The CoMo White Paper

Gianluca Iannaccone†, Christophe Diot†, Derek McAuley†§,
Andrew Moore§, Ian Pratt§, Luigi Rizzo‡

† Intel Research
Cambridge, UK

§ Computer Laboratory
Cambridge University, UK

‡ Dip. Ingegneria dell'Informazione
University of Pisa, Italy

***Abstract*— CoMo (Continuous Monitoring) is a passive monitoring system. CoMo has been designed to be the basic building block of an open network monitoring infrastructure that would allow researchers and network operators to easily process and share network traffic statistics over multiple sites. This paper identifies the challenges that lie ahead in the deployment of such an open infrastructure. These main challenges are: (1) the system must allow any generic metric to be computed on the incoming traffic stream, (2) it must provide privacy and security guarantees to the owner of the monitored link, the network users and the CoMo users, and (3) it must be robust in the face of anomalous traffic patterns. We describe the high-level architecture of CoMo and, in greater detail, the resource management, query processing and security aspects.**

I. Introduction

Despite the great interest of recent years in measurement based Internet research, the number and the variety of data sets and network monitoring viewpoints remains unacceptably small. Several players in the network measurement area, like CAIDA [2], NLANR [19], RouteViews [25], RIPE [22], Internet2 [13] and, recently, GEANT [9] have provided data sets with various degrees of accuracy and completeness. Some large commercial ISPs also deploy private infrastructures and share limited information with the rest of the research community [6], [8]. Other network operators (e.g., corporate networks, stub ISPs, Universities) instead tend to lack a measurement infrastructure or, in case they do have one, do not share any data or even report the existence of such infrastructure.

This situation constitutes a major obstacle for network researchers. It hampers the ability to validate the results over a wide and diverse range of datasets. It makes it difficult to generalize results and identify the presence of traffic characteristics that are invariant and common to all networks. For example, it is difficult to quantify the magnitude of a denial of service attack or a worm infection, to evaluate the relevance of network pathologies such as in-network packet duplication and reordering, or to simply identify the dominant applications in the Internet.

So far, several barriers have limited the ability of researchers to deploy and share large number of network datasets:

- The cost of the monitoring infrastructure that should be present on a large set of links with speeds varying from the few Mbps of small organizations up to the 10Gbps of ISPs' networks.
- The lack of software tools for managing a passive monitoring infrastructure. Today, monitoring systems use ad-hoc tools that are not appropriate for large infrastructures. For example, they tend to provide poor and inefficient query interfaces.
- The lack of a standard open interface to access the monitoring system. Although, several efforts exist to provide a single packet trace format (e.g., tcpdump/libpcap [24], IETF IPFIX working group [14]), no standard exists to access high speed network monitoring devices.
- Organizations do not see any benefit in deploying passive monitoring systems and sharing the information. Clearly, this is not a need for classical network administration tasks (e.g., reachability tests, loss rates, delays), but we believe that the proliferation of denial of service attacks, viruses and worms will help finding the right incentive in sharing information and participating in a common effort for understanding network traffic.
- The difficulty in controlling the access to the data avoiding to disclose private information about network users or organizations.

We have designed CoMo, an open software platform for passive monitoring of network links. CoMo has been designed to be flexible, scalable, and to run on commodity hardware. With CoMo, we intend to lower the barriers described above and encourage the deployment of a large scale monitoring infrastructure. We believe this effort constitutes a necessary first step towards a better understanding of network protocols and traffic. For example, the extensible nature of CoMo allows early deployment of novel methods for traffic analysis, anomaly diagnosis or network performance evaluation.

The rest of the paper describes the challenges posed by the design of the CoMo platform and its resulting architecture. Then, we focus on specific aspects of CoMo that are open, longer term issues, namely the query engine, resource management and security.

II. Challenges

This section describes the requirements the CoMo systems need to satisfy and the design challenges they need to address in order to be successfully deployed.

Revision date: September 25th, 2004.
Project web page: http://www.cambridge.intel-research.net/como.

