High-Speed Dynamic Packet Filtering

Luca Deri
deri@ntop.org
ntop.org

Abstract

One problem encountered while monitoring gigabit networks, is the need to filter only those packets that are interesting for a given task while ignoring the others. Popular packet filtering technologies enable users to specify complex filters but do not usually allow multiple filters to be specified.

This paper describes the design and implementation of a new dynamic packet filtering solution that allows users to specify several IP filters simultaneously with almost no packet loss even on high-loaded gigabit links. The advantage is that modern traffic monitoring applications such as P2P, IPTV, VoIP monitoring, and lawful interception can dynamically set packet filters to efficiently discard packets into the operating system kernel according to traffic, calls and users being monitored.

Keywords

Passive packet capture, packet filtering, traffic monitoring, Linux kernel.

1. INTRODUCTION

With the advent of gigabit networks, many existing applications such as IDS (Intrusion Detection Systems), traffic monitoring applications, and packet sniffers are faced with problems such as high packet-loss and CPU utilization due to the amount of traffic to be analyzed. The industry and academia have produced some solutions both in hardware [9] [18] [7] [27] [28] and software [1] [19] [10] [26] [29] able to accelerate packet capture in order to avoid packet loss due to high incoming packet rate. This is the theory as well as in practice that monitoring applications still need to process incoming packets at high-speed in order not to loose packets. The reason is that all the above solutions are suitable for accelerating packet capture at wire speed with minimum packet size, but offer very little in terms of packet filtering as usually the number of filters that can be specified is quite limited. In other application domains such as lawful interception [25] the problem is even worse as network operators usually provide a copy of packets flowing through a link where several hundred users are connected, while the law states that only the traffic of intercepted users can actually be captured and analyzed. This means that if an xDSL user is intercepted, only a few tenths pps (packets per second) need to be captured out of million pps flowing on the link. The analysis of VoIP (Voice over IP) traffic is also challenging as monitoring applications analyze signaling protocols such as SIP and H.323 in order to dynamically figure out the IP and port where the video/voice RTP stream for a given call will happen. In general P2P traffic monitoring [22] [23] [24] is the most difficult traffic to be tracked as it is by nature very dynamic, often encrypted/ scrambled, with continuously evolving protocols.

In a nutshell the above examples demonstrate that high-speed packet capture without advanced kernel filtering capabilities is useless in many scenarios; this is because the overall system performance can be improved only if both the kernel and applications do not waste several CPU cycles just for pushing unnecessary packets to user space that will be later discarded. Furthermore modern applications require dynamic packet filtering based on simple VLAN/IP address/port number criteria whereas popular packet filtering facilities such as BPF (Berkeley Packet Filter) or router-based ASIC filtering [16] allow one (or few) static filter whose reconfiguration has an impact

on active packet capture applications in terms of packet loss. The aim of this paper is to prove that it is possible to provide a positive answer to the above challenges, even using pure software-based approaches without the need to adopt costly hardware-based packet capture cards. Of course this work can be applied to cases where packet filtering needs to be performed, as it brings no advantage to those applications that need to analyze the whole input stream without any filtering at all.

2. MOTIVATION AND SCOPE OF WORK

The problem of packets filtering is well known [12] and it has been tackled very often in literature. Historically the BPF (Berkeley Packet Filter) [4] dated 1990, an evolution of early packet filtering efforts carried on at Carnegie-Mellon University, is still most the widely used solution to the problem. BPF includes a packet filtering machine able to execute programs. Each program is an array of instructions that sequentially execute some actions on a pseudo-machine state. The popular tcpdump tool allows the filtering program to be easily inspected (e.g. `tcpdump -d "tcp and port 80"`). In BPF checking whether a given packet is TCP takes 8 instructions; a slightly more complex check such as verifying if a packet is http, takes more than double the number of instructions. This example shows that even pretty simple filters can require a large number of instructions whose number increases significantly if boolean operators are used in the filter. The release of BPF greatly stimulated the research community as over the years several improvements to BPF have been produced [5] [11] [14]. Unfortunately all these efforts basically present the same characteristics:

- Only very few filters can be specified; in general only one filter can be specified although it can be divided into sub-filters linked with boolean operators that can be arranged on a filter graph.
- The filtering expression is implemented using pseudo instructions whose number is proportional to the number and complexity of the filters.
- Adding/removing filters require a general reconfiguration that can lead to packet loss as capturing activities might be temporarily interrupted.
- In some cases, packet filtering is accelerated using network processors that are no longer available on the market.

The conclusion is that the family of BPF-based approaches are suitable for some applications that require one arbitrarily complex filter, such as a packet sniffer like tcpdump or wireshark, but not for dynamic applications such as those previously listed.

Hardware-based packet filters such as those based on FPGA/NPU-accelerated [21] [15] cards, do not generally present limitations in terms of filtering speed as filters operate at wire rate with no packet loss. Instead the limitations are:

- Very few filters can be specified (usually in the 8-64 range), as they are limited by the space available on the silicon/RAM used for storing filters.
- The more complex the filter, the less filters can be specified. This is because filters are apparently implemented in a similar way to BPF, where the length of the filtering program depends on the filter complexity.
- Filters are usually very basic, compared to the richness and expressiveness of BPF, and sometimes they need to be specified in byte-code.
- Negative filtering (e.g. `not <expression>`, `not (tcp and port 80)`) is often not supported.
- Adding/removing filters can require a general reconfiguration of the hardware (e.g. when FPGAs are used), in some case this can take up to a minute, which is not compatible with most traffic monitoring applications that require to operate continuously.
- Filters are often not able to detect mixed encapsulation. For instance if MPLS-tagged packets are mixed with plain (ethernet+IP) packets and VLAN tagged packets, the filter is not

able to operate properly as they usually assume a unique type of encapsulation. This is because filters are defined with offsets that change according to the encapsulation, hence they fail when mixed encapsulations are used, which is very common on high-speed links where different kinds of traffic are transported using tagging.

ASIC-based filtering facilities present in some network routers such as Juniper JunOS or Cisco IOS, allow packets to be filtered efficiently using filtering expressions that are not as powerful as BPF but sufficient for most applications. The drawback of using routers as packet filters is the cost, complexity and feasibility of the whole solution, in addition to the fact that routers have not been designed to be used as packet filters with continuous/frequent (e.g. when VoIP traffic is analyzed, tracking RTP streams based on signaling) configuration changes.

The lack of inexpensive solutions for dynamic packet filtering in host-based systems has been the motivation for this research work. In fact both hardware and software-based solutions, with the exception of ASIC-based filtering that is too costly to be deployed in several scenarios, have interesting features but do not satisfy the requirements of modern dynamic network monitoring applications (e.g. VoIP monitoring, lawful interception, multimedia/IPTV, P2P traffic monitoring) and applications that need to handle hundreds of filtering rules such as host-based IDSs:

- Ability to specify a thousand different IP packet filters. A VoIP gateway can very well support thousand calls simultaneously, or an IDS can have several hundred active rules.
- Ability to dynamically add/remove filters without having to interrupt existing applications. In other words, filter reconfiguration should not require stopping the whole system even for a short period of time, instead the system must be able to operate while filters are manipulated.
- The filter processing speed and memory being used should not be proportional to the number of filters but independent from their number and complexity.
- Filters do not need to be as rich as BPF but header-based filtering 'VLAN/Protocol/IP address/port number' is enough for the targeted applications.
- Due to the nature of applications for which this filter has been designed, only "precise" filters (e.g. `host X and port Y`) are supported. Features like ranges (e.g. port > 1024) and sets (e.g. `host X or host Y or host Z`) are not supported as, if necessary, they can be expanded in several precise filters that are instead supported by the system. Note that with a little effort it is also possible to support subnetworks, that have not been taken into account as they are used very seldom to monitor the above targeted applications.
- In order to achieve a reasonable performance, the system can have a limited false-positive filtering rate (i.e. the filtering system can report that a given packet matches the filters even if it not so) but no false-negative rate (i.e. no packet matching one or more filter should be discarded by mistake by the system).

In a nutshell, the goal of this work is to create a low-cost system based on commodity hardware, able to filter packets at wire speed directly into the kernel so that dynamic monitoring applications receive only those packets they are interested in. This does not require that the filtering process needs to be fully accurate as long as dynamic in-kernel filtering dramatically reduces the amount of work that applications need to carry-on, also because applications might still need to do some extra filtering based on more complex criteria (e.g. TCP connection state, or last message exchanged on the connection). The goal will be achieved only if this system has a very high performance compared to the traditional approach "get every packet and filter it in user-space" applied by most applications or whenever (e.g. in FPGA/NPU filtering systems) it is not possible to define as many filters as necessary to the application due to hardware limitations.
This work will also be beneficial to applications such as IDSs that need to handle hundreds of rules such as `alert <source> -> <destination> (<extra criteria>)`. In fact for each of the above rules a filtering rule '`<source> -> <destination>`' can be set so that packets are early filtered into

the kernel and the IDS gets only those packets that are potentially interesting. Doing the same with BPF would result in a giant filter `((packet header filter for rule 1) || (packet header filter for rule 2)...)` that will be inefficient and handled by BPF implementations.

In general the scope of this work is not to replace BPF-like filters, useful for several applications, but to implement a pre-BPF filtering layer designed on the requirements of emerging traffic monitoring applications.

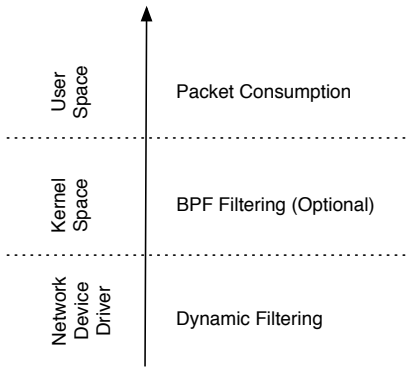

Figure 1. Network Packet Journey

For this reason, dynamic filtering is implemented directly into the network device driver because:

- This is the earliest place on the system where a packet pops up. This means that if packets are dropped on this component due to filtering, all components on top of the device driver (kernel and user space applications) will benefit from it. Instead, if dynamic filtering is implemented on top of the device driver or, even worse, in user-space, the amount of work needed to move a packet from the device to the filtering component will be wasted for those packets that do not match any filter.
- This is the layer under the kernel where the BPF (or other BPF-like filters) resides, hence dynamic filtering can be used by BPF as a pre-filtering stage in order to filter out all those packets that will definitively not match any BPF filter. From the user's point of view, this means that dynamic filtering is transparent to existing applications so no modification at all is necessary in order to take advantage of it.
- Dynamic filtering can take advantage of existing packet capture acceleration facilities that can further accelerate the journey of filtered packets through the kernel.

Reducing (by means of filtering) the amount of work that an application has to carry on in order to achieve a certain task, has a positive effect on the overall system performance. Later in this paper it will be shown that early (in kernel) and efficient packet filter leads to better application performance than when using a non-filtering accelerated network driver with filtering inside the monitoring application. On the other hand it is out of the scope of this paper to discuss how user space application performance can be improved in terms of packet capture and analysis performance.

The following chapter describes the design choices and implementation of the filtering. Furthermore it gives an evaluation in terms of performance of the solution.

3. THE DESIGN OF DYNAMIC FILTERS

In order to achieve the planned goal of having thousand of filters that can be dynamically added and removed, for the reasons explained before, it is obvious that it is not feasible to base it on a pseudo-machine as the one used by the BPF family. Instead it is necessary to use a different solution that guarantees constant memory consumption regardless of the number of filters with low/limited false positives and no false negatives. Bloom [2] [3] [6] [20] filters are a perfect solution to the above

problem as they are space-efficient, do not present false negatives, and allow elements to be dynamically added from the filter set. A bloom filter is an array of m bits all initially set to zero, and a set of k different hash functions each of which maps a key value in the 0...m-1 range. When a key element is added to the set, each hash is applied to the key, and all the bits that correspond to the results of the hash computation are set to one. Checking whether an element belongs to the set is pretty simple, as the k hashes are applied to the element, and if all the bits that correspond to the result of the hash functions are set to one, then the element belongs to the set. Bloom filters have several advantages with respect to other efficient data structures such as binary search trees and tries, as the time needed to add items or check whether an element belongs to the filtering set is constant, independently of the number of elements that belong to the set, similar to what happens with sparse hash tables [8]. The disadvantage is that elements can be added to blooms but they cannot be removed as each bit can have been set to one by several hashes hence if during removal it is set to zero this will break the logic. For this reason if removal is necessary a counting bloom [13] is used, where each bloom array element is not a bit but a counter that is incremented/decremented every time the bit is set/reset. Unlike filtering sets based on hash tables, adding an element never fails due to the overflow of the data structure, although the false positive rate increases as elements are added to the set.
Even if bloom filters seem to have several interesting features and few limitations, they are generally used only on hardware as their implementation in software can be rather costly because:

- Every time an element is checked if it belongs to the set, k hash functions need to be calculated. This is not a problem in hardware as hashes are computed in parallel, whereas in software they are computed sequentially with obvious limitations in terms of speed. It is worth to note that computing them sequentially and synchronizing results using semaphores does not bring any speed advantage.
- In order to both add and remove elements from the filter set, it is necessary to use counting blooms that unfortunately have the drawback of using too much memory, as they replace bits with counters, in particular if implemented inside the kernel where contiguous memory, necessary for allocating the bloom array, is always an issue.

Starting from the principle that a bloom-like approach is a good solution to the problem of keeping in memory a large amount of filters, the author tried to distillate a hybrid solution able to feature the advantages of blooms in a way that it could be efficiently implemented in software. It is clear that given a large number of filters, all the procedural algorithms (e.g. all those based on pseudo machines such as BPF) cannot be used, whereas a “compression” algorithm such as hashes or blooms is a good approach as it creates a small “fingerprint” easy to store and maintain. For this reason the author has decided to implement software bloom filters with the following differences with respect to the original recipe:

- The number of hash functions is limited to 2 (k=2). If the bloom size is large and the hash function is good, this solution does not lead to many false positives while providing a good performance as only two hashes need to be computed in the worst case whenever a packet is checked for inclusion in the filter set.

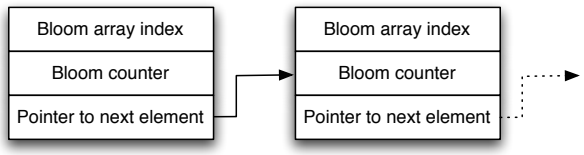

Figure 2. Counting Bloom Implementation

- As counting blooms are necessary but the author is not willing to pay for the drawback in terms of large memory usage, a novel implementation of counting bloom is presented, based on the fact that hash collisions (i.e. two different filters produce the same hash result), in particular with a large bloom array, are pretty rare. For this reason the bloom array is still im-

plemented with single bits, but whenever a new element is added to the filter set, in case of collision, a list of collisions is maintained. For instance if a new filter is added and the result of the hash function is x, if the bloom[x] is one, the collision list is searched for x. If the collision element does not exist, it is created and its counter is set to two, the original value of bloom[x] plus one (the collision), instead if it already exists it is incremented. If a filter with hash y is removed, before setting bloom[y] to zero the collision list is searched for y; if it exists the corresponding bloom counter is decremented by one, if it does not exists bloom[y] is set to zero. In case the bloom counter is decremented and the new value is one, the collision element for y is removed from the list as there are no more collisions for such element. With this solution it is possible to implement counting bloom at little cost as only the extra collision list needs to be maintained with respect to the original bloom.

Before discussing implementation details and performance, it is worth analyzing how the proposed solution behaves in terms of probability of false positives, i.e. packets that pass the filtering but that do not match any rule. The standard equation for calculating the probability of a false positive in a bloom filter is $q = 1 - (1 - (1/m))^{nk} \approx q = (1 - e^{-nk/m})^k$ where:

- q is the probability that a random bit of the bloom filter is 1.
- n be the number of elements that have been added to the bloom filter.
- m is the number of bits used to implement a Bloom filter.
- k is the number of hash functions that as stated before will be set to 2.

The increased probability of setting k to 2 (a relatively low value as in hardware usually 8 or more hashes are used) can be balanced using a large value for m, that it is generally not a problem in software as in hardware, even when coding inside the kernel operating system that has more constraints than user space. So doing some simple computations [17], supposing to set 1000 filters, the memory necessary for having a false negative probability of 10^{-6} is about 244 KB. This means that implementing blooms properly, it is possible to dramatically reduce the amount of packets to be analyzed by the monitoring application at the cost of few KB or memory, no false negatives, and with a constant filtering time as it is not proportional to the number of filters and bloom dictionary size.

Bloom filters are rather easy to implement as they are a contiguous array of bits that are set to 0/1 according to a hash function calculated on some input value, that is usually a concatenation of specific packet properties such as VLAN id, IP address, protocol and port. This concatenation is not unique as it is affected by the nature of the monitoring application how packets are filtered out.

Application Type	Packet Filtering Criteria
P2P	IP (P2P server). Port, used to track default P2P ports. P2P Session (e.g. TCP, IP address/port). Payload.
Lawful Interception	Intercepted IP address. Port (e.g. radius/1812) regardless of the user.
VoIP, Streaming IPTV	Signaling protocol (e.g. H.323, SIP: UDP and port 5060). RTP audio/video (UDP, IP address/port). RTCP for audio/video synchronization.
IDS	Home network (IP adddress/mask) to be analyzed. Port (e.g. http/80) where signatures are searched for match.

Table I. Comparison of Filtering Criteria based on Application Type

The previous table shows that there are no common criteria for filtering traffic among the listed applications, although all of them are pretty similar. In order to satisfy all of them, the author decided to compute the value of the bloom hash function on the concatenation of protocol, IP address and port.

Figure 3. Bloom Hash Calculation

This means that for each incoming packet, the hash function in the worst case is calculated twice on both the source and destination IP address/port using the same protocol value. Note that if the match using the first hash fails, the second hash is not computed at all. If some filtering rules require a wildcard, the value used for the corresponding hash element is set to zero. For instance in the rule "TCP/any-address/port=80" the value for the any-address is set to zero.
Wildcard implementation however is not cheap as:

- When the rule is added to the dictionary, its wildcard value is zero. Note that this algorithm does not lead to errors whenever the wildcard is used on a field where zero is a valid value (e.g. protocol id 0 corresponds to IP).
- When a packet is searched for match, for each wildcard value, both the real packet value and the wildcard need to be used in the hash. For instance if port can have a wildcard value, two hashes need to be calculated (protocol/ip/port and protocol/ip/0) hence the bloom dictionary is checked twice, one per hash value.

The consequence is that using wildcards with blooms is possible, if this practice is limited to one or two fields at most, because wildcard support significantly increases filtering time. Another practice to be avoided is the ability to handle large value ranges, such as IP addresses with a small mask (e.g. /16), because it will be necessary to explode all the addresses and compute a hash with each of them. If ranges support is mandatory, then it is recommended to implement several blooms one for each address space (e.g. bloom1 will handle proto/ip address with mask 32/port, bloom2 will handle proto/ip address with mask 24/port, and so on) than to have a single bloom and compute the hash with all the possible range values). Luckily all the monitoring applications taken into account into this work require precise matching with limited wildcard support, however it is worth to mention that if value range support and wildcard is strongly required, either some tricks are used or bloom filters need to be avoided as they are not adequate in terms of performance and false positive rate. In general, the work described into this paper is quite general and from case to case it can be implemented on a different way depending on the measurement requirements of the monitoring application.

4. IMPLEMENTING DYNAMIC FILTERING

While it is desirable to have a perfect filtering algorithm, monitoring applications can tolerate a relatively low false positive rate at the cost of dramatically reducing the amount of work that they instead would have to carry on without any kind of filtering. Therefore the idea is to implement a two stage packet filtering: in addition to the already implemented traffic filtering and selections facilities, a lower filtering layer based on bloom filters is used to reduce the number of packets that are forwarded to the application and that will be discarded later on.

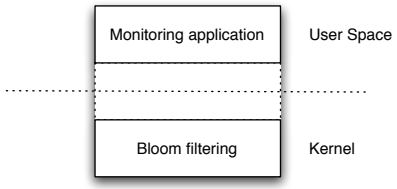

Figure 4. Two Stage Traffic Filtering

In order to efficiently implement bloom filtering and avoid modifying exiting applications, the best place to put them is inside the kernel as:

- Early packet filtering (as close as possible to the network adapter from which packets are received) reduces the amount of work necessary to drop filtered packets.
- Implementing it under the BPF layer facility used by most applications makes it transparent to the application while significantly improving the overall performance.
- Bloom complexity and memory requirements are rather limited, which makes them suitable to be implemented inside the kernel, where there are more limitations than in user space.

The linux 2.6 operating system has been selected as reference platform for the implementation. For performance reasons, blooms should be implemented as close as possible to the network adapter and the author has decided to implement them inside the network device driver rather than inside the kernel as this is the first place where incoming packets show up. The drawback of this design choice is that the implementation is NIC-dependent, even if as explained later in this paper, porting the code across different adapters is pretty simple as all the network device drivers share the same data structures hence most of the code can work with no change. The author selected the Intel GE adapter (e1000 driver) as reference card, and then ported the code to the Broadcom GE adapter (tg3 driver) in order to verify the code portability and as well demonstrate that the work was not tight to an adapter but that is pretty general.
On linux, packets are fetch from network adapters using NAPI, a network API that implements efficient packet polling. This means that whenever there is an incoming packet just received by the adapter and ready to be handled by the driver, NAPI does the job by calling a function (e.g. tg3_rx() on the Broadcom driver) that polls packets out of the adapter.

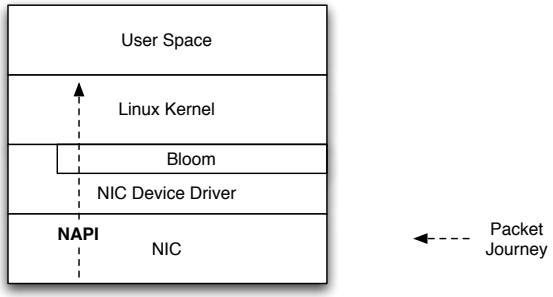

Figure 5. Packet Journey from NIC to User Space

Bloom filtering is implemented into this function, so that only those packets that satisfy the bloom filters are returned, whereas those having no corresponding filtering rule are dropped at this stage and never handled by NAPI. The advantage of this solution is that packets do not enter at all the protocol stack and hence there is no need to allocate memory (this is called skbuff or socket buffer on Linux) to store the packet that will then be deallocated later when the packet is dropped. Reducing the number of memory (de)allocations is already a significant performance improvement as this has to be done for each incoming packet.
NAPI is based on interrupts: an interrupt is generated as soon as an the adapter receives an incoming packet. At this point NAPI takes over the control, disables NIC interrupts and starts polling packets as long as there are packets to receive. When no more packets are available,

interrupts are restored. In the current Linux kernel implementation, packet polling is a single thread of execution, meaning that only one NIC at a time can be polled and that packet polling is a start-and-stop activity: at each NAPI polling session only some packets are fetched, then the kernel carries on other activities, then polling continues. Although this mechanism is rather efficient, it does not fully exploit multi-core/processor architectures where it would be possible to run several polling threads on different cores/CPUs. For this reason the author has implemented bloom filtering in two flavors:

- In the first implementation, the function called by NAPI for polling packets (e.g. tg3_rx() for Broadcom adapters) has been enhanced with bloom filtering, so as soon as NAPI fetches packets, before they are returned to NAPI they are filtered and then passed to NAPI. This approach requires very little code changes as it is simply an enhancement over the existing NIC driver.
- In the second implementation, as soon as the network card is initialized, a kernel thread is allocated for each active NIC. This thread, that can be spawn on a physical core/CPU (a.k.a. processor affinity) or be bounced across CPUs (this is the standard behavior for kernel threads under Linux). Each thread acts as a private NAPI poller for the NIC as it continuously polls packets as NAPI does (e.g. disabling/enabling interrupts as needed). In this solution, packets are polled as soon as they become available, then filtered with blooms and if they pass this stage they are pushed up the network stack by calling the netif_rx() function. The obvious advantage of this approach is that packet latency is decreased, but the greatest advantage is that the the NIC never interrupts the linux kernel as is the NIC that pushes packets when necessary. In particular with bloom filters when an application needs only a small subset of packets (i.e. most of the packets are discarded), the kernel will be interrupted by netif_rx() only for those packets that pass the filter. Instead of using the classical NAPI, the kernel would have to call NAPI (hence be interrupted during its activities) several times as there are many incoming packets, although most of the times NAPI will not return any packet, as most of them will not pass the bloom filters. The drawback of this approach is that the code is more complex with respect to the first implementation, as kernel threads need to be started/stopped according to the NIC state, problem that does not be to be addressed in the first implementation.

In the following section, the performance of both approaches will be analyzed and compared in order to quantify the speed advantage of the threaded-NAPI over the classical NAPI.

As blooms are implemented inside the kernel whereas applications run in user space, it has been necessary to implement a way for applications to pass commands (add/remove/reset blooms) to the kernel. In Unix usually this is done using the sysctl()/ioctl() system calls, but this would have required modifications to the socket layer and also to the libpcap. Therefore the author has decided to use a different solution that has the advantage of allowing existing applications to immediately take advantage of this work without any code change. Each bloomed-adapter registers a few entries into the `/proc/net/ethX` (where X is the index of the ethernet interface as listed by Linux) filesystem so that filters can be manipulated from user space or even from the command line:

- `/proc/net/ethX/enable`
- `/proc/net/ethX/reset`
- `/proc/net/ethX/rules`

Setting `enable` to 1 means that all incoming packets on the adapter ethX are first filtered with blooms and then passed to upper layers only if they pass the filters, whereas 0 disable blooms and everything works as without blooms. Setting `reset` to 1 causes all bloom filters to be reset and all existing filters to be removed. The rules entry is the most interesting as it allows filters to be added/removed as shown below:

- `echo "+ip=192.168.0.10,port=80" > /proc/net/eth1/rules` [add filter]
- `echo "-proto=tcp" > /proc/net/eth1/rules` [remove filter]

As you can read, filters can be manipulated in a user-friendly way from the command line or from applications in a simple way. However it is worth noting that filters are specified per network interface and that due the way blooms work, it is not possible to list stored filters being them stored as array of bits. For further implementation details, readers are advised to read the source code of the driver.

5. PERFORMANCE EVALUATION

The platform used to evaluate the implementation is a dual Xeon 3.2 GHz with HyperThreading (total 4 CPUs) equipped with a dual Intel Gbit card 64-bit/PCI-X and Linux 2.6.13. The traffic generator is an IXIA 400T equipped with 4 Gbit ethernet cards. The goal is to evaluate the performance on a real environment, hence the traffic has been injected on both adapters simultaneously so that we simulate a network tap that sends each traffic direction to each card, one for RX and one for TX. A simple packet capture application based on the standard libpcap has been developed. This application sets a few bloom filters and counts the number of full (i.e. no header-only or partial capture) packets received thorough libpcap on the specified adapters. A test session is successful if and only if there is no packet loss, i.e. all the packets sent by the traffic generator that pass the bloom filters are received by the application. In each test session, the IXIA sends a specified number of packets so that the pcap application can count the number of packets that have been received, and hence verify if there has been some packet loss. In all tests the IXIA injected a fixed number (300 million) of TCP packets with a fixed source (10.10.10.1) and an increasing destination (192.168.0.0-254). The following table shows the test outcome using the threaded version of blooms, and positions it with respect to the standard Intel driver and the same pcap application with user-space packet filtering. Note that:

- As there is only one IP match (i.e. match rate is 1:256) and the packet format is static, the filter does not require packet parsing as it simply checks a 32 bit integer (destination IP address).
- In case of vanilla kernel, the kernel/driver do not discard packets that are all received by the application.

ID	Intel Driver	Bloom Filters	Packet Size	Total Input Rate (both adapters)	System Load	Packet Loss
1	Threaded	No filters	900 bytes	270 Kpps	1.14	No
2	Vanilla	No filters	900 bytes	270 Kpps	3.12	Moderate (< 10%)
3	Threaded	One IP match	900 bytes	270 Kpps	1.66	No
4	Threaded	No filters	Random 64-1518	890 Kpps	1.34	No
5	Vanilla	No filters	Random 64-1518	890 Kpps	2.45	Moderate (< 10%)
6	Threaded	One IP match	Random 64-1518	890 Kpps	1.40	No
7	Threaded	No filters	64 bytes	2.89 Mpps	3.68	Moderate (< 20%; interface counters do not keep up with traffic)
8	Threaded	One IP match	64 bytes	2.89 Mpps	> 4	Strong (> 20%)

Table II. Bloom Performance Evaluation

Using the IXIA ability to precisely increase input traffic rate, it has been verified that the pcap application starts loosing packets when the input rate exceeds 1.8 Mpps. This is an excellent result as:

- It is a vast improvement over the vanilla Intel driver and it is more that enough for most monitoring applications as this is an extreme operating condition. Furthermore this technique shows that it's possible to handle continuos packet bursts with no loss and little system load (e.g. please refer to test 6).
- Even if blooms are not used, this technique features a high performance packet capture acceleration, due to the kernel threads that pump packets faster than the standard NAPI does.
- The above numbers do not change if a few hundred non-matching blooms are used in addition to the matching bloom. This demonstrates that filtering performance is not affected by the number of filters, but just false positive rate.
- This technique overcomes in terms of performance existing technologies. Just to give an idea, on the same system and in the same conditions, nCap [19] presents some packet loss over 560 Kpps whereas with this technique we need to go over 1.8 Mpps to see some packet loss.
- The non-threaded version of blooms has filtering performance between the vanilla linux and the threaded version, although it does not have any improvement in terms of accelerating packet capture as the threaded version does.

6. DYNAMIC FILTERING IN REAL LIFE: P2P TRAFFIC ANALYSIS

As explained before, dynamic filtering is very useful in every application and in particular to those situations where simple filters need to be added/removed very quickly. During the implementation phase, the author has developed a test application used to verifying the implementation. The application is a simple P2P accounting application that works as follows. Instead of accounting P2P traffic that it is quite hard using classic signature-based techniques, the application filters out non-P2P traffic using blooms, hence by difference it accounts P2P traffic. At startup the application adds some static filters for detecting well-known non-P2P traffic (e.g. DNS, SMTP, POP3) that is discarded by blooms used in a negative form, as all traffic that matches blooms is not forwarded but only the unmatched traffic. Note that usually there is no P2P traffic on those ports/protocols as firewalls usually check those ports for consistency (e.g. internet SMTP traffic is between two peers that both use port 25, so a communication between port 25 and port Y, where Y != 25, won't be allowed) so the application does not have false positives. The rest of the traffic is passed to the application that analyzes the packets payload and if known traffic on a non-standard port is detected (e.g. http on a port that's not 80), a new bloom filter is dynamically added in order to filter out this traffic. Although conceptually simple, the application demonstrated to account P2P traffic pretty well in addition to the advantage of configuring known traffic (a simple task for a network administrator) that has very well known patterns (e.g. they are often documented in RFCs) instead of pretending to detect P2P traffic that uses non-standard protocols changing over the time. Furthermore this application has been a good playground for testing both dynamic filtering and false positive rate, besides demonstrating that dynamic filtering is important when very dynamic traffic needs to be monitored effectively.

7. OPEN ISSUES AND FUTURE WORK

Bloom filters have been implemented per network interface hence all packets received from a bloom-enabled interface are filtered. Instead BPF filters are set per network socket (Linux) or BPF device (BSD). The main difference is that per-interface filtering is much more efficient as only those packets that pass filters are moved on upper layers, whereas per-socket filtering is more flexible as each socket can have a different filter. As the author wants to have a fast implementation able to sit under the BPF layer, blooms have been implemented per-interface (i.e. the bloom filtering takes place on all packets received by the interface) however it might be interesting to

explore how the overall performance changes when blooms are implemented per-socket (i.e. the bloom filtering happens only on packets received by the socket) similar to BPF. This is currently under implementation in PF_RING [30].

8. FINAL REMARKS

This paper described a novel approach to packet filtering that enables the creation of efficient network monitoring applications that need to track dynamic traffic; these applications include P2P, VoIP, IPTV traffic monitoring and lawful interception. This approach overcomes the limitations of BPF-like filters both in terms of number of simultaneous filters and ability to dynamically add or remove filters without any reconfiguration or downtime.

Furthermore the threaded bloom implementation shows that it is possible to capture and filter 1.8 Mpps without any packet loss on a commodity PC, which is a vast improvement with respect to the existing state of the packet capture techniques and more than enough for most monitoring applications.

9. AVAILABILITY

This work is distributed under the GPL2 license and is available at the ntop home page (http://www.ntop.org/) and other mirrors on the Internet.

10. ACKNOWLEDGMENT

The author would like to thank Alexander Tudor <alexander.tudor@agilent.com> for the several discussions about bloom filters, and RCS Lab for partially funding this research work.

11. REFERENCES

[1] A. Biswas, A High Performance Real-time Packet Capturing Architecture for Network Management Systems, Masters Thesis, Concordia University, 2005.

[2] B. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM, July 1970.

[3] F. Baboescu and G. Varghese, Scalable packet classification, ACM Sigcomm, 2001.

[4] S. McCanne and V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture, Proceedings of USENIX Conference, 1993.

[5] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture, proceedings of SIGCOMM 1999.

[6] A. Broder and M. Mitzenmacher, Network Applications of Bloom Filters: A Survey, proceedings of the 40th Annual Allerton Conference on Communication, Control, and Computing, 2002.

[7] T. Kratochvíla and others, Verification of COMBO6 VHDL Design, CESNET Technical Report 17/2003, 2003.

[8] T. Cormen and others, Introduction to Algorithms, Prentice Hall, 1990.

[9] The DAG Project, Univ. of Waikato, http://dag.cs.waikato.ac.nz/.

[10] L. Degioanni and G. Varenni, Introducing Scalability in Network Measurement: Toward 10 Gbps with Commodity Hardware, proceedings of IMC '04, 2004.

[11] D. Engler and M. Kaashoek, DPF: Fast, flexible message demultiplexing using dynamic code generation, SIGCOMM'96, 1996.

[12] M. Accetta and R. Rashid, The Enet packet filter, Carnegie-Mellon University, 1980.

[13] L. Fan and others, Summary cache: A scalable wide-area Web cache sharing protocol, proceedings of SIGCOMM '98, 1998.

[14] H. Bos and others, FFPF: Fairly Fast Packet Filters, proceedings of OSDI'04, 2004.

[15] Intel Corporation, Intel IXP2800 Network Processor Datasheet, 2002.

[16] Juniper Networks, Filter-based Forwarding - Technology Note, 2001.

[17] P. Manolios, Bloom Filter Calculator, http://www-static.cc.gatech.edu/~manolios/bloom-filters/calculator.html.

[18] Napatech A/S, The Napatech Traffic Analyzer Solution – White Paper, 2005.

[19] L. Deri, nCap: Wire-speed Packet Capture and Transmission, E2EMON, May 2005.

[20] S. Song and others, Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing, Washington University, 2005.

[21] Xilinx Inc., Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, November 2004.

[22] L. Zhou and others, P2P Traffic Identification by TCP Flow Analysis, Proceedings of IWNAS'06.

[23] Y. Gong, Identifying P2P users using traffic analysis, Security Focus, http://www.securityfocus.com/infocus/1843, 2005.

[24] P-Cube Inc., Approaches To Controlling Peer-to-Peer Traffic: A Technical Analysis, White Paper, 2003.

[25] C. Rogialli, Today's Challenges in Lawful Interception, RIPE 51, October 2005.

[26] D. Eppstain and S. Muthukrishnan, Internet packet filter management and rectangle geometry, Proceedings of the 12th annual ACM-SIAM symposium on Discrete algorithms, 2001.

[27] C. L. Schuba and others, Scaling Network Services Using Programmable Network Devices, Computer, v.38 n.4, p.52-60, April 2005

[28] D.E. Taylor, Survey and Taxonomy of Packet Classification Techniques, Tech. report WUCSE200424, Dept. Computer Science and Eng., Washington Univ., 2004.

[29] T. Y. C. Woo, A Modular Approach to Packet Classification: Algorithms and Results, Proceedings of IEEE Infocom, 2000.

[30] L. Deri, Improving Passive Packet Capture: Beyond Device Polling, Proceedings of SANE 2004, 2004.

12. BIOGRAPHY

Luca Deri is the leader of the ntop project (http://www.ntop.org/) aimed at developing an open source monitoring platform for high speed traffic analysis. He currently shares his time between NETikos S.p.A. and the University of Pisa where he has been appointed as lecturer at the CS department. His home page is http://luca.ntop.org/.