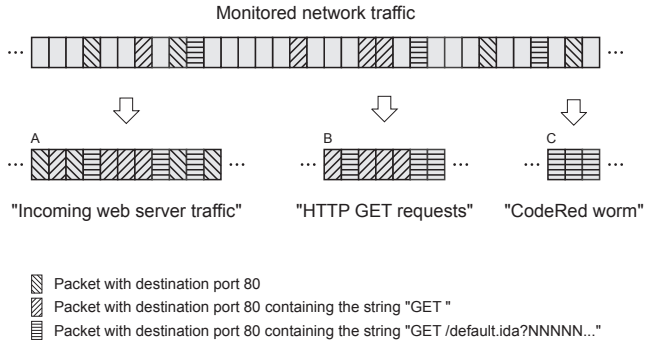

Fig. 1. Network flow examples.

operations. Section III describes the design of DiMAPI, along with some usage examples. In Section IV we present in detail the distributed monitoring architecture and the implementation of DiMAPI. Section V presents the results of the experimental evaluation of DiMAPI. Finally, Section VI summarizes related work and Section VII concludes the paper.

II. Background: The Monitoring API

DiMAPI has been designed and realized by building on the Monitoring Application Programming Interface (MAPI) [6], an expressive and flexible API for passive network traffic monitoring over a single local monitoring sensor. MAPI builds on a generalized network flow abstraction, flexible enough to capture emerging application needs, and expressive enough to allow the system to exploit specialized monitoring hardware, where available. 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. 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.

In this section we introduce the main concepts of MAPI, and briefly describe its most important operations. A complete specification of MAPI is provided in the MAPI man pages [7].

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

Figure 1 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 [8]". 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 [9], 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.

B. Basic MAPI Operations

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

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

This call creates a network flow and returns a flow descriptor `fd` that refers to it. By default, a newly created flow consists of all network packets that go through the monitoring interface `dev`. When a network flow is not needed any more, it can be closed using `mapi_close_flow()`.

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

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

The above call applies the function `f` 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.

MAPI provides several *predefined* functions that cover a broad range of standard monitoring needs. Several functions

are provided for restricting the packets that will constitute a network flow. For example, applying the BPF_FILTER function with parameter "tcp and dst port 80" restricts the packets of a network flow to the TCP packets destined to port 80, as in flow A of Figure 1. STR_SEARCH can be used to restrict the packets of a flow to only those that contain a specified byte sequence. Network flows B and C in Figure 1 would be configured by applying both BPF_FILTER and STR_SEARCH. Many other functions are provided for processing the traffic of a flow. Such functions include PKT_COUNTER and BYTE_COUNTER, which count the number of packets and bytes of a flow, SAMPLE, which can be used to sample packets, HASH, for computing a digest of each packet, and REGEXP, for pattern matching using regular expressions.

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. Results retrieval is achieved using the following call:

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

The above function returns a pointer to the memory where the result of the function with the identifier fid, which has been applied to the network flow fd, has been stored. Once a flow is established, packets belonging to that flow can be read one-at-a-time using the following blocking call:

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

The above function reads the next packet that belongs to flow fd. In order to read packets, the function TO_BUFFER (which returns the relevant fid parameter) must have previously been applied to the flow. TO_BUFFER instructs the monitoring system to store the captured packets into a shared memory area, from where the user can directly read the packet, supporting this way efficient zero-copy packet capturing platforms [10], [11].

III. Distributed Passive 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 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 sub-network (as in AS 2). An authorized user, who is not necessarily located in any of the participating ASes, can run monitoring applications that involve an arbitrary number of the available monitoring sensors.

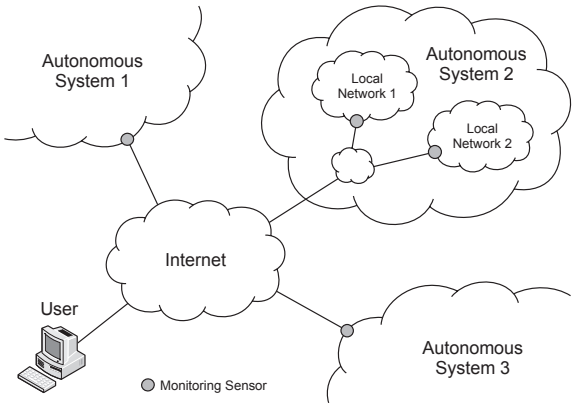

Fig. 2. A high-level view of a distributed passive network monitoring infrastructure.

In order to take advantage of information from multiple vantage points, monitoring applications need a uniform access platform for interaction with the remote monitoring sensors. DiMAPI fulfils this requirement by facilitating the programming and coordination of a set of remote sensors from within a single monitoring application. This is achieved by building on the abstraction of the network flow introduced in MAPI. However, MAPI supports the creation of network flows associated with a *single* local monitoring interface, and thus, in MAPI, a network flow receives network packets 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 in DiMAPI enables 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, it effectively manipulates and extracts information concurrently from all these monitoring points.

A. Extensions to MAPI

In order to support the abstraction of scope in DiMAPI, the interface and implementation of mapi_create_flow() have been extended to support the definition of multiple remote monitoring interfaces. A remote monitoring interface is

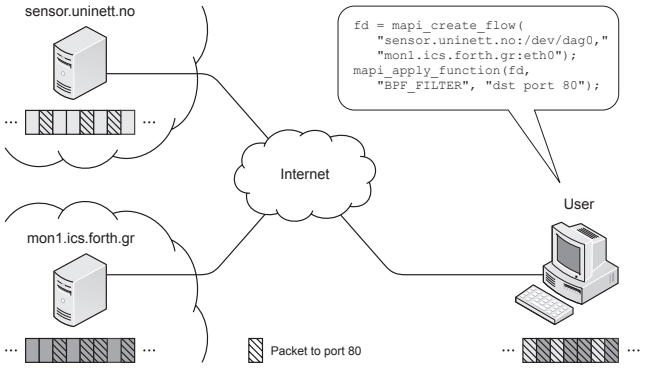

Fig. 3. An example where a monitoring application manipulates a network flow associated with two remote monitoring sensors located in different administrative domains.

defined as a `host:interface` pair, where `host` is the host name or IP address of the remote sensor and `interface` is the device name of the monitoring interface or the name of a packet trace file. The scope of a network flow is composed 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 3, a user creates a network flow associated with two remote sensors located in two different organizations, FORTH and UNINETT. The user then applies the function `BPF_FILTER` in order to restrict the packets of the flow to only those that are destined to some web server (some code has been omitted for clarity). As a result, the application receives packets with destination port `80` that are captured at both UNINETT's and FORTH's sensors.

The scope abstraction also allows the creation of flows associated with multiple interfaces on the same local or remote 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.forth.gr:/dev/dag0, m1.forth.gr:eth1");
```

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

B. Monitoring Application Examples

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—one may think of much more complicated monitoring applications to exploit the full power of DiMAPI.

1) Web Traffic Byte Counter: The following pseudocode illustrates a simple DiMAPI application that counts the total bytes of the packets received by the web servers of several monitored networks within a predefined interval.

```
1  fd = mapi_create_flow(
2      "host1:eth2, host2:/dev/dag0, host3:eth1");
3
4  /* keep only packets directed to a web server */
5  mapi_apply_function(fd, "BPF_FILTER",
6      "tcp and dst port 80");
7
8  /* and just count them */
9  fid = mapi_apply_function(fd, "BYTE_COUNTER");
10
11 mapi_connect(fd);
12 sleep(10);
13
14 bytes = mapi_read_results(fd, fid);
15 ...
```

The application operates as follows. We initially define a network flow with a scope of three remote sensors (line 1). Then, we restrict the packets of the flow to only those destined to some web server, by applying the `BPF_FILTER` function (line 5). 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 9). Finally, we activate the flow (line 11). After 10 seconds, the application reads the result by calling `mapi_read_results()` (line 14).

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, masquerading as normal web traffic in order to bypass strict firewall configurations aiming to block P2P traffic. The following pseudocode illustrates how DiMAPI can be used for writing a simple application that identifies file sharing clients joining the Gnutella network using covert web traffic.

```
1  fd = mapi_create_flow(
2      "host1:eth2, host2:/dev/dag0, host3:eth1");
3
4  /* keep only web packets */
5  mapi_apply_function(fd, "BPF_FILTER",
6      "tcp and port 80");
7
8  /* indicating Gnutella traffic */
9  mapi_apply_function(fd, "STR_SEARCH",
10     "GNUTELLA CONNECT");
11
12 /* and just count them */
13 fid = mapi_apply_function(fd, "PKT_COUNTER");
14
15 mapi_connect(fd);
16
17 /* forever, report the number of packets */
18 while(1) {
19     sleep(60);
20     cnt = mapi_read_results(fd, fid);
21     ...
22 }
```

Similarly to the previous example, we initially create a network flow that receives the packets seemingly destined to, or

coming from, some web server. 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 that contain this characteristic string (line 9). After specifying the characteristics of the flow, we instruct the monitoring system that we are interested in just counting the number of packets by applying the PKT_COUNTER function (line 13). Finally, we activate the flow (line 15). After 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 current value of the counter in a infinite loop (line 20).

Implementing the above simple distributed monitoring application using existing tools and libraries would have been a tedious process, resulting in longer code and higher overheads. For example, we could use tools like snort [12] or ngrep [13], 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.

Alternatively, one could build the application using solely WinPcap [14] or rpcap [15]. Both libraries extend libpcap [16] with remote packet capture capabilities, allowing captured packets at a remote host to be transferred to a local host for further processing. 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, and then identify locally the Gnutella packets. 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 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.

Clearly, such custom schemes do not scale well and cannot offer the ease of use and flexibility of DiMAPI for building distributed monitoring applications. Furthermore, DiMAPI exploits any specialized hardware available at the monitoring sensors, and efficiently shares the monitoring infrastructure among many users. 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 [6].

IV. DiMAPI Architecture

Figure 4 illustrates the architecture of a monitoring sensor that supports DiMAPI. The overall architecture includes one or more monitoring interfaces for capturing traffic, a monitoring agent, which provides optimized passive monitoring services,

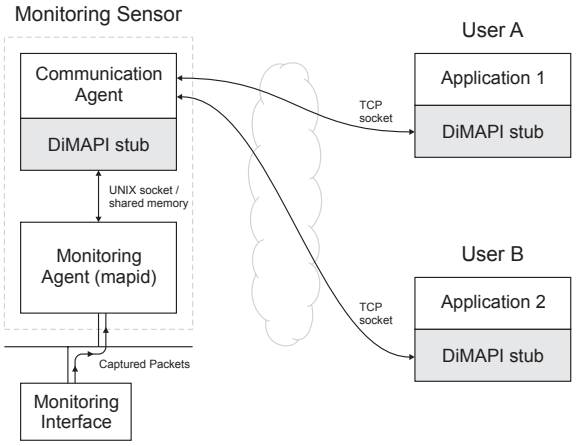

Fig. 4. Architecture of a DiMAPI monitoring sensor. Remote user applications interact with the Monitoring Agent through the Communication Agent.

the DiMAPI stub, for writing monitoring applications, a communication agent for interaction with the remote monitoring applications, and finally, the actual monitoring applications.

The monitoring host 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 network "taps." Packets are captured and processed by the monitoring agent [6], called mapid: a user-level process with exclusive access to the captured packets. Mapid is optimized to perform intensive monitoring operations 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 [6].

Remote applications must be able to communicate their monitoring requirements to each sensor through the Internet. A straightforward approach for enabling applications to communicate with a remote sensor would be to modify mapid to interact directly with the remote applications through the DiMAPI stub. However, this design requires changes to be made to the monitoring agent, which poses several risks. Indeed, mapid is a complex part of the software architecture and is already responsible for handling important "heavy-duty" tasks, as this is where all the processing of the monitoring requirements of the users' applications takes place, and thus, has to keep up with intensive high-speed packet processing. Besides increasing the software complexity of mapid, extending it 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 4. We have introduced an *intermediate* agent between

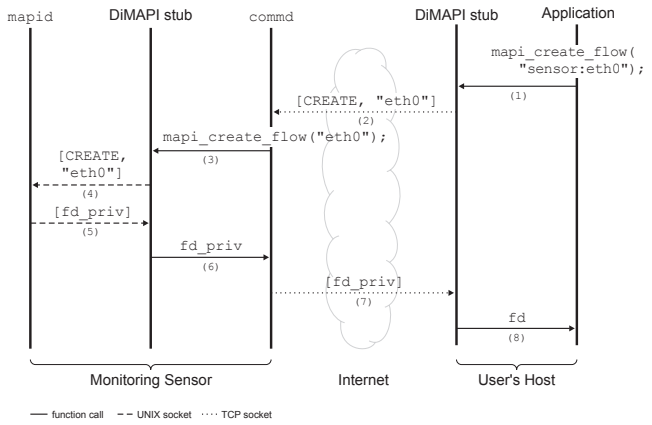

Fig. 5. Control sequence diagram for the execution of the function `mapi_create_flow()`.

`mapid` and the remote applications, for handling all remote communication. This *Communication Agent* (`commd`), 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 `commd` is completely transparent to user applications, which continue to operate as if they were interacting directly with `mapid`—only the DiMAPI stub is aware of the presence of `commd`. Furthermore, the presence of `commd` is also transparent to `mapid` since it operates as a typical local monitoring application on top of MAPI.

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

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

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

The communication agent listens for monitoring requests from DiMAPI applications to a known predefined port. A new thread is spawned for each new remote application and thereafter handles all the communication between the monitoring application and `commd`. The DiMAPI stub that is linked with the application sends a control message to `commd`, for each DiMAPI call invocation, which in turn repeats the call. This time though, the stub of `commd` will interact directly with the `mapid` running on the same host. `Commd` then returns the result to the stub of the remote application, which in turn returns it to the user. Note that although `commd` is implemented on top of DiMAPI, it uses only the subset of DiMAPI that is intended for local network monitoring [6], and communicates locally with `mapid` solely through shared memory and UNIX sockets, since it never manipulates network flows associated with remote monitoring sensors.

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

Finally, the stub of the application has to return a flow identifier back to the user. However, in case that the network flow is associated with more than one monitoring sensors, the DiMAPI stub of the application will receive several flow descriptors, one for each of the monitoring interfaces constituting the scope of the network flow.[1] Thus, the stub generates and returns a new unique flow identifier, and internally stores the mapping between the received flow descriptors and the newly created identifier (step 8).

[1]In that case, steps 2–7 in Figure 5 are repeated for each sensor of the network flow's scope.

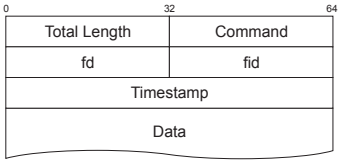

Fig. 6. Format of the control messages exchanged between the DiMAPI stub and `commd`.

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, are performed very fast. The operations with the highest overhead 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 V.

B. Communication Protocol

All communication between the monitoring sensors and the remote applications is encapsulated in the DiMAPI stub. The design target of the communication protocol was to provide communication with minimal overhead and good scalability over a large number of monitoring sensors. DiMAPI stub library calls exchange control messages with `commd` through TCP sockets. Each message contains all the necessary information for the operation to be executed. After sending a request, the stub waits for the corresponding acknowledgement from the sensor, indicating the successful completion of the requested action, or a specific error in case of failure.

The format of the messages exchanged between the DiMAPI stub and `commd` is shown in Figure 6. Each message has variable length, denoted by the field `Total Length`. The `Command` field contains the operation type, sent by the stub to `commd`, or the acknowledgment value for a request that `commd` has processed. It takes values from an enumeration of possible message types. 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`, and so on.

Fields `fd` and `fid` contain the descriptors of the network flow and the applied function instance being manipulated, respectively. `Timestamp` contains the time at which the result included in the communication message was produced. Finally, the field `Data`, the only with variable size, serves different purposes depending on the contents of the `Command` field. Specifically, when the message is a reply (acknowledgement) from `commd` to a call of `mapi_read_results()`, it contains the results of an applied function. If the `Command` field contains a request, sent from the DiMAPI stub, e.g., to apply some function to a network flow, it may contain the arguments of the relevant DiMAPI function. For example, in a call to `mapi_apply_funtion()`, it contains the name of the function to be applied along with its arguments.

C. Security and Privacy

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

To counter such threats, each sensor is equipped with a firewall, configured using a conservative policy that selectively allows inbound traffic only from the predefined IP addresses of legitimate users. Inbound traffic from any other source is dropped. Since our system is based on the Linux OS, such a policy can be easily implemented using `iptables` [17].

The administrator of each monitoring sensor is responsible for issuing credentials to users who want to access the monitoring sensor with DiMAPI. The credentials specify the usage policy applicable to that user. Whenever a user's monitoring application connects to some monitoring sensor and requests the creation of a network flow, it passes the user's credentials. The monitoring sensor performs *access control* based on the user's request and credentials. In this way, administrator delegates authority to use that sensor, using public key authentication. Access control in our system is based on the KeyNote [18] trust-management system, which allows direct authorization of security-critical actions.

Since all communication between user applications and the remote sensors will be made through public networks across the Internet, special measures must be taken in order to ensure the *confidentiality* of the transferred data. Data transfers through TCP are unprotected against eavesdropping from third-parties that have access to the transmitted packets, since they can reconstruct the TCP stream and recover the transferred data. This would allow an adversary to record DiMAPI's control messages, forge them, and replay them in order to access a monitoring sensor and impersonate a legitimate user. For protection against such threats, any communication between the DiMAPI stub and a remote sensor is encrypted using the Secure Sockets Layer protocol (SSL).

In a distributed monitoring infrastructure that promotes sharing of network packets and statistics between different parties, exchanged data should be *anonymized* before made publicly available for security, privacy, and business competition concerns that may arise due to the lack of trust between the collaborating parties. The DiMAPI architecture supports an advanced framework for creating and enforcing anonymization policies [19]. 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.

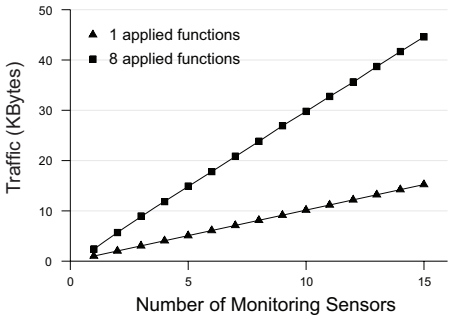

Fig. 7. Total network traffic exchanged during the initialization phase, i.e., creation, configuration, and instantiation of a network flow, when applying 1 and 8 functions.

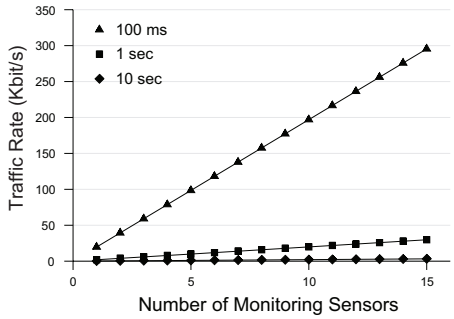

Fig. 8. Network overhead incurred with a DiMAPI monitoring application that uses function `BYTE_COUNTER`, with polling periods 0.1, 1, and 10 seconds.

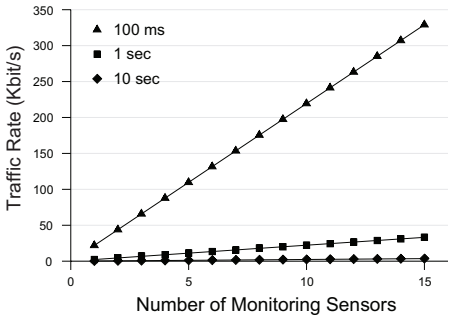

Fig. 9. Network overhead incurred with a DiMAPI monitoring application that uses function `HASHSAMP`, with polling periods 0.1, 1, and 10 seconds.

V. Performance Evaluation and Experience

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. Furthermore, we discuss our experiences with a real world application built on top of DiMAPI.

A. Experimental Environment

For the experimental evaluation of DiMAPI we used two different monitoring sensor deployments. The first system consists of 15 monitoring sensors distributed across the internal network of FORTH. All nodes are interconnected through 100 Mbps Ethernet for the sensor control interface. Each sensor is equipped with a second Ethernet interface for the actual passive network monitoring. The monitored test traffic is generated using `iperf` [20] and `tcpreplay` [21]. 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. The operating system of the sensors in both deployments is Linux (various distributions).

B. Network Overhead

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

For the experiments of this section, we implemented a test monitoring application that creates a network flow, configures it by applying several functions, and then periodically reports some result according to the applied functions. This application operates in a similar fashion to the examples presented in Section III-B. 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 DiMAPI application running on the same host with the test application. This local application reports the amount of DiMAPI control traffic by creating a network flow that captures all packets to and from the DiMAPI control port. Since it is a local monitoring application, it incurs no network traffic. We validated our results using `tcpdump`.

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 `commd` during the creation, configuration, and instantiation of a network flow. For example, in the pseudo-code of Section III-B.1, the initialization phase includes lines 1–11, and comprises four DiMAPI function calls, while in the second example of Section III-B.2, the initialization phase is between lines 1–15, and includes calls to five functions.

Figure 7 shows the amount of traffic generated during the initialization phase for two variations of the test application. In the first variation, corresponding to the dashed line, the network flow is configured by applying only one function, which results to a total of three DiMAPI library function calls for the initialization phase. In the second variation, the network flow is configured by applying 8 functions, a rather extreme case, resulting to a total of 11 DiMAPI library function calls. The incurred traffic grows linearly with the number of monitoring sensors, and, for 15 sensors, reaches about 15 KBytes for the first variation and 45 KBytes for the second. In both cases, the network overhead remains low, and can be easily amortized during the lifetime of the application.

In the next experiment we measured the rate of the network traffic incurred during the lifetime of the application due to the periodic results retrieval. After the initialization phase, the test application constantly reads the new value of the result by periodically calling `mapi_read_results()` at a

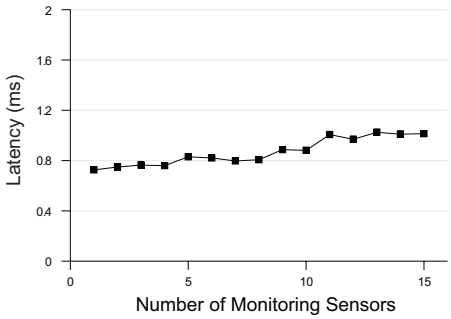

Fig. 10. Completion time for `mapi_read_results()`. This includes the processing within DiMAPI stub, the processing within each of the remote monitoring sensors, and the network round-trip time.

predefined time interval. The measured traffic includes both the control messages of DiMAPI and the data transferred, across all monitoring sensors. We modified the test application to read the number of bytes of a network flow in three different periodic intervals, and plotted the mean rate of the generated traffic for one hour in Figure 8.

In case that the application 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/s, when using a network flow with a scope of 15 sensors. However, for periodic intervals of one second or more, the generated traffic is negligible.

The result of the `BYTE_COUNTER` function is an unsigned 8-byte integer. In order to see the effects of larger result structures, we repeated the experiment by reading the results of the `HASHSAMP` function. `HASHSAMP` is used to perform hash-based sampling on the packets of a network flow, and its results format is a 36-byte structure. The traffic rate when reading the results of `HASHSAMP` is shown in Figure 9. We see that there is only a slight increase in the traffic rate due to the larger size of the produced results.

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

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

We measured the time for retrieving a result by calling `mapi_read_results()` using the same test application we used for the experiments of Section V-B 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.

TABLE I

COMPARISON BETWEEN THE COMPLETION TIME OF A DIMAPI CALL AND THE NETWORK ROUND TRIP TIME.

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	-

Figure 10 shows 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 slight increase in the delay for retrieving the result. Since all the sensors are located within the FORTH LAN, the network latency for each monitoring sensor is almost constant and remains very low. Thus, the delay for retrieving the result from 15 sensors also remains very low, below 1 ms.

In order to explore how the network latency affects the delay of DiMAPI calls under more realistic conditions, we repeated the experiment using the second sensor deployment. As described in Section V-A, 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 I. The third column shows the actual RTTs for each sensor, as measured from the end host using `ping`. We measured the delay of `mapi_read_results()` for reading a result from each monitoring sensor. The results of Table I 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.

D. Distributed Intrusion Detection with DiMAPI

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

number, and defines a string search for `/bin/perl.exe` in the packet payload.

Implementing a distributed NIDS is a rather complicated task. Several basic operations like packet classification, TCP/IP stream reconstruction, and pattern matching, must be crafted together to form a fully functional system. Each one of these operations alone requires deliberate decisions for its design and considerable programming effort for its implementation. Furthermore, the resulting system is usually targeted to a specific hardware platform. For instance, the majority of current NIDSes are built on top of `libpcap` [16] packet capture library using commodity network interfaces set in promiscuous mode. As a result, given that `libpcap` provides only basic packet delivery and filtering capabilities, the programmer has to provide considerable amount of code to implement the large and diverse space of operations and algorithms required by a NIDS.

In contrast, DiMAPI inherently supports the majority of the above operations in the form of predefined functions which can be applied to network flows, and thus, can be effectively used for the development of a simple NIDS. Consequently, a great burden is released from the programmer who has now a considerably easier task. Furthermore, a NIDS based on DiMAPI is not restricted to a specific hardware platform. DiMAPI operates on top of a diverse range of monitoring hardware, including sophisticated lower level components [22], and thus, can further optimize overall system performance, considering that certain functions can be pushed to hardware.

We have developed a distributed NIDS using DiMAPI. Based on the observation that a rule which describes a known intrusion threat can be represented by a corresponding network flow, the overall implementation is straightforward. As an example, consider the following `snort` [12] signature for detecting malicious activity from a host infected with a backdoor:

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

All packets that match the above rule can be returned by a network flow, after the application of only two DiMAPI functions:

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

Our DiMAPI-based NIDS operates as follows: during program start-up, the files that contain the set of rules are parsed, and for each rule, a corresponding network flow is created. Rules are written in the same description language used by `snort`. The rest of the functionality is left to DiMAPI, which will optimize the functional components of all the defined rules and deliver any matching packets.

We should note that DiMAPI is not designed to fully replace fully-blown NIDS platforms. Our experience with intrusion detection shows that DiMAPI provides reasonably good support and performance for basic intrusion detection functionality, especially in a distributed monitoring environment. This is useful for providing the defense capabilities needed to respond to coordinated large-scale attacks. A key advantage of our DiMAPI-based NIDS is that the it can rely on more than one vantage points for attack detection. The scope of the network flows that correspond to the signatures is user-defined, thus the system is able to observe and correlate malicious activity from multiple sources. This functionality is crucial for building distributed early-warning systems for large-scale attacks like Internet worms [3].

VI. Related Work

There are several techniques and tools currently available for passive network monitoring, which can be broadly categorized into three categories [2]: 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 probided 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 passive packet monitoring is `libpcap` [16], which provides a portable API for user-level packet capture. The `libpcap` interface supports a filtering mechanism based on the BSD Packet Filter [23], which allows for selective packet capture based on packet header fields. The Linux Socket Filter [24] offers similar functionality with BPF for the Linux OS, while xPF [25] and FFPF [26] provide a richer programming environment for network monitoring at the packet filter level. FLAME [27] is an architecture that allows users to directly install custom modules on the monitoring system, similarly in principle to Management-by-Delegation models [28]. Windmill [29] is an extensible network probe environment which allows loading of "experiments" on the probe for analyzing protocol performance.

The `libpcap` library has been widely used in several passive monitoring applications such as packet capturing [13], [30], network statistics monitoring [31], and intrusion detection systems [12]. `WinPcap` [14] and `rpcap` [15] 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 functionality through the scope abstraction for *multiple* distributed monitoring sensors. 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 [32]. `libcoral`

provides an API for monitoring applications that is independent of the underlying monitoring hardware. Nprobe [33] 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.

DiMAPI leverages current passive network monitoring/capturing approaches that are tied to a single monitoring host, into a distributed environment. Indeed, DiMAPI is implemented on top of various monitoring architectures, including `libpcap`-based interfaces and DAG cards [11], and provides a flexible interface on top of them for building distributed monitoring applications. To the best of our knowledge, DiMAPI is also more expressive than existing tools. For instance, DiMAPI can be used to filter packets based on payload data, apply arbitrary functions on packets, and keep statistics based on the results of these functions.

Except from packet capture oriented systems, there has been significant activity in the design of systems providing flow-based measurements. Cisco IOS NetFlow technology [9] collects and measures traffic data on a per-flow basis. In this context, a flow is usually defined as all packets that share a common protocol, source and destination IP addresses, and port numbers. In contrast to packet capture systems, NetFlow only extracts and maintains flow-level records, from which various traffic statistics can be derived. NeTraMet [34], much like NetFlow, can collect traffic data on a per-flow basis focusing only on flows that match a specific rule.

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` [35] offer NetFlow record generation by capturing packets using commodity hardware. Although DiMAPI shares some goals with the above flow-based monitoring systems, we believe that it has significantly more functionality. For example, by being able to examine packet payloads, DiMAPI is able to provide sophisticated traffic statistics, e.g., for applications that use dynamically allocated ports [6].

There is also significant activity within the IETF for defining network traffic monitoring standards, from several working groups such as RMON [36], PSAMP [37], and IPFIX [38]. The Simple Network Management Protocol (SNMP) [39] facilitates the retrieval of traffic statistics from network devices. Though SNMP is useful it is limited in capabilities, since we cannot perform fine-grained monitoring. RMON offers significantly more capabilities than SNMP, however its complexity and overhead prohibit wide deployment [2].

As network traffic monitoring is becoming increasingly important for the operation of modern networks, several passive monitoring infrastructures have been proposed. Gigascope [40] is a stream database for storing captured network data in a central repository for further analysis using the GSQL query language. A similar approach is followed by the CoMo project [41], which allows users to query network data gathered from multiple administrative domains, and Sprint's passive monitoring system [42], which also collects data from different monitoring points into a central repository for analysis. Arlos et al. [43] propose a distributed passive measurement infrastructure that supports various monitoring equipment within the same administrative domain.

Finally, a lot of work is being done in the area of monitoring of high-performance computing systems, such as clusters and Grids. Ganglia [44] is a distributed monitoring system based on a hierarchical design targeted at federations of clusters. GridICE [45] is a distributed monitoring tool integrated with local monitoring systems with a standard interface for publishing monitoring data. Note that such systems could utilize at lower levels the functionality offered by DiMAPI.

VII. Conclusions and Further Work

We have presented the design, implementation and performance evaluation of DiMAPI, an API for building distributed 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 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 scope.

A broad range of advanced monitoring applications can benefit from DiMAPI to efficiently perform fine-grained monitoring tasks over a potentially large number of passive monitoring sensors. Such application include collaborative early warning of large-scale attacks, accurate traffic distribution characterization, even for applications that use dynamic ports, identification of unexpected performance problems, and extraction of detailed traffic characteristics.

Currently, our efforts are focused on the deployment of DiMAPI-enabled monitoring sensors across several autonomous systems, aiming to create a large-scale passive monitoring infrastructure. We are also looking at the automation of the sensor discovery process. There has been a lot of work done for scalable resource discovery [46], so we are currently looking at how these approaches can be integrated with our design. Finally, we are exploring the possibility of extending the functionality of the communication agent in order to allow the monitoring sensors to communicate with each other, for potential monitoring applications that require such functionality.

Availability

DiMAPI is available at `http://mapi.uninett.no`

ACKNOWLEDGMENTS

This work was supported in part by the IST project LOBSTER funded by the European Union under contract number 004336, and in part by the project CyberScope funded by the Greek General Secretariat for Research and Development under contract number PENED 03ED440. Michalis Polychronakis, Antonis Papadogiannakis, Michalis Foukarakis, and Evangelos P. Markatos are also with the University of Crete. The work of Panos Trimintzios was done while at ICS-FORTH. We would like to thank Kostas Anagnostakis and the members of the DCS Lab at ICS-FORTH for their valuable assistance and support.

REFERENCES

[1] Peter Morriessy, "RMON2: To the Network Layer and Beyond!" *Network Computing*, Feb. 1998, http://www.nwc.com/903/903f1.html.

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

[3] 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)*, 2003, pp. 190–199.

[4] 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.

[5] 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.

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

[7] "MAPI Public Release," http://mapi.uninett.no.

[8] eEye Digital Security, ".ida "Code Red" Worm," http://www.eeye.com/html/Research/Advisories/AL20010717.html.

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

[10] 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.

[11] *DAG 4.3GE dual-port gigabit ethernet network monitoring card*, Endace measurement systems, 2002, http://www.endace.com/.

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

[13] J. Ritter, "ngrep – Network grep," http://ngrep.sourceforge.net/.

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

[15] S. Krishnan, "rpcap," http://rpcap.sourceforge.net/.

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

[17] Rusty Russell, "Linux 2.4 Packet Filtering HOWTO," 2002, http://www.netfilter.org/documentation/HOWTO//packet-filtering-HOWTO.html.

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

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

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

[21] A. Turner, "tcpreplay," http://tcpreplay.sourceforge.net/.

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

[23] 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*, January 1993, pp. 259–270.

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

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

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

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

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

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

[30] The Tcpdump Group, "tcpdump," http://www.tcpdump.org/.

[31] L. Deri, "ntop," http://www.ntop.org/.

[32] 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.

[33] 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.

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

[35] L. Deri, "nProbe," http://www.ntop.org/nProbe.html.

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

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

[38] 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.

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

[40] 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*, 2003.

[41] 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.

[42] 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.

[43] 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)*, 2005, pp. 215–227.

[44] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience," *Parallel Computing*, vol. 30, no. 7, July 2004.

[45] 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*, vol. 21, no. 4, pp. 559–571, Apr. 2005.

[46] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat, "Distributed Resource Discovery on PlanetLab with SWORD," in *Proceedings of the 1st Workshop on Real, Large Distributed Systems*, Dec. 2004.