Realtime High-Speed Network Traffic Monitoring Using ntopng

Luca Deri *†, Maurizio Martinelli*, Alfredo Cardigliano†
IIT/CNR*
ntop†
Pisa, Italy
{deri, cardigliano}@ntop.org, {luca.deri, maurizio.martinelli}@iit.cnr.it

***Abstract*—Monitoring network traffic has become increasingly challenging in terms of number of hosts, protocol proliferation and probe placement topologies. Virtualised environments and cloud services shifted the focus from dedicated hardware monitoring devices to virtual machine based, software traffic monitoring applications.**

This paper covers the design and implementation of ntopng, an open-source traffic monitoring application designed for high-speed networks. ntopng's key features are large networks real-time analytics and the ability to characterise application protocols and user traffic behaviour. ntopng was extensively validated in various monitoring environments ranging from small networks to .it ccTLD traffic analysis.

Keywords—Traffic monitoring, real-time analytics, open-source software, monitoring of virtual and cloud environments.

I. INTRODUCTION

Network traffic monitoring standards such as sFlow [1] and NetFlow/IPFIX [2, 3] have been conceived at the beginning of the last decade. Both protocols have been designed for being embedded into physical network devices such as routers and switches where the network traffic is flowing. In order to keep up with the increasing network speeds, sFlow natively implements packet sampling in order to reduce the load on the monitoring probe. While both flow and packet sampling is supported in NetFlow/IPFIX, network administrators try to avoid these mechanisms in order to have accurate traffic measurement. Many routers have not upgraded their monitoring capabilities to support the increasing numbers of 1/10G ports. Unless special probes are used, traffic analysis based on partial data results in inaccurate measurements.

Physical devices cannot monitor virtualised environments because inter-VM traffic is not visible to the physical network interface. Over the years however, virtualisation software developers have created virtual network switches with the ability to mirror network traffic from virtual environments into physical Ethernet ports where monitoring probes can be attached. Recently, virtual switches such as VMware vSphere Distributed Switch or Open vSwitch natively support NetFlow/sFlow for inter-VM communications [4], thus facilitating the monitoring of virtual environments. These are only partial solutions because either v5 NetFlow (or v9 with basic information elements only) or inaccurate, sample-based sFlow are supported. Network managers need traffic monitoring tools that are able to spot bottlenecks and security issues while providing accurate information for troubleshooting the cause. This means that while NetFlow/ sFlow can prove a quantitative analysis in terms of traffic volume and TCP/UDP ports being used, they are unable to report the cause of the problems. For instance, NetFlow/IPFIX can be used to monitor the bandwidth used by the HTTP protocol but embedded NetFlow probes are unable to report that specific URLs are affected by large service time.

Today a single application may be based on complex cloud-based services comprised of several processes distributed across a LAN. Until a few years ago web applications were constructed using a combination of web servers, Java-based business logic and a database servers. The adoption of cache servers (e.g. memcache and redis) and mapReduce-based databases [5] (e.g. Apache Cassandra and MongoDB) increased the applications' architectural complexity. The distributed nature of this environment needs application level information to support effective network monitoring. For example, it is not sufficient to report which specific TCP connection has been affected by a long service time without reporting the nature of the transaction (e.g. the URL for HTTP, or the SQL query for a database server) that caused the bottleneck. Because modern services use dynamic TCP/UDP ports the network administrator needs to know what ports map to what application. The result is that traditional device-based traffic monitoring devices need to move towards software-based monitoring probes that increase network visibility at the user and application level. As this activity cannot be performed at network level (i.e. by observing traffic at a monitoring point that sees all traffic), software probes are installed on the physical/virtual servers where services are provided. This enables probes to observe the system internals and collect information (e.g. what user/process is responsible for a specific network connection) that would be otherwise difficult to analyse outside the system's context just by looking at packets.

Network administrators can then view virtual and cloud environments in real-time. The flow-based monitoring paradigm is by nature unable to produce real-time information [17]. Flows statistics such as throughput can be computed in flow collectors only for the duration of the flow, which is usually between 30 and 120 seconds (if not more). This means that using the flow paradigm, network administrators cannot

have a real-time traffic view due to the latency intrinsic to this monitoring architecture (i.e. flows are first stored into the flow cache, then in the export cache, and finally sent to the collector) and also because flows can only report average values (i.e. the flow throughout can be computed by dividing the flow data volume for its duration) thus hiding, for instance, traffic spikes.

The creation of ntopng, an open-source web-based monitoring console, was motivated by the challenges of monitoring modern network topologies and the limitations of current traffic monitoring protocols. The main goal of ntopng is the ability to provide a real-time view of network traffic flowing in large networks (i.e. a few hundred thousand hosts exchanging traffic on a multi-Gbit link) while providing dynamic analytics able to show key performance indicators and bottleneck root cause analysis. The rest of the paper is structured as follow. Section 2 describes the ntopng design goals. Section 3 covers the ntopng architecture and its major software components. Section 4 evaluates the ntopng implementation using both real and synthetic traffic. Section 5 covers the open issues and future work items. Section 6 lists applications similar to ntopng, and finally section 7 concludes the paper.

II. NTOPNG DESIGN GOALS

ntopng's design is based on the experience gained from creating its predecessor, named ntop (and thus the name ntop next generation or ntopng) and first introduced in 1998. When the original ntop was designed, networks were significantly different. ntopng's design reflects new realities:

- Today's protocols are all IP-based, whereas 15 years ago many others existed (e.g. NetBIOS, AppleTalk, and IPX). Whereas only limited non-IP protocol support is needed, v4/v6 needs additional, and more accurate, metrics including packet loss, retransmissions, and network latency.
- In the past decade the number of computers connected to the Internet has risen significantly. Modern monitoring probes need to support hundreds of thousand of active hosts.
- While computer processing power increased in the last decade according to the Moore's law, system architecture support for increasing network interface speeds (10/10 Mbps to 10/40 today) has not always been proportional. As it will be later explained it is necessary to keep up with current network speeds without dropping packets.
- While non-IP protocols basically disappeared, application protocols have significantly increased and they still change rapidly as new popular applications appear (e.g. Skype). The association UDP/TCP port with an application protocol is no longer static, so unless other techniques, such as DPI (Deep Packet Inspection) [6] are in place, identifying applications based on ports is not reliable.
- As TLS (Transport Layer Security) [7] is becoming pervasive and no longer limited to secure HTTP, network administrators need partial visibility of encrypted communications.
- The HTTP protocol has greatly changed, as it is no longer used to carry, as originally designed, hypertext only. Instead, it is now used for many other purposes including audio/video streaming, firewall trespassing and in many peer-to-peer protocols. This means that today HTTP no longer identifies only web-related activities, and thus monitoring systems need to characterise HTTP traffic in detail.

In addition to the above requirements, ntopng has been designed to satisfy the following goals:

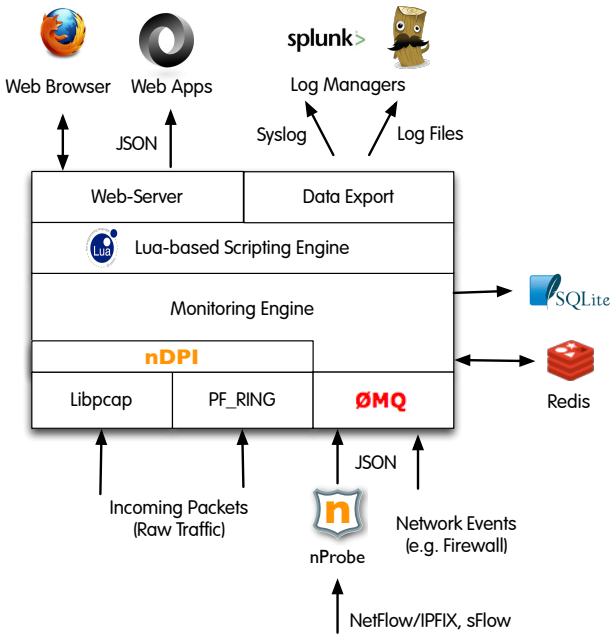
- Created as open-source software in order to let users study, improve, and modify it. Code availability is not a minor feature in networking as it enables users to compile and run the code on heterogeneous platforms and network environments. Furthermore, the adoption of this license allows existing open-source libraries and frameworks to be used by ntopng instead of coding everything from scratch as it often happens with closed-source applications.
- Operate at 10 Gbit without packet loss on a network backbone where user traffic is flowing (i.e. average packet size is 512 bytes or more), and support at least 3 Mpps per core on a commodity system, so that a low-end quad-core server may monitor a 10 Gbit link with minimal size packets (64 bytes).
- All monitoring data must be immediately available, with traffic counters updated in real-time without measurement latency and average counters that are otherwise typical of probe/collector architectures.
- Traffic monitoring must be fully implemented in software with no specific hardware acceleration requirements. While many applications are now exploiting GPUs [8] or accelerated network adapters [9], monitoring virtual and cloud environments requires pure software-based applications that have no dependency on specific hardware and that can be migrated, as needed, across VMs.
- In addition to raw packet capture, ntopng must support the collection of sFlow/NetFlow/IPFIX flows, so that legacy monitoring protocols can also be supported.
- Ability to detect and characterise the most popular network protocols including (but not limited to) Skype, BitTorrent, multimedia (VoIP and streaming), social (FaceBook, Twitter), and business (Citrix, Webex). As it will be explained below, this goal has been achieved by developing a specific framework instead of including this logic within ntopng. This avoids the need of modifying ntopng when new protocols are added to the framework.
- Embedded web-based GUI based on HTML5 and dynamic web pages so that real-time monitoring data can be displayed using a modern, vector-based graphical user interface. These requirements are the foundation for the creation of rich traffic analytics.
- Scriptable and multi-threaded monitor engine so that dynamic web pages can be created and accessed by multiple clients simultaneously.
- Efficient monitoring engine not only in terms of packet processing capacity, but in its ability to operate on a wide range of computers, including low-power embedded systems as well as multi-core high-end servers. Support of low-end systems is necessary in order to embed ntopng into existing network devices such as Linux-based routers. This feature is

to provide a low-cost solution for monitoring distributed and SOHO (Small Office Home Office) networks.

The following section covers in detail the ntopng software architecture and describes the various components on which the application is layered.

III. NTOPNG SOFTWARE ARCHITECTURE

ntopng is coded in C++ which enables source code portability across systems (e.g. X86, MIPS and ARM) and clean class-based design, while granting high execution speed.



1. ntopng Software Architecture.

ntopng is divided in four software layers:

- Ingress data layer: monitoring data can be raw packets captured from one or more network interfaces, or collected NetFlow/IPFIX/sFlow flows after having been preprocessed.
- Monitoring engine: the ntopng core responsible for processing ingress data and consolidating traffic counters into memory.
- Scripting engine: a thin C++ software layer that exports monitoring data to Lua-based scripts.
- Egress data layer: interface towards external application that can access real-time monitoring data.

A. *Ingress Data Layer*

The ingress layer is responsible for receiving monitoring data. Currently three network interfaces are implemented:

- libpcap Interface: capture raw packets by means of the popular libpcap library.
- PF_RING Interface: capture raw packets using the open-source PF_RING framework for Linux systems [10] developed by ntop for enhancing both packet capture and transmission speed. PF_RING is divided in two parts: a kernel module that efficiently interacts with the operating system and network drivers, and a user-space library that interacts with the kernel module, and implements an API used by PF_RING-based applications. The main difference between libpcap and PF_RING, is that when using the latter it is possible to capture/receive minimum size packets at 10 Gbit with little CPU usage using commodity network adapters. PF_RING features these performance figures both on physical hosts and on Linux KVM-based virtual machines, thus paving the way to line-rate VM-based traffic monitoring.
- ØMQ Interface. The ØMQ library [12] is an open-source portable messaging library coded in C++ that can be used to implement efficient distributed applications. In ntopng it has been used to receive traffic-related data from distributed systems. ntopng creates a ØMQ socket and waits for events formatted as JSON (JavaScript Object Notation) [16] strings encoded as "<element id>": "<value>", where <element id> is a numeric identifier as defined in the NetFlow/IPFIX RFCs. The advantages of this approach with respect of integrating a native flow collector, are manyfold :
  - The complexities of flow standards are not propagated to ntopng, because open-source applications such as nProbe [13] act as a proxy by converting flows into JSON strings delivered to ntopng via ØMQ.
  - Any non-flow network event can be collected using this mechanism. For instance, Linux firewall logs generated by netfilter, can be parsed and sent to ntopng just like in commercial products such as Cisco ASA.

Each ingress interface is self-contained with no cross-dependencies. When an interface is configured at startup, ntopng creates a data polling thread bound to it. All the data structures, described later in this section, used to store monitoring data are defined per-interface and are not global to ntopng. This has the advantage that each network interface can operate independently, likely on a different CPU core, to create a scalable system. This design choice is one of the reasons for ntopng's superior data processing performance as will be shown in the following section.

B. *Monitoring Engine*

Data is consolidated in ntopng's monitoring engine. This component is implemented as a single C++ class that is instantiated, one per ingress interface, in order to avoid performance bottlenecks due to locking when multiple interfaces are in use. Monitoring data is organised in flows and hosts, where by flow we mean a set of packets having the same 6-tuple (VLAN, Protocol, IP/Port Source/Destination) and not as defined in flow-based monitoring paradigms where flows have additional properties (e.g. flow duration and export). In ntopng a flow starts when the first packet of the flow arrives, and it ends when no new data belonging to the flow is observed for some time. Regardless of the ingress interface type, monitoring data is classified in flows. Each ntopng flow instance references two host instances (one for flow source and the other for flow destination) that are used to keep statistics about the two peers. This is the flow lifecycle:

• When a packet belonging to a new flow is received, the monitoring engine decodes the packet and searches a flow instance matching the packet. If not found, a flow instance is created along with the two flow host instances if not existing.
• The flow and host counters (e.g. bytes and packets) are updated according to the received packets.
• Periodically ntopng purges flows that have been idle for a while (e.g. 2 minutes with no new traffic received). Hosts with no active flows that have also been idle for some time are also purged from memory.

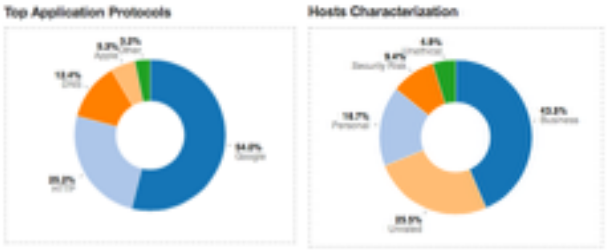
Purging data from memory is necessary to avoid exhausting all available resources and discard information no longer relevant. However this does not mean that host information is lost after data purge but that it has been moved to a secondary cache. Fig. 1 shows that monitoring engine connects with Redis [14], a key-value in-memory data store. ntopng uses redis as data cache where it stores:
• JSON-serialised representation of hosts that have been recently purged from memory, along with their traffic counters. This allows hosts to be restored in memory whenever they receive fresh traffic while saving ntopng memory.
• In case ntopng has been configured to resolve IP address into symbolic names, redis stores the association numeric-to-symbolic address.
• ntopng configuration information.
• Pending activities, such as the queue of numeric IPs, waiting to be resolved by ntopng.

Redis has been selected over other popular databases (e.g. MySQL and memcached) for various reasons:
• It is possible to specify whether stored data is persistent or temporary. For instance, numeric-to-symbolic data is set to be volatile so that it is automatically purged from redis memory after the specified duration with no action from ntopng. Other information such as configuration data is saved persistently as it happens with most databases.
• Redis instances can be federated. As described in [15] ntopng and nProbe instances can collaborate and create a microcloud based on redis. This microcloud consolidates the monitoring information reported by instances of ntopng/ nProbe in order to share traffic information, and effectively monitor distributed networks.
• ntopng can exploit the publish/subscribe mechanisms offered by redis in order to be notified when a specific event happens (e.g. a host is added to the cache) and thus easily create applications that execute specific actions based on triggers. This mechanism is exploited by ntopng to distribute traffic alerts to multiple consumers using the microcloud architecture described later on this section.

In ntopng all the objects can be serialised in JSON. This design choice allows them to be easily stored/retrieved from redis, exported to third party applications (e.g. web apps), dumped on log files, and immediately used in web pages though Javascript. Through JSON object serialisation it is possible to migrate/replicate host/flow objects across ntopng instances. As mentioned above, JSON serialisation is also used to collect flows from nProbe via ØMQ and import network traffic information from other sources of data.

In addition to the 6-tuple, ntopng attempts to detect the real application protocol carried by the flow. For collected flows, unless specified into the flow itself, the application protocol is inferred by inspecting the IP/ports used by the flows. For instance, if there is a flow from a local PC to a host belonging to the Dropbox Inc network on a non-known port, we assume that the flow uses the dropbox protocol. When network interfaces operate on raw packets, we need to inspect the packets' payload. ntopng does application protocol discovery using *nDPI* [18], a home-grown GPLv3 C library for deep packet inspection. To date nDPI recognises over 170 protocols including popular ones such as BitTorrent, Skype, FaceBook, Twitter[1], Citrix and Webex. nDPI is based on an a protocol-independent engine that implements services common to all protocols, and protocol-specific dissectors that analyse all the supported protocols. If nDPI is unable to identify a protocol based on the packet payload it can try to infer the protocol based on the IP/port used (e.g. TCP on port 80 is likely to be HTTP). The library is designed to be used both in user-space inside applications like ntopng and nProbe, and in the kernel inside the Linux firewall. The advantage of having a clean separation between nDPI and ntopng is that it is possible to extend/modify these two components independently without polluting ntopng with protocol-related code. As described in [19], nDPI accuracy and speed is comparable to similar commercial products and often better than other open-source DPI toolkits.

In addition to DPI, ntopng is able to characterise traffic based on its nature. An application's protocol describes how data is transported on the wire, but it tells nothing about the nature of the traffic.



2. Application Protocol Classification vs. Traffic Characterisation

To that end ntopng natively integrates Internet domain categorisation services freely provided to ntopng users by http://block.si. For instance, traffic for cnn.com is tagged as

[1] Please note that technically FaceBook is HTTP(S) traffic from/to FaceBook Inc. servers. This also applies to Twitter traffic. However nDPI assigns them a specific application protocol Id in order to distinguish them from plain HTTP(S) traffic.

"News and Media", whereas traffic for FaceBook is tagged as "Social". It is thus possible to characterise host behaviour with respect to traffic type, and thus tag hosts that perform potentially dangerous traffic (e.g. access to sites whose content is controversial or potentially insecure) that is more likely to create security issues. This information may also be used to create host traffic patterns that can be used to detect potential issues, such as when a host changes its traffic pattern profile over time; this might indicate the presence of viruses or unwanted applications. Domain categorisation services are provided as a cloud-service and accessed by ntopng via HTTP. In order to reduce the number of requests and thus minimise the network traffic necessary for this service, categorisation responses are cached in redis similar to the IP/host DNS mapping explained earlier in this section.

C. Scripting Engine

The scripting engine sits on top of the monitoring engine, and it implements a Lua-based API for scripts that need to access monitoring data. ntopng embeds the Lua JIT (Just In Time) interpreter, and implements two Lua classes able to access ntopng internals.

- interface: access to interface-related data, and to flow and host traffic statistics.
- ntop: it allows scripts to interact with ntopng configuration and the redis cache.

The scripting engine decouples data access from traffic processing through a simple Lua API. Scripts are executed when they are requested though the embedded web server, or based on periodic events. ntopng implements a small cron daemon that runs scripts periodically with one second granularity. Such scripts are used to perform periodic activities (e.g. dump the top hosts that sent/received traffic in the last minute) as well data housekeeping. For instance every night at midnight, ntopng runs a script that dumps on a SQLite database all the hosts monitored during the last 24 hours; this way ntopng implements a persistent historical view of the recent traffic activities.

The clear separation of traffic processing from application logic has been a deliberate choice in ntopng. The processing engine (coded in C++) has been designed to do simple traffic-related tasks that have to be performed quickly (e.g. receive a packet, parse it, update traffic statistics and move to the next packet). The application logic instead can change according to user needs and preferences and thus it has been coded with scripts that access the ntopng core by means of the Lua API. Given that the Lua JIT is very efficient in terms of processing speed, this solution allows users to modify the ntopng business logic by simply changing scripts instead of modifying the C++ engine.

```
dirs = ntop.getDirs()
package.path = dirs.installdir .. "/scripts/lua/
modules/?.lua;" .. package.path
require "lua_utils"
sendHTTPHeader('text/html')
print('<html><head><title>ntop</title></
head><body>Hello ' .. os.date("%d.%m.%Y"))
print('<li>Default ifname = ' ..
interface.getDefaultIfName()
```

3. Simple ntopng Lua Script

When a script accesses an ntopng object, the result is returned to the Lua script as a Lua table object. In no case Lua references C++ object instances directly, thus avoiding costly/ error-prone object locks across languages. All ntopng data structures are lockless, and Lua scripts lock C++ data structures only if they scan the hosts or flows hash. Multiple scripts can be executed simultaneously, as the embedded Lua engine is multithreaded and reentrant.

D. Egress Data Layer

ntopng exports monitoring data through the embedded HTTP server that can trigger the execution of Lua scripts. The web GUI is based on the Twitter Bootstrap JavaScript framework [20] that enables the creation of dynamic web pages with limited coding. All charts are based on the D3.JS [25] that features a rich set of HTML5 components that can be used to represent monitoring data in an effective way.

4. ntopng HTML5 Web Interface

The embedded web server serves static pages containing JavaScript code that triggers the execution of Lua scripts. Such scripts access ntopng monitoring data and return their results to the web browser in JSON format. Web pages are dynamically updated every second by the JavaScript code present in the web pages, that requests the execution of Lua scripts.

As stated earlier in this section, ntopng can manipulate JSON objects natively, thus enabling non-HTML applications to use ntopng as a server for network traffic data as well. Through Lua scripts, it is possible to create REST-compliant (Representational State Transfer) [21] Web services on top of ntopng.

Another way to export monitoring data from ntopng, is by means of log files. With the advent of high-capacity log processing applications such as Splunk and ElasticSearch/ Logstash, ntopng can complement traditional service application logs with traffic logs. This allows network administrators to correlate network traffic information to service status. Export in log files is performed through Lua scripts that can access the monitoring engine and dump data into log files or send it via the syslog protocol [22], a standard for remote message logging.

IV. EVALUATION

ntopng has been extensively tested by its users in various heterogeneous environments. This section reports the results of some validation tests performed on a lab using both synthetic and real traffic captured on a network. The tests have been performed using ntopng v.1.1.1 (r7100) on a system based on a

low-end Intel Xeon E3-1230 running at 3.30 GHz. ntopng monitors a 10 Gbit Intel network interface using PF_RING DNA v.5.6.1. The traffic generator and replay is *pfsend*, an open-source tool part of the PF_RING toolset. In case of real traffic, pfsend has reproduced in loop at line rate the pcap file captured on a real network. In the case of synthetic traffic, pfsend has generated the specified number of packets by forging packets with the specified hosts number. Please note that increasing the number of active hosts also increases the number of active flows handled by ntopng.

