Enabling High-Speed and Extensible Real-Time Communications Monitoring

Francesco Fusco† fusco@ntop.org
Felipe Huici* felipe.huici@nw.neclab.eu
Luca Deri† deri@ntop.org
Saverio Niccolini* saverio.niccolini@nw.neclab.eu
Thilo Ewald* thilo.ewald@nw.neclab.eu
† University of Pisa
* NEC Europe Ltd.

Abstract—**The use of the Internet as a medium for real-time communications has grown significantly over the past few years. However, the best-effort model of this network is not particularly well-suited to the demands of users who are familiar with the reliability, quality and security of the Public Switched Telephone Network. If the growth is to continue, monitoring and real time analysis of communication data will be needed in order to ensure good call quality, and should degradation occur, to take corrective action. Writing this type of monitoring application is difficult and time consuming: VoIP traffic not only tends to use dynamic ports, but its real-time nature, along with the fact that its packets tend to be small, impose non-trivial performance requirements.**

In this paper we present RTC-Mon, the Real-Time Communications Monitoring framework, which provides an extensible platform for the quick development of high-speed, real-time monitoring applications. While the focus is on VoIP traffic, the framework is general and is capable of monitoring any type of real-time communications traffic. We present testbed performance results for the various components of RTC-Mon, showing that it can monitor a large number of concurrent flows without losing packets. In addition, we implemented a proof-of-concept application that can not only track statistics about a large number of calls and their users, but that consists of only 800 lines of code, showing that the framework is efficient and that it also significantly reduces development time.

I. Introduction

The last few years have seen a sharp increase in the use of the Internet for voice communications. While end-users are part of the trend, placing cheap Voice-over IP (VoIP) calls, service providers and enterprises are also benefiting from this change. Indeed, VoIP means that the same network can be used for both voice and data services, reducing equipment, operation and maintenance costs. Further, the use of IP enables the creation of converging voice and video services not available on traditional telephone networks.

Because of their experience with the PSTN (Public Switched Telephone Network), users expect a high quality service when it comes to voice communications. More specifically, the PSTN was designed, among other criteria, to achieve 99.999% availability, provide good sound quality and be resilient to attacks such as identify theft. However, the design goals of the Internet were quite different, and so VoIP presents a difficult set of challenges if it is to provide a service akin to that of the PSTN.

In order to cope with these challenges, it is paramount to monitor the network to ensure that the best-effort model of IP does not result in degradation of important VoIP quality of service criteria such as latency and packet loss, or that if degradation occurs, it can be detected in real-time and corrective action taken. However, monitoring VoIP is difficult for many reasons. First, the monitoring system must correlate separate signalling and media connections in order to generate a report for a single communication; worse, these connections tend to use dynamic ports, making the task even harder. In addition, the real-time nature of VoIP means that monitoring must also be done in real-time in order to quickly correct service degradation when it happens. Performance is also a concern, since VoIP communications are not only real-time but tend to contain small packets, further taxing the monitoring system.

These characteristics of VoIP mean that current solutions are not well-suited to monitor this sort of traffic. General-purpose monitoring tools do not have the ability to, among other things, correlate signalling and media connections, provide an analysis of VoIP quality of service metrics, nor track traffic efficiently in real-time. Even the monitoring services provided by VoIP devices such as SIP proxies are quite limited, often providing only billing and call log facilities. While these shortcomings are important enough, the bigger problem is that these tools cannot be easily extended to support the necessary functionality. In [5], the authors present a solