

Fast Prototyping of Network Data Mining Applications

Gianluca Iannaccone
Intel Research Cambridge

Abstract. Developing new tools to analyze network data is often a complex and error-prone process. Current practices require developers to possess an in-depth understanding of the original data sets and to develop ad-hoc software tools to first extract the relevant information from the data and then implement the internals of the new algorithm. This development process results in long delays during the analysis of the data and in the production of software that is often hard to reuse, debug or optimize. We present the design and implementation of CoMo, a system for fast prototyping network data mining applications. CoMo provides an abstraction layer both for the network data as well as for the hardware architecture used to collect and process the data. This allows developers to focus on the correctness of the implementation of their analysis tools while the system makes the tool amenable to optimization when running on different hardware architectures. In this paper we discuss CoMo’s design challenges, our solution to address them and evaluate the resulting software in terms of performance, flexibility and ease of use.

1 Introduction

Implementing, prototyping, and debugging network measurement applications is a time consuming and error-prone effort. A lengthy development process needs to take place to implement new analysis methods on existing monitoring systems. First, the developer needs to gain a deep understanding of the network data set. The data model will depend on the device that is capturing the traffic, the software used for storing the data, the transport media, etc. Then, the developer will build a series of ad-hoc tools to extract the relevant information from the data sets and verify the resolution and accuracy of the data. Finally, the analysis tool is implemented and its performance is measured (the original goal!).

One striking characteristic of this development process is the time spent to implement tools that are not directly required for the measurement application. This has several undesired consequences. The ad-hoc tools for extracting the relevant information are usually not “designed to last” and certainly not to be reused by other analysis tools that require similar information. To speed up development, the application is often built as a monolithic piece of software tailored to the specific capture device, operating system and architecture the developer is using at the moment. Overall, this leads to applications that are hard to distribute, hard to debug, hard to reuse by other researchers and that

require to be rewritten from scratch if a different capture device or hardware architecture is used.

In this paper we explore the design and implementation of CoMo, a platform that allows the fast prototyping of network data mining applications. We refer to network data mining applications in a very broad sense: all computations to be performed on a network data stream that cannot be easily expressed using common declarative query languages (such as SQL, for example). Luckily, the measurement community gives us many examples of such applications: worm fingerprinting [13], application identification [9], TCP sender state tracking [8], and many more.

The CoMo architecture presents an abstraction layer for the capture devices as well as for the hardware architecture of the nodes that are running the analysis. Developers can implement new algorithms for processing network data streams (real time or off line) without any explicit knowledge of the internals of the monitoring system, transport media as well as memory and storage organization. CoMo follows a classical modular approach that has proven to be successful in similar contexts [10, 2]. Users write a single (simple) plug-in module in C, making use of a feature-rich API provided by the core system. Each module uses a common data model and specifies the information of interest (together with its resolution and accuracy) and the system identifies if that information is available. The end result is that the developers can focus on the implementation of their applications while CoMo makes sure that it can run using different capture devices or hardware without any modification to the code.

We have implemented a full-featured CoMo prototype and released the code under a BSD open source license [16]. The next section presents some related work, while Section 3 and 4 describe the key design concepts and architectural components of the system. Section 5 reports on the implementation status, our experiences with the system and some preliminary performance results. Section 6 concludes the paper.

2 Related Work

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 `tcpdump` and the OC3MON [11] and Sprint’s IPMON [5] experience. These systems are basically data warehousing efforts with a centralized data collection and application-specific analysis tools.

Recently, the database community has approached the problem of Internet measurements and proposed several solutions. Gigascope [4] is an example of a stream database that is dedicated to network monitoring. It supports a subset of SQL with the addition of a “window” operator to perform continuous queries. Other systems in the same space are Aurora [2] and TelegraphCQ [3].

However, the works in the literature that are most closely related to CoMo are Pandora [12], Flame [1] and Scriptroute [14]. Pandora [12] provides flexibility in specifying a monitoring task by allowing the developer to construct a graph of dependencies between a set of monitoring components. However, it enforces a

strict dependency among the monitoring components requiring good knowledge of the underlying system and does not allow to dynamically load/unload the components. Flame [1] lets users specify passive monitoring functions in kernel modules written with a safe language. The main focus in that work is to manage system resources avoiding that malicious or incorrect modules affect the performance of the system. Finally, in the active monitoring area, Scriptroute [14] allows user to define active measurements writing simple Ruby scripts. The goals of Scriptroute are similar to CoMo: foster the deployment of a common platform for network measurements.

3 Design concepts

In any system for fast prototyping the main challenge is to find the right balance between ease of use, expressiveness and performance. Ease of use refers to the data and programming models used to prototype new tools: they have to be familiar to the developers in order to lower the entry barrier. The trade off between expressiveness and performance is instead something all designers face when building a new system. Restricting the type of computations a user can perform allows designers more freedom when optimizing the system. On the other hand, more flexibility to the users means that the system has much less prior information on what can be optimized and how. Keeping these challenges in mind, we can summarize CoMo's design concepts as follows:

Network data model. The first task for developers is to understand the traffic data collected and extract the relevant information. This usually leads developers to define ad-hoc data models for all the intermediate results that are tailored to the specific working environment (i.e., capture device, storage architecture, etc.). Our goal is to find a unified network data model that fits all network data mining applications. Clearly, this data model will have to be extensible (we cannot know in advance all possible application requirements) and able to answer questions on data quality and lineage.

When dealing with network traffic a natural starting point for the data model is the IP packet. All developers are very familiar with the semantics of the information contained in an IP packet and transport layer protocols. We can then add link layer information (e.g., Ethernet, 802.11, HDLC, etc.), physical layer information if present (e.g., signal to noise ratio for wireless devices) and some meta-data information in front of every packet (e.g., packet timestamps, routing information, etc.). The data model is then completed with a meta description of the entire packet stream. This description provides information on how the packet stream has been processed in the past (e.g., packet filters, anonymization processes, etc.) and what information is actually available in the packets.

The challenge then is how to translate the *raw* information collected from the data source (that may be an Ethernet link, a wireless interface or a NetFlow stream from a router, for example) to the common data model. CoMo uses *software sniffers* to interface with raw data streams, convert, and merge them into a single unified data stream. This way the developer just needs to know one

data format and select the most appropriate sniffer for the raw source of interest. We will describe the operations of the sniffers in more details in Section 4.2.

Programming model. The need for expressiveness forces us to discard common declarative query languages (e.g., SQL) because too restrictive. Furthermore, we need to pick a language that is of common use among network researchers and application developers. For this reason we have chosen to use the C programming language. The system then follows a classical modular approach. Developers implement an *application module* that has to contain a set of predefined entry points (“callbacks”). The only information the developer needs is the semantic that the rest of the system associates with each of the callbacks. Some of the callbacks are self-explanatory (e.g., `store()`, `load()`, `update()`) but others requires understanding the abstractions CoMo provides. Therefore, the challenge is to identify the minimum set of callbacks that suffice most applications. Too many callbacks would affect the simplicity and ease of use of the system while too few callbacks would severely limit its flexibility.

Data management. The per-packet monitoring cost is often dominated by the cost of I/O operations (disk reads/writes or data transfers from the capture devices). In CoMo there are two types of data sources: the sniffers that collect the incoming data streams and the application modules that store their own results. The rest of the system does not consume or generate data but is in charge of moving data around, i.e. connect modules to sniffers or to other modules.

The application modules internally maintain a private data representation but provide an interface to convert their data in other formats. In particular a module may “regenerate” a packet stream (using our network data model) that is an approximation of the original packet stream it has processed. This way the core system can have a module use the results of another module without having to know *what* the two modules compute but just the packet stream they are *capable* of generating or receiving.

Hardware abstraction. Declarative languages are often used to decouple *what* computations are performed on a data set from *how* they are actually implemented. Although we do not make use of declarative languages, we still desire to inject enough “structure” into the application modules to allow a certain degree of optimization for the hardware architecture without requiring any explicit intervention of the developer.

We define three abstraction layers: one for the CPU, one for the memory and one for the storage. This way we abstract away the hardware to relieve the developers of laying out the data on disk, handling memory allocations and synchronize multiple processes when more than one CPU is available. The application modules use a pre-defined set of callbacks and a dedicated memory and storage interface. The developer is not aware of *where* a given callback is running nor *how* the data is organized on disk or in memory. Each callback is called by the system and receives as input parameter a packet and specific memory regions to operate with. Performance optimization is obtained by carefully partitioning the different callbacks among different processes, CPUs or even entire systems.

4 Architecture

Having outlined the major design concepts behind CoMo, we now present a description of the current system. In the interest of space, we limit this description to the major system components but we refer the interested reader to the documentation and the source code available in [16].

4.1 Data flow

The data flow across the system is illustrated in Figure 1. The white boxes indicate the application modules while the gray boxes represent CoMo core processes. On the left side, the system receives a set of raw data streams that are converted by the sniffers and then merged in a unified packet stream. On the other side, the system receives user requests to retrieve the data computed by an application over a user-specified time window.

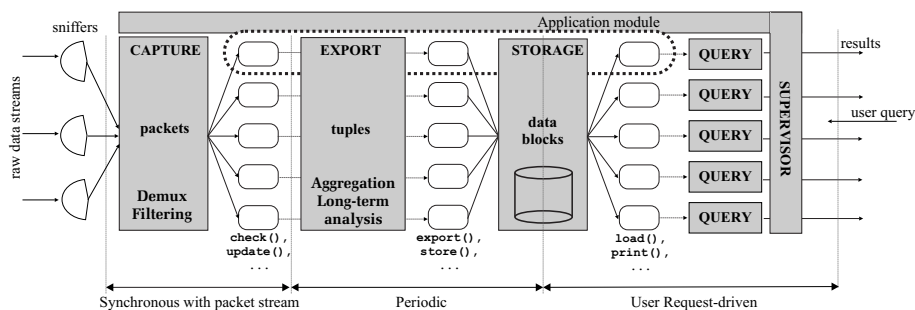


Fig. 1. Data flow in the CoMo system

The core processes are in charge of all data movements within the system and perform operations common to all modules (e.g, packet filtering, storage, memory allocation). The modules are seen from these processes just as a set of APIs used to transform the data streams as they traverse the system. The core processes therefore maintain a tight control on the data path to guarantee an efficient use of the system resources (i.e, CPU, memory, I/O bandwidth).

The resources and functionalities of the system are strictly partitioned among the multiple processes following two major guidelines: (i) functions with stringent real-time requirements are confined within dedicated processes (e.g., CAPTURE deals with online packet capture and STORAGE manage disk reads and writes); (ii) each hardware device is assigned to a single process to avoid competition on the resources and allow a better scheduling¹ (e.g., STORAGE is the only process to access the disk array).

The various processes communicate and synchronize using a unidirectional message passing system. The strict partitioning of resources and functionalities

¹ We discarded the alternative of delegating to the operating system the scheduling of the modules because when running real time on high speed links we require a more fine-grained control on all system activities.

allows to deploy CoMo in a variety of environments. It can be deployed on a single machine or a cluster of nodes with dedicated hardware (e.g, network processors or active storage [7]) without requiring any modification to the application modules. Finally, another important feature of our architecture is the decoupling between real-time tasks (e.g., tasks that have to be synchronized with incoming data streams) and user driven tasks (e.g., retrieving data computed by modules). This is visualized by the vertical lines in Figure 1. This decoupling allows us to control and predict more efficiently the resource usage in CoMo allowing to absorb a sudden increase in incoming traffic without compromising overall system performance.

4.2 Data management

As mentioned before, the sniffers and the application modules are the two data sources in CoMo. The modules and the end-users are the data consumers. Sources and consumers instruct the core processes on “how to connect” them to each other by describing their capabilities and needs in terms of the packet stream they can process or generate. The description informs of which packet fields can be provided in the stream. For example, the sniffer that processes Netflow data streams provides a description that specifies that each packet will contain the 5 tuple information, an averaged packet size, a timestamp with a granularity that is equal to the maximum duration of the flows², plus informs the module of the presence of some additional routing information in the packet headers (e.g, AS number, prefix length). Then the sniffer can generate a sequence of packets that if re-processed by NetFlow would give the same set of flow records³. Another example is the Snort application module implemented in CoMo. It can generate packet streams that consists of all packets that matched a given Snort rule. The description would have to add the rule applied to the packet stream to the original description of the packets (to maintain data lineage information).

4.3 Application modules

Application 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 over 30 second intervals”, 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 and store a value every 30 seconds.

The filter is described in the module configuration and run by the CAPTURE process. The function is instead expressed in a set of callback functions that are called by the CAPTURE, EXPORT or QUERY processes. Table 1 provides a summary of the callbacks available to a module and the process that uses them.

² Current NetFlow implementations define an activity timer that sets an upper bound on the maximum duration of the flow before being exported.

³ The sequence of packets may contain every packet that was in the individual flow or just the first packet with the count in the meta-header of how many packets it actually represents.

CAPTURE callbacks	QUERY callbacks
<code>check()</code> : Stateful filters	<code>load()</code> : Load records from disk
<code>update()</code> : Packet processing	<code>print()</code> : Format records
EXPORT callbacks	SUPERVISOR callbacks
<code>export()</code> : Process records sent by CAPTURE	<code>init()</code> : Initialize the module
<code>store()</code> : Store records to disk	<code>replay()</code> : Generate a packet stream
<code>action()</code> : Decides what to do with a record	

Table 1. Summary of the callbacks. The `update()`, `store()` and `load()` callbacks are mandatory. The others are optional.

For each packet that matches the filter, CAPTURE uses `check()` and `update()` to maintain the data structures managed internally by the module (the memory manager is provided by the core system to avoid the direct use of the `malloc/free` interface).

Periodically, CAPTURE flushes the data structures of all modules and sends their contents to the EXPORT process(es). Flushing out the data structure periodically allows CAPTURE to maintain limited state information and thus reduce the cost of insertion, update and deletion of the information stored by the modules. It also allows to pipeline the function executed by a module to improve the performance with multi processor systems. For example, a flow classification can be pipelined by having CAPTURE perform the first stage of sorting the packets while EXPORT maintains the data structure long term, expires and stores the records.

Once the packet stream reaches the EXPORT process it has already undergone a first transformation (i.e., the only data available is the data maintained by the `update()` callback). The EXPORT maintains data structures for the application module (`export()`) and stores them on disk (`store()` callback). As opposed to CAPTURE, EXPORT uses the module callbacks to know when to discard or store an entry (`action()`).

The STORAGE process behaves like a storage server. It treats all data equally as data blocks and is in charge of scheduling disk accesses and managing disk space. The QUERY process is instantiated when a user request is received and uses the `load()`, `print()` callbacks to return the application results to the user. Finally, SUPERVISOR monitors the other processes and decides which modules can run based on the available resources, access policies and priorities. It also uses the `replay()` callback to connect multiple modules.

5 Implementation

The implementation of a full featured prototype of the CoMo system is publicly available under BSD open source license [16]. The current version is a user-space implementation that runs under Linux 2.4.x and 2.6.x, FreeBSD 4.x and 5.x and Microsoft Windows (with Cygwin environment). The code runs also on ARM architectures and it has been ported to the StarGate platform [15].

We have implemented several sniffers to process data from NIC cards (using BPF or `libpcap`), flow-tools file (NetFlow), Endace DAG capture cards and trace

files, Sysconnect cards and Prism-II based 802.11 wireless cards. We have also implemented a sniffer for other CoMo systems that allows an application module to retrieve data from other nodes to enable distributed network data mining applications.

5.1 Experiences

A small set of application modules has been implemented as proof of concept of the flexibility and ease of use of CoMo. Some of the modules perform simple computations on the packet streams (e.g., utilization, application breakdown, active connections, top destinations) while others are more complex (e.g., kismet-like passive detection of 802.11 networks, full featured Snort intrusion detection module). The set of callbacks has proven to be enough to express this diverse set of applications. Furthermore, the implementations are rather simple thanks to the rich API presented by the core system. For example, the kismet-like module consist of 127 “semicolons” in C while the module that computes the top destinations over a time window counts 64 semicolons. The actual source code is available at [16].

The configuration of the system is also straightforward. The following lines are needed to set up a sniffer for flowtools files, a module that computes the top 20 TCP destination over 5 minutes windows and to instruct STORAGE to maintain the last 5GB of results:

```
sniffer flowtools "/path/flow-tools/ft*"
module "top destinations"
  source "topdest.so"
  filter "tcp"
  args "n=20 interval=300s"
  streamsize 5GB
end
```

Querying the system via the `print()` callback has also proven extremely flexible. We have defined an HTTP interface so that a query can be posted as follows: `http://<address>:<port>/?module=<name>&time=-5m:0`. This query returns the results of the module for the past 5 minutes.

5.2 Evaluation

We have tested our implementation on three different network environments: (i) the Gigabit Ethernet access link to the Intel Research Cambridge lab using an Endace DAG card; (ii) a packet trace collected using a Sysconnect card and stored in pcap format; (iii) the NetFlow v5 flow records collected from the UK router in the GEANT European academic backbone network [6].

We consider three different modules that exercise different components of the systems. The first application (“basic”) is very lightweight. It processes every packet and maintains a count of the bytes and packets observed every 5 minutes. The second application (“cpu”) is more CPU intensive given that it classifies all

flows by destination and every time interval (5s for packet-level traces, 5min for NetFlow records) sorts the flows by byte count and stores the 20 largest. Finally, the third application (“io”) is very I/O intensive given that it stores all packet headers to disk. We let the system run over the first one million packets of the three data sources. We use the processor TSC counter to estimate the number of cycles spent per-packet to perform five different operations: (i) run the core processes (overhead); (ii) process the incoming raw data stream; (iii) run the packet filter; (iv) execute the module callbacks; and (v) save the results to disk (the results reside on a different disk array than the traces).