The table below reports the test with traffic captured on a real network and reproduced by pfsend at line rate

Hosts Number	*PPS*	*Gbps*	*CPU Load*	*Packet Drops*	*Memory Usage*
350	1.735.000	10	80%	0%	27 MB
600	1.760.000	10	80%	0%	29 MB

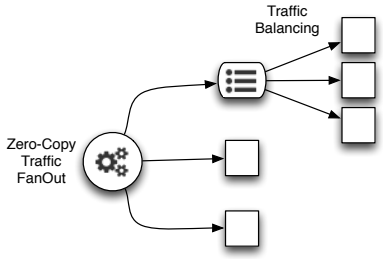
5. Tests Using Real Traffic (Average Packet Size 700 bytes)

The result shows that ntopng is able to monitor a fully loaded 10 Gbit link without loss and with limited memory usage. Considered that the test system is a low-end server, this is a great result, which demonstrates that it is possible to monitor a fully loaded link with real traffic using commodity hardware and efficient software. Using synthetic traffic we have studied how the number of monitored hosts affects the ntopng performance. Increasing the cardinality of hosts and flows, ntopng has to perform heavier operations during data structure lookup and periodic data housekeeping.

Packet Size	**64 bytes**	**128 bytes**	**512 bytes**	**1500 bytes**
Hosts Number	**Processed PPS**	**Processed PPS**	**Processed PPS**	**Processed PPS**
100	8.100.000	8.130.000	2.332.090	2.332.090
1.000	7.200.000	6.580.000	2.332.090	820.210
10.000	5.091.000	4.000.000	2.332.090	819.000
100.000	2.080.000	2.000.000	1.680.000	819.000
1.000.000	17.800	17.000	17.000	17.000

6. Synthetic Traffic: Packet Size/Hosts Number vs. Processed Packets (PPS)

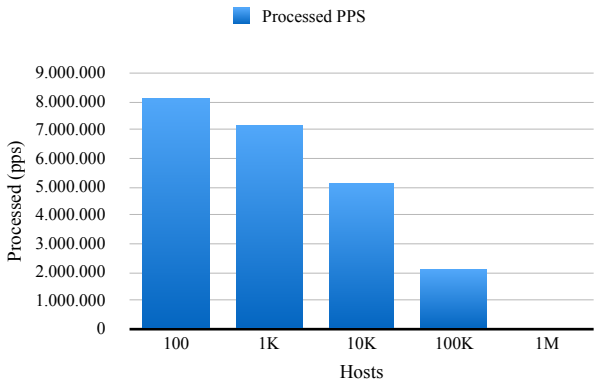
The above figure shows how the number of hosts and packet size influence the number of processes packets. Packet capture is not a bottleneck due to the use of PF_RING DNA. However, ntopng's processing engine performance is reduced in proportion with the number of active hosts and flows. Although networks usually have no more than a few thousand active hosts, we tested ntopng's performance across many conditions ranging from a small LAN (100 hosts), a medium ISP (10k hosts) and large ISP (100k hosts) to a backbone (1M hosts). The setup we used was worst case, because, in practice it is not a good choice to send traffic from a million hosts to the same ntopng monitoring interface.



7. pfdnacluster_master: PF_RING Zero Copy Traffic Balancing

The PF_RING library named *libzero* has the ability to dispatch packets in zero-copy to applications, threads and KVM-based VMs. The open-source application *pfdnacluster_master*[2] can read packets from multiple devices and implement zero-copy traffic fan-out (i.e. the same ingress packet is replicated to various packet consumers) and/or traffic balancing. Balancing respects the flow-tuple, meaning that all packets of flow X will always be sent to the egress virtual interface Y; this mechanisms works also with encapsulated traffic such as GTP traffic used to encapsulate user traffic in mobile networks [23]. This application can create many egress virtual interfaces not limited by the number and type of physical interfaces from which packets are received.

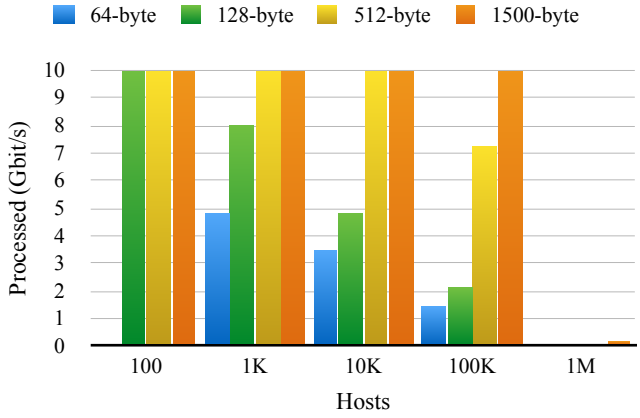
Thanks to PF_RING it is possible to balance ingress traffic to many virtual egress interfaces, all monitored by the same ntopng process that binds each packet processing thread to a different CPU core. This practice enables concurrent traffic processing, as it also reduces the number of hosts/flows handled per interface, thus increasing the overall performance. In our tests we have decided to measure the maximum processing capability per interface so that we can estimate the maximum ntopng processing capability according to the number of cores available on the system. Using the results reported in Fig. 5 and 6, using real traffic balanced across multiple virtual interfaces, ntopng could easily monitor multi-10 Gbit links, bringing real-time traffic monitoring to a new performance level.



8. Synthetic Traffic: Hosts Number vs. Processed Packets (PPS)

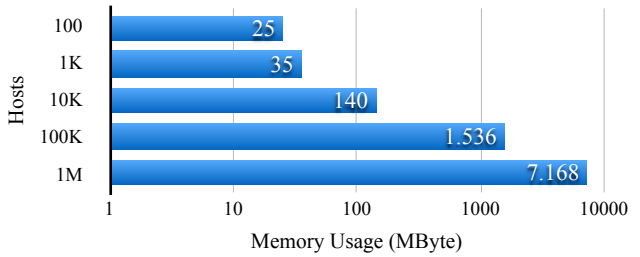
[2] Source code available at https://svn.ntop.org/svn/ntop/trunk/PF_RING/userland/examples/pfdnacluster_master.c

The previous chart above depicts the data in Fig. 6 by positioning the processing speed with respect to the number of hosts. As reported in Fig. 5 using real traffic on a full 10 Gbit link we have approximately 1.7 Mpps. At that ingress rate, ntopng can successfully handle more than 100K active hosts per interface, thus making it suitable for a large ISP. The figure below shows the same information as Fig. 8 in terms of Gbps instead of PPS.



9. Synthetic Traffic: Hosts Number vs. Processed Packets (Gbps)

Similar to processing performance, ntopng's memory usage is greatly affected by the number of active hosts and flows. As the traffic is reproduced in loop, hosts and flows are never purged from memory as they receive continuously fresh new data.



10. Hosts Number vs. Memory Usage

Memory usage ranges from 20 MB for 100 active hosts, to about 7 GB for 1 million hosts. Considered that low-end ARM-based systems [26] such as the RaspberryPI and the BeagleBoard feature 512 MB of memory, their use enables the monitoring of ~40k simultaneous hosts and flows. This is an effective price-performance ratio given the cost ($25) and processing speed of such devices. ntopng code compiles out of the box on these devices and also on the low-cost (99$) Ubiquity EdgeMax router where it is able to process 1 Mpps.

V. Open Issues and Future Work Items

While we have successfully run ntopng on systems with limited computation power, we are aware that in order to monitor a highly distributed network such as cloud system, it is necessary to consolidate all data in a central location. As both VMs and small PCs have limited storage resources, we are working on the implementation of a cloud-based storage system that allows distributed ntopng instances to consolidate monitoring data onto the same data repository.

Another future work item is the ability to further characterise network traffic by assigning it a security score. Various companies provide something called IP reputation [24] a number which the danger potential of a given IP. We are planning to integrate cloud-based reputation services into ntopng similarly to what we have done for domain categorisation. This would enable spot monitoring of hosts that generate potentially dangerous network traffic.

VI. Related Work

When the original ntop had been introduced in 1998 it was the first traffic open-source monitoring application embedding a web server for serving monitoring data. Several commercial applications that are similar to ntopng are available from companies such as Boundary [26], AppNeta FlowView [33], Lancope StealthWatch [31], and Riverbed Cascade [32]. However, these applications are proprietary, often available only as a SaaS (Software as a Service) and based on the flow-paradigm (thus not fully real-time nor highly accurate) These applications are difficult to integrate with other monitoring systems because they are self-contained. Many open source network-monitoring tools are also available : packet analysers such as Wireshark [30], flow-based tools such as Vermont (VERsatile MONitoring Toolkit) [27] or YAF (Yet Another Flowmeter) [29]. Yet, 15 years after its introduction, ntopng offers singular performance, openness and ease of integration.

VII. Final Remarks

This paper presented ntopng, an open-source, real-time traffic monitoring application. ntopng is fully scriptable by means of an embedded Lua JIT interpreter, guaranteeing both flexibility and performance. Monitoring data is represented using HTML 5 served by the embedded web server, and it can be exported to external monitoring applications by means of a REST API or through log files that can be processed by distributed log processing platforms. Validations tests have demonstrated that ntopng can effectively monitor 10 Gbit traffic on commodity hardware due to its efficient processing framework.

Code Availability

This work is distributed under the GNU GPLv3 license and is freely available in source format at the ntop home page https://svn.ntop.org/svn/ntop/trunk/ntopng/ for both Windows and Unix systems including Linux, MacOS X, and FreeBSD. The PF_RING framework used during the validation phase is available from https://svn.ntop.org/svn/ntop/trunk/PF_RING/.