A. Requirements

The design of a system such as CoMo is driven by the trade-off between openness and resilience. The system should be open enough to allow any user to compute any metrics on the observed traffic. At the same time, the system should be robust to guarantee no "black-out" periods, when it cannot sustain the incoming packet rate, and no abuses by unauthorized users. We can summarize the system requirements as follows:

Openness. The users should be able to customize the system and the software platform to their specific needs and deployment environment. For example, a user interested in intrusion detection or performance analysis may only need limited storage; on the other hand, an ISP's network operator interested in post-mortem analysis might require to store and retrieve a large and detailed packet-level or flow-level information.

The metrics also need to be dynamically configurable in order to address a large range of applications such as network trouble-shooting, anomaly and intrusion detection, SLA computation, etc. In addition, the system should allow users to easily interact and interface with it in order to start or stop some metric computation or to run a query on the collected datasets.

Resilience. This requirement is often orthogonal to the previous one. First and most important, the system should be able to monitor *and* analyze the traffic in any load condition, and in particular in the presence of unexpected traffic anomalies that may overload the system resources. The system should control its resources carefully. For example, the computation of one metric should not monopolize the use of resources, starving other crucial system tasks (e.g., packet trace collection). The owner of the system needs to be able to control the access to the system. Various type of users will want to access a monitoring system, including malicious ones. Because of its exporting capabilities, a system can impact the network it measures. Different request will be processed with different priorities. The system should also make sure it does not compute the same metric twice for two different users or applications.

B. Design Challenges

Given the requirements described above, we identify four main design challenges:

Ease of deployment. The success of the CoMo infrastructure will be a function of how simple it is for user to access the infrastructure, specify and implement traffic metrics and analysis methods, query the data from the system. As we will point out in Section III, many of the design decisions are driven by the need to trade architecture simplicity for efficiency and performance.

Query interface. Designing a query interface with the constraints defined in the requirement section poses two problems: (i) how to express the query; (ii) how to run the query without explicit built-in support into the system. Expressing a query may be particularly hard given that the system is supposed to be used by a large range of users, defining new traffic metrics and analysis methods. It is unlikely that all metrics will be specified well enough to be translated in a standard query language; metrics may require new constructs that are not present in the original query language. However, the system should still allow custom-built queries to run. We will address this problem in Section IV.

Resource Management. Opening the system to a potentially large number of users requires a very careful resource management, i.e. CPU, memory, I/O bandwidth or storage space. Indeed, allowing users to compute any metric on the traffic stream may result in analysis that are particularly computing intensive, and in a large amount of data to be exported. Therefore, the system needs to define strict policies for analysis metrics and needs to be able to enforce them and possibly to adapt based on the load of the system. We will address the problem of resource management in Section V.

Security issues. CoMo users will have different rights on the system depending also on the system environment. For example, a network operator may be allowed to inspect the entire packet payload in order to spot viruses or worms, while a generic user may only be able to access the packet header (probably anonymized). A second security aspect is related to the vulnerability of the monitoring system. CoMo systems contain confidential information. They can also export large amounts of traffic and impact the network where they are installed. An attacker may also target directly the CoMo system by preventing users to access the system or by corrupting the collected data. In Section VI we will describe the threat model for CoMo and propose initial solutions.

III. Architecture

This section presents a high-level description of the architecture and an overview of the major design choices. We call *data* any measurement related information. Data include original packets captured on the monitored links, as well as statistics computed on the packets and other representations of the original packet trace such as flow records.

A. High level architecture

The system is made of two principal components. The *core processes* control the data path through the CoMo system, including packet capture, export, storage, query management and resource control. The *plug-in modules* are responsible for various transformations of the data.

The data flow across the CoMo system is illustrated in Figure 1. The white boxes indicate plug-in modules while gray boxes represent the core processes. On one side, CoMo collects packets (or subsets of packets) on the monitored link. These packets are processed by a succession of core processes and end stored onto hard disks. On the other side, data are retrieved from the hard disk on user request (by the way of queries addressed to a CoMo system). Before being exported to users, those data go through an additional processing step.

As explained earlier, the modules execute specific tasks on data. The core processes are responsible for the "management"

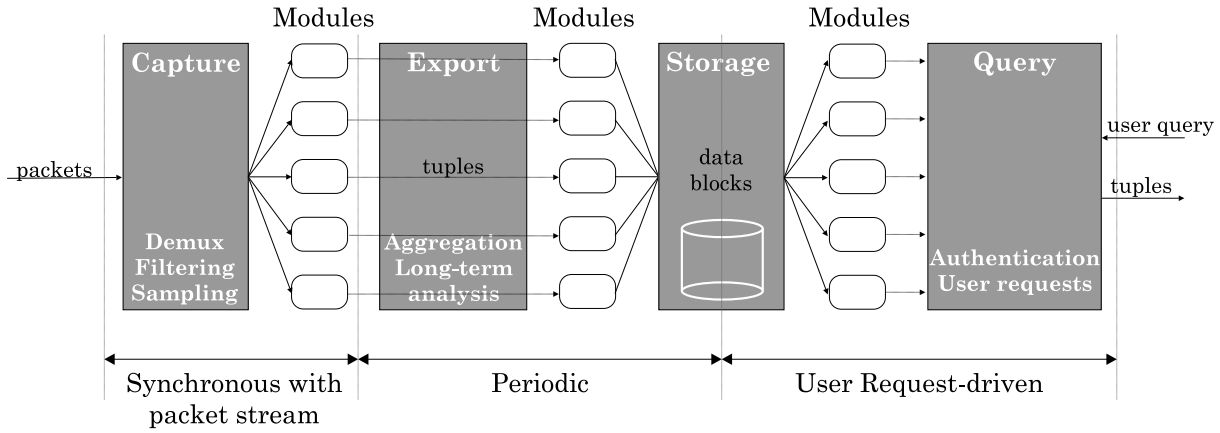

Fig. 1. Data flow in the CoMo system

operations, common to all modules (e.g., packet capture and filtering, data storage). The following tasks also fall under the responsibility of the core component: (i) resource management, i.e., deciding which plug-in modules are loaded and running, (ii) policy management to manage the access privileges of the modules, (iii) on-demand requests handling to schedule and respond to user queries, and finally (iv) exception handling to manage the situation of traffic anomalies and the possible graceful degradation of system performance.

The modules take data on one side and deliver user-defined traffic metrics or processed measurement data on the other side. One of the challenges identified in Section II is to keep modules very simple. All complex functions should be implemented within the core component. This strict division of labor allows us to optimize the core component[1], while the modules can run sub-optimally and can be implemented independently by any CoMo user.

B. The core processes

The core processes are in charge of data movement operations (i.e., from the packet capture card to memory and to the disk array). Moving data in a PC is the most expensive task given memory, bus and disk bandwidth limitations. Therefore, in order to guarantee an efficient use of the resources, it is better to maintain a centralized control of the data path. However, one of the goal of the architecture is to allow the deployment of CoMo as a cluster using dedicated hardware systems (such as network processors) for high performance monitoring nodes.

Communication between core processes is governed by a unidirectional message passing system to enable the partition of functionality over a cluster.

In a single system, a CoMo node uses instead shared memory and Unix sockets for the signaling channel. The use of processes instead of threads is justified by the need of high portability of the software over different operating systems.

Two basic guidelines have driven the assignment of the functionalities among the various processes. First, functionalities with stringent real-time requirements (e.g., packet capture or disk access) are confined within a single process (*capture* and *storage*, respectively). The resources assigned to these processes must be able to deal with worst case scenarios. Other processes instead operate in a best-effort manner (e.g., *query* and *supervisor*) or with less stringent time requirements (e.g., *export*). Second, each hardware device is assigned to a single process. For example, the *capture* process is in charge of the network sniffer, while *storage* controls the disk array.

Another important feature of our architecture is the decoupling between real-time tasks and user driven tasks. This is visualized by the vertical lines in Figure 1. This decoupling allows us to control more efficiently the resources in CoMo and to avoid that a sudden increase in traffic starves query processing, and vice-versa.

We now describe the five main processes that compose the core of CoMo:

- The *capture* process is responsible for the packet capture, filtering, sampling and maintaining per-module state information;
- The *export* process allows long term analysis of the traffic and provides access to additional networking information (e.g., routing tables);
- The *storage* process schedules and manages the accesses to the disks;
- The *query* process receives user requests, applies the query on the traffic (or reads the pre-computed results) and returns the results;
- The *supervisor* process is responsible for handling exceptions (e.g., process failures) and to decide whether to load, start or stop plug-in modules depending on the available resources or on the current access policies.

The *capture* process receives packets from the network card (that could be a standard NIC card accessed via the Berkeley Packet Filter [17], or using dedicated hardware such an Endace DAG card [7]). The packets are passed through a filter that identifies which modules are interested in processing the packets. Then the *capture* process communicates with the modules to have them process the packets and update their own data structures. Note that those data structure may also be maintained by the *capture* process in order to keep the module simple.

[1]The CoMo code is open source and we aim to build an open community of developers in charge of the core components.

Periodically, *capture* polls the data structures updated by the modules and sends its content to the *export* process. These data structures are then ready to be processed again by the modules. As explained earlier, this way, we decouple real-time requirements of the *capture* process that deals with incoming packets at line rate from storage and user oriented tasks. Flushing out the data structure periodically also allows *capture* to maintain limited state information and thus reduce the cost of insertion, update and deletion of the information stored by the modules.

The *export* process mimics the behavior of *capture* with the difference that *export* handles state information rather than incoming packets. Therefore, *export* communicates with the modules to decide how to handle the state information. A module can request *export* to store the information and/or to maintain additional, long term information. Indeed, as opposed to *capture*, *export* does not flush periodically its data. Instead, it needs to be instructed by the module to get rid of any data.

The *storage* process takes care of storing *export* data to the hard disk. The *storage* process is data agnostic and treats all data blocks equally[2]. It can thus focus on an appropriate scheduling of disk accesses and on managing disk space. The *storage* process understands only two type of requests: "store" requests from *export* and "load" requests from *query*.

The *query* process manages users' requests, and if access is granted to the user, it gets the relevant data from disk via the *storage* process and returns the query results to the user. If the requested data is not available on disk, the *query* process can (i) perform the analysis on the packet trace stored on the disk (if the request refers to a period in the past) or (ii) request the initialization of a new module by the *supervisor* process to perform the query computation on the incoming packet stream.

Finally, the *supervisor* monitors the other processes and decides which modules can run based on the available resources, access policies and priorities. The *supervisor* communicates with all the other processes to share information about the overall state of the system.

C. Plug-in Modules

The traffic metrics and statistics are computed within a set of plug-in modules. The modules can be seen as a pair *filter:function*, where the filter specifies the packets on which the function should be performed. For example, if the traffic metric is "compute the number of packets destined to port 80", then the filter would be set to capture only packets with destination port number 80, while the function would just increment a counter per packet. Note that all modules do not necessarily compute statistics. Modules can simply transform the incoming traffic, like for example transform a set of packets in a flow record.

The core processes are responsible for running the packet filters and communicate with the modules using a set of callback functions. Actually there are several sets of callback functions, one for each of the core processes (represented by the three columns of white boxes in Figure 1). Going back to the previous example, the *capture* process will use a callback (`update()`) to let the module increment the counter. Then the *export* process will use a different callback (`store()`) to move the counter value to the disk. Also, *export* could use another callback to allow the module to apply, for example, a low pass filter on the counter values. The *Query* process will then use a different callback (`load()`) to retrieve the counter value from disk.

It is important to observer that the core processes are agnostic to the state that each module computes. Core processes just provide the packets to the module and take care of scheduling, policing and resource management tasks.

IV. Querying Network Data

The query engine is the CoMo gateway to the rest of the world. The main function of queries is to request CoMo to export data. The range of data to be exported can vary significantly, from raw packet sequences to aggregated traffic statistics.

The processing of a query can be divided in three steps: (i) validate and authorize the query (as well as its origin), (ii) find and/or process the data and, finally, (iii) send the data back to the requester.

The amount of data stored on a CoMo system can be very large (in the order of 1TB on current prototypes). It is thus desirable to reduce the amount of processing needed to answer a query to a minimum. This is, indeed, the main purpose of the CoMo modules: pre-compute data to minimize the cost of processing incoming queries.

In CoMo we identify three types of queries:

- *Static queries* defined in the system configuration together with the relevant module. This kind of query will appear in the form `'send-to <IP address>:<port>'` and follow a push information model, i.e. as the module computes the metric on the traffic stream, data is sent to the specified IP address. It is clear that this type of queries does not require any explicit support from the query engine.
- *On-demand queries* explicitly specify the relevant module. This could happen in two ways: indicating the name of the module in the query itself or sending directly the module source code. The query would then have to indicate the packet filter to be applied to the packet stream and the time window of interest. The response consist in the output of the module. On reception of this query, the CoMo system has to authenticate the module and the requester, and then figure out if the same module has already been installed. If the same module has been running during the time of interest, then this query revert to a static query. Otherwise, it requires the module to run on the stored packet-level trace[3] with an obvious impact on the query response time.

[2]A viable alternative is to allow *storage* to filter some of the data blocks as early as possible to reduce data movement and processing, following an approach similar to Diamond [12].

[3]Note that every CoMo system is supposed to keep a packet-level trace at all times. The duration of the packet trace will depend on the available storage space and the link speed.

- *Ad-hoc queries* have no explicit module defined. These queries are written in a specific query language and code for the modules is generated on the fly. A similar approach is followed in systems like Aurora [3], TelegraphCQ [4], Gigascope [6] or IrisNet [10]. The caveat of this kind of approach is that it cannot exploit existing modules that have been already running on the packet stream. One alternative would be to have all modules, even custom-made one, to specify the computation they are performing using the query language and then compare the received query with all the modules to find a module that is computing a *super-set* of the response. Then, compile the new query using as input the super-set as well. Clearly, this method cannot be applied to modules that compute metrics that cannot be expressed in the query language. In that case, the user has no other option than to write a custom module and use the on-demand query approach.

One of the big challenges of the system is to manage queries both in terms of computing resources and data transfers. Processing a query should not jeopardize the ability of the system to collect the packet-level trace without losses. Thus, it is important for the system to predict the usage of resources of a query before scheduling it to run. This can be easily done for static queries (the module is known in advance to the system), while it is more challenging for other type of queries.

Moreover, the system should take into account the priority of queries and that of all the other modules in the system that are pre-processing data for future queries. Indeed, some queries might be urgent (e.g., real-time security related information) and it might be appropriate not to delay the computation of the query (at the cost of other CoMo tasks).

Finally, it is most probable that multiple CoMo systems will be present in a network. Consequently, more than one system may be needed to answer a query or multiple systems may coordinate to identify the most appropriate subset of them over which to run the query. For example, a system could be more appropriate than others depending on the relevance of the monitored traffic for the query, (e.g., when tracking a denial of service attack) or the current system load. Moreover, some of these systems will have data to export, some will not. Therefore, an additional challenge is to design a query management system that minimizes the search and export cost in the context of distributed queries. For this specific challenge, we will also investigate innovative solutions for query management proposed in the context of sensor networks [1], [16].

V. Resource Management

Managing system resources (i.e. CPU, hard disk, memory) in CoMo is challenging because of two conflicting system requirements. On the one hand, the system should be open to users to add plug-in modules for new traffic metrics and to query the system. On the other hand, the system needs to be always available, guarantee a minimum performance level and compute accurately the metrics that are of interest at a given time.

We divide the major causes of resource consumption in three categories: traffic characteristics, measurement modules, and queries. In the following we will address each of them separately.

Traffic characteristics. Resource utilization depends heavily on the incoming traffic. Unfortunately, periods of high traffic load are impossible to predict. Moreover the traffic characteristics that overloads the system largely depends on the modules (and filters) that are running at the time. Indeed, incoming traffic impacts the modules' activity, not simply the activity of the core processes. For example, a module could be idle for long periods and then have a burst of activity when packets hit its filter. A module computing flow statistic would become very greedy in case of DoS attack. Moreover, the amount of computations per packet will often depend on the packet content (e.g., IDS). The characteristics of the traffic, together with the nature and number of active modules also impact directly the storage. For reasons listed above, there will be periods where large amounts of data will have to be stored to disk.

Modules. In order to control the resources used by modules, we have defined a set of constraints on their capabilities:

- *Modules can be started and stopped at any time*. Modules are prioritized based on resource consumption and managed accordingly. We also rely on the fact that most module can be run off-line at a later time on the packet trace.
- *Modules have access to a limited set of system calls*. They cannot allocate memory dynamically and have no direct access to any I/O device. This allows us to maintain the control over the resource usage within the core processes and, at the same time, it keeps the module source code simpler.
- *Modules do not communicate with each other*. Modules are independent. They do not share any information with other modules. This constraint simplifies resource management, although it introduces redundancies. For example, one module cannot pass pre-processed information to another module. This way, different modules might perform the same computations on the same packets.

These three module requirements allow the CoMo system to regulate the amount of computing resources used at a given time. However, the decision on whether to start or stop a module depends on two parameters: (i) the measured resource usage of the module and (ii) the relevance of the metric computed by the module. The optimization of the sets of modules run at a given time is also a challenging issue. It is very difficult to estimate the resource usage (that depends on incoming traffic) and the relevance of a given metric can also vary significantly over time.

Queries. Queries can come at anytime and cause resource consumption for authentication, module insertion or removal, data processing. The query engine will have to manage the impact of the query processing on the system resources and active modules. The main issue with queries is that they should have higher priority than existing modules. A query indicate

that a user exists and is waiting for results, while modules are just pre-computing queries based on the assumption that some users will be interested. On the other hand, in presence of a large number of queries, running them at high priority may force the CoMo system to be a simple packet collector, reducing the usefulness of the modules.

A. Resource control options

As described above, the prediction of resource utilization is almost impossible given the different factors that affect it. Therefore, we have no other option than accurately measuring resource utilization and timely react to overload situations. We need to identify what "knobs" can be turned to control resource utilization.

In CoMo, resource management is threshold-based. The resource usage of a CoMo system is monitored continuously in term of CPU cycles, memory usage and disk bandwidth. If the usage exceeds the pre-defined high-threshold, the following actions can be taken: (i) sample the incoming traffic for certain modules; (ii) stop some modules; (iii) block incoming queries.

The specific action to be taken should depend on the module priorities. As we said earlier, most modules can be delayed and run at a later time on the stored packet trace. The priority of a module should depend on the requested response time of the potential query that needs the data computed by the module. For example, a module computing a metric for anomaly detection should have high priority given that a query for anomalies requires in general a very short response time.

The module priority should then adapt to the query currently running on the system as well as on the historical queries that the system has received. At configuration time, the system administrator will define a static priority for each module that will then vary depending on how often the data processed by a module are actually read.

VI. Security Issues

There is no doubt that the greatest challenge in providing an open monitoring service to users consists in enforcing access policies and safeguarding the privacy of network users.

One static policy applied to all systems is not enough given the large number of different uses that we envision for the monitoring infrastructure. For example, the network operator that owns the monitored link may have complete access to the traffic, including the payload. Other users should instead only be allowed to view the packet header or even just an anonymized versions of it. Finally, some queries could be limited to a subset of the users in order to avoid a constant overload of the system or to increase network traffic (e.g., all queries that require large data transfers).

Hence, a rich and descriptive policy language is needed. The policy should define which modules can be plugged into the system, which users can plug modules in, and which users can post queries to the system together with the type of queries they can post.

A. Access policies

The only access points to the system for users is the plug-in interface where new modules are added and the query interface. However, note that a query always results in a module being plugged into the system. Therefore, in the rest of this section we will consider only the case of a user requesting to plug in a module.

The module is described by the two components *filter:function*. It is possible to assign *access request levels* to each component. These access requests levels indicate the privilege level at which the module has to run (and that has to match the user access privileges). For example, a filter that specifies "anonymized packet headers" will have the lowest access request level, while a filter that desires to have access to "not anonymized packet payloads" will be marked with the highest access request level.

Assigning access requests level for the *functions* of the modules is a much harder problem. It requires a deep understanding of what the module is computing and storing to disk. The solution that is often adopted is to allow the function to perform any computation but to restrict significantly the filter. NLANR, for example, provides only anonymized packet header traces but does not impose any condition on what user do with the traces [19]. Unfortunately, this approach is not appropriate in general. For example, worm signature detection requires full inspection of the packet stream although the state information it maintains (worm traffic) has little relevance and would certainly have a low access request level.

The approach followed by CoMo is to allow to load on the system only "signed" modules, for which the original developer can be authenticated. Then the module's function will inherit the developer privileges.

User access privileges will initially depend on the CoMo system itself: (i) public access system will allow any user to plug-in modules and query the system; (ii) restricted system where only a subset of the users are allowed to plug-in new modules and the rest of the users can only perform queries on metrics for which a module already exists; (iii) private access, where only a subset of users can plug-in modules and query the system. In the future, we envision that each user will have individual privilege levels that will decide whether a *filter:function* pair is allowed to run on CoMo.

B. Infrastructure Attacks

So far, we have only addressed the security of the data. We now discuss the possible attacks on the monitoring infrastructure. We consider two types of attacks:

Denial of service attacks. Attacks in this class may come in the form of a module that uses a disproportionate amount of resources or that corrupts the data owned by other modules.

The former type of attacks could be dealt directly with the resource manager and the use of "module black list" to forbid a module to run again in the system. Also, the use of a sandbox or of signed modules may help in avoiding this class of attacks and finding out a module's real "intentions". In fact, because modules process incoming traffic, on which we have little if

any control, even a perfectly legal module could be driven into consuming large amount of resources in response to certain input patterns. In general this problem is dealt with by the generic resource control mechanisms discussed in Section V.

The second class of attacks (data corruption) is harder to defend against. One first immediate solution is to provide memory isolation between modules. This can be achieved running modules as separate processes or moving some of the CoMo functionalities in the kernel. This introduces some overhead on the system but would guarantee that two modules will not interfere with each other.

Attack on the access policies. This class of attacks include attacks on the user privileges or on the access request level of a filter or function. For example, one can envision an attack on the packet anonymization scheme that would allow a user with low privileges to run a filter with high access level. An attack on the anonymization scheme could consist in sending carefully-crafted packets to the system and use a module that captures the anonymized version of those packets in an attempt to break the anonymization scheme [26].

VII. Related Work

The list of software and techniques for active monitoring and network performance metrics computation is long. NIMI [21] is the pioneer in the deployment of active monitoring infrastructures, while CAIDA [2] has made available a large set of tools for active monitoring. The IETF IPPM Working Group is also working on the standardization of metrics for active monitoring [15].

The area of passive monitoring is much less rich than active monitoring, mostly because of the deployment constraints of passive monitoring systems (i.e. active monitoring systems can be deployed at the edge of networks, when passive monitors must be deployed inside networks). The first generation of passive measurement equipment has been designed to collect packet headers at line speed on an on-demand basis. This generation of monitoring systems is best illustrated by the OC3MON [19], Sprint's IPMON [8] or NProbe [18] experience. Pandora [20] allows to specify monitoring components and this way provides greater flexibility in specifying the monitoring task. However, it differs from our approach in that it enforces a strict dependency among components and does not allow to dynamically load/unload some components.

Routers also embed monitoring software such as for example Cisco's Netflow [5]. Netflow collects flow level statistics on router line cards. Given the severe power and space constraints on routers, Netflow cannot store large amount of records but it exports all the information it capture to an external collector. This forces network operators to apply aggressive packet sampling (in the order of 0.1%) to reduce the data transfer rate from the routers.

Recently, the database community has approached the problem of Internet measurements. Several solutions have been proposed that deploy stream databases techniques. AT&T's Gigascope [6] is an example of a stream database that is dedicated to network monitoring. The system support a subset of SQL but it is proprietary and no measurement data is made publicly available. Other systems such as Telegraph [4] or PIER [11], IrisNet [10], Aurora [3], PHI [23] address the problem of continuous and distributed queries and as such are very relevant to CoMo.

CoMo is different from the existing passive monitoring projects in various ways. First, CoMo is designed to be open to implement a wide and dynamic set of traffic metrics. CoMo intends (1) to use a stream database approach to manage network data and queries, (2) to develop a complex resource management and access policy mechanism, (3) to follow a distributed approach to network monitoring (i.e. CoMo systems can cooperate to monitor a network more efficiently).

The successful design of CoMo requires the convergence of successful system design (integration, resource control, security) and challenging fundamental research in areas such as data summarization, query and data management, distributed sampling.

VIII. Conclusion

We have presented the architecture of an open system for passive network monitoring. We have justified the design choices and indicated the three main open issues that are crucial for the success of the monitoring infrastructure: query engine, resource management and security of the system.

There is a number of other issues that have not been addressed in this paper but are currently under investigation: (i) coordination of multiple CoMo system to respond to a query or balance the computation load; (ii) optimal placement of CoMo system as well as modules to guarantee visibility on the traffic even in presence of network failures or re-routing events; (iii) use of sampling for reducing the load on the system in a controlled fashion; (iv) how to port the current architecture to other hardware systems, such as routers or network processors.

Acknowledgments

We would like to thank Larry Huston, Pere Barlet, Euan Harris, Lukas Kencl, and Timothy Roscoe for their help, feedback and comments on this work.

References

[1] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of ACM Sigcomm*, Sept. 2004.
[2] CAIDA: Cooperative Association for Internet Data Analysis. http://www.caida.org.
[3] D. Carney et al. Monitoring streams - a new class of data management applications. In *Proceedings of VLDB*, 2002.
[4] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing of an uncertain world. In *Proceedings of CIDR*, 2003.
[5] Cisco Systems. NetFlow services and applications. White Paper, 2000.
[6] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of ACM Sigmod*, June 2003.
[7] Endace. http://www.endace.com.
[8] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, R. Rockell, D. Moll, T. Seely, and C. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 2003.
[9] GEANT. http://www.dante.net.
[10] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An architecture for a world-wide sensor web. *IEEE Pervasive Computing*, 2(4), Oct.

[11] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of VLDB*, 2003.
[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Stayanarayanan, G. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Usenix FAST*, Mar. 2004.
[13] Internet2. http://www.internet2.org.
[14] IP Flow Information eXport Working Group. Internet Engineering Task Force. http://www.ietf.org/html.charters/ipfix-charter.html.
[15] IP Performance Metrics Working Group. Internet Engineering Task Force. http://www.ietf.org/html.charters/ippm-charter.html.
[16] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of the ACM Sensys*, Nov. 2003.
[17] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *USENIX Winter*, Jan. 1993.
[18] A. Moore, J. Hall, E. Harris, C. Kreibech, and I. Pratt. Architecture of a network monitor. In *Proceedings of Passive and Active Measurement Workshop*, Apr. 2003.
[19] NLANR: National Laboratory for Applied Network Research. http://www.nlanr.net.
[20] S. Patarin and M. Makpangou. Pandora: A flexible network monitoring platform. In *Usenix*, 2000.
[21] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large-scale internet measurement. *IEEE Communications*, 36(8), 1998.
[22] RIPE Rèseaux IP Europèens. http://www.ripe.net.
[23] T. Roscoe and J. Hellerstein. Phi lrp white paper. Technical report, Intel Research, Sept. 2004.
[24] tcpdump/libpcap. http://www.tcpdump.org.
[25] University of Oregon Route Views Project. http://www.routeviews.org.
[26] J. Xu, J. Fan, M. Ammar, and S. Moon. Prefix-preserving IP address anonymization: Measurement-based security evaluation and a new cryptography-based scheme. Nov. 2002.