Figure 2 shows the breakdown of the cycles for the three different sniffers and when the three applications run in isolation or all together. Although we have not yet optimized the code for performance, we can draw some preliminary observations from this figure. First of all, the results confirm the underlying assumption of the CoMo design that the per-packet monitoring cost is dominated by I/O operations (receiving the data stream and saving results to disk).

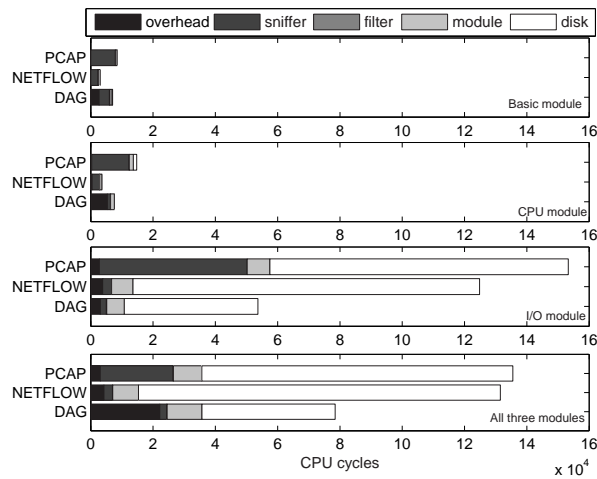


Fig. 2. Breakdown of CPU cycles for different application modules and sniffers

Furthermore, there is a clear advantage in using the sniffer for NetFlow records given that the additional processing required to convert the flow records in a packet stream is offset by the gain in reduced I/O operations.

The flexibility CoMo gives to the developers comes at some cost in performance as illustrated by the overhead introduced in the packet processing. However, from Figure 2 we can see that the overhead is limited and always a minor component of the total per-packet processing cost. The experiments with the DAG card show larger overheads due to the low data rate of that access link. In this case, we pay a high price for polling the card periodically to receive just few packets.

6 Future directions

We have presented the design and implementation of CoMo, a platform for fast prototyping network data mining applications. We have shown that the system provides enough flexibility to implement a wide range of applications that can operate with different network data sets and hardware architectures. On-going research activity is devoted to four main areas: *(i)* performance optimization given that so far we have mainly addressed the problem of ease of use, flexibility and expressiveness; *(ii)* load shedding techniques to avoid system overloads and allow a graceful degradation of the system performance; *(iii)* module isolation to reduce the impact of buggy or malicious modules; *(iv)* distributed network monitoring applications with a module querying other CoMo systems to retrieve intermediate results computed by other modules.

Acknowledgements

The author would like to thank the other members of the CoMo project, including Pere Barlet, Luigi Rizzo, Euan Harris, Davide Vercelli and Richard Gass that have contributed to the software code releases.

References

1. K. Anagnostakis et al. Open packet monitoring on FLAME: Safety, performance and applications. In *Proceedings of IWAN*, 2002.
2. D. Carney et al. Monitoring streams - a new class of data management applications. In *Proceedings of VLDB*, 2002.
3. S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing of an uncertain world. In *Proceedings of CIDR*, 2003.
4. C. Cranor, T. Johnson, O. Spatschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of ACM Sigmod*, June 2003.
5. C. Fraleigh et al. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 2003.
6. GEANT. <http://www.dante.net>.
7. L. Huston et al. Diamond: A storage architecture for early discard in interactive search. In *Usenix FAST*, Mar. 2004.
8. S. Jaiswal et al. Inferring TCP connection characteristics through passive measurements. In *Proceedings of IEEE Infocom*, Mar. 2004.
9. T. Karagiannis, K. Papagiannaki, and C. Faloutsos. BLINC: Multilevel traffic classification in the dark. In *Proceedings of ACM Sigcomm*, Aug. 2005.
10. R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Proceedings of ACM Symposium on Operating Systems Principles*, Dec. 1999.
11. NLANR: National Laboratory for Applied Network Research. <http://www.nlanr.net>.
12. S. Patarin and M. Makpangou. Pandora: A flexible network monitoring platform. In *Usenix*, 2000.
13. S. Singh et al. Automated worm fingerprinting. In *OSDI*, Dec. 2004.
14. N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public internet measurement facility. In *Proceedings of Usenix Conference*, 2003.
15. Stargate platform. <http://platformx.sourceforge.net>.
16. The CoMo Project. <http://como.intel-research.net>.