Acknowledgment

Our thanks to Italian Internet domain Registry that has greatly supported the development of ntopng, Alexander Tudor <alex@ntop.org> and Filippo Fontanelli <fontanelli@ntop.org> for their help and suggestions.

References

1. P. Phaal, S. Panchen, and S. McKee, InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks, RFC 3176, September 2001.

2. B. Claise, Cisco Systems NetFlow Services Export Version 9, RFC 3954, October 2004.
3. S. Leinen, Evaluation of Candidate Protocols for IP Flow Information Export (IPFIX), RFC 3955, October 2004.
4. A. Caesar, Enabling NetFlow on a vSwitch, http://www.plixer.com/blog/network-monitoring/enabling-netflow-on-a-vswitch/, May 2013.
5. R. Lämmel, Google's MapReduce Programming Model — Revisited, Science of Computer Programming, 2008.
6. R. Bendrath, M. Müller, The end of the net as we know it? Deep packet inspection and internet governance, Journal of New Media & Society, November 2011.
7. T. Dierks, E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.1, RFC 4346, April 2006.
8. F. Fusco, M. Vlachos, X. Dimitropoulos, L. Deri, Indexing million of packets per second using GPUs, Proceedings of IMC 2013 Conference, October 2013.
9. N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, L. Jianying, NetFPGA - An Open Platform for Gigabit-Rate Network Switching and Routing, Proceeding of MSE '07 Conference, June 2007.
10. F. Fusco, L. Deri, High Speed Network Traffic Analysis with Commodity Multi-core System, Proceedings of IMC 2010 Conference, November 2010.
11. A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, kvm: The Linux Virtual Machine Monitor, Proceedings of the 2007 Ottawa Linux Symposium, July 2007.
12. P. Hintjens, ZeroMQ: Messaging for Many Applications, O'Reilly, 2013.
13. L. Deri, nProbe: an Open Source NetFlow Probe for Gigabit Networks Proceedings of Terena TNC 2003 Conference, 2003.
14. J. A. Kreibich, S. Sanfilippo, P. Noordhuis, Redis: the Definitive Guide: Data Modelling, Caching, and Messaging, O'Reilly, 2014.
15. L. Deri, F. Fusco, Realtime MicroCloud-based Flow Aggregation for Fixed and Mobile Networks, Proceedings of TRAC 2013 workshop, July 2013.
16. D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627, 2006.
17. B. Harzog, Real-Time Monitoring: Almost Always a Lie, http://performancecriticalapps.prelert.com/articles/share/281286/, December 2013.
18. ntop, nDPI, http://www.ntop.org/products/ndpi/, 2014.
19. T. Bujlow, V. Carela-Español, P. Barlet-Ros, Comparison of Deep Packet Inspection (DPI) Tools for Traffic Classification, Technical Report, Version 3, June 2013.
20. M. L. Reuven, At The Forge: Twitter Bootstrap, Linux Journal, June 2012.
21. L. Richardson, S. Ruby, RESTful Web Services, O'Reilly, 2007.
22. R. Gerhards, The Syslog Protocol, RFC 5424, March 2009.
23. 3GPP, General Packet Radio Service (GPRS); Service Description, Stage 2, Technical Specification 3GPP SP-56, V11.2.0, 2012.
24. M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, N. Feamster, Building a Dynamic Reputation System for DNS., Proceedings of USENIX Security Symposium, 2010.
25. M. Bostock, Data-Driven Documents (d3.js): a Visualization Framework for Internet Browsers Running JavaScript, http://d3js.org, .2012.
26. M. Joshi, G. Chirag, Agent Base Network Traffic Monitoring, International Journal of Innovative Research in Science, Engineering and Technology, Vol. 2, Issue 5, May 2013.
27. B. Cook, Boundary Meter 2.0 – Design, http://boundary.com/blog/2013/09/27/welcome-to-meter-2-design/, September 2013.
28. R. Lampert, et al., Vermont-A Versatile Monitoring Toolkit for IPFIX and PSAMP, Proceedings of MonAM 2006, 2006.
29. C. Inacio, B. Trammell, Yaf: yet another flowmeter, Proceedings of the 24th LISA Conference, 2010.
30. A. Orebaugh, G. Ramirez, J. Beale, Wireshark & Ethereal Network Protocol Analyzer Toolkit, Syngress, 2006.
31. Lancope Inc., StealthWatch Architecture, http://www.lancope.com/products/stealthwatch-system/architecture/, 2014.
32. Riverbed Inc., Riverbed Cascade, http://www.riverbed.com/cascade/products/riverbed-nba.php, 2014.
33. AppNeta Inc., Flow View, http://www.appneta.com/products/flowview/, 2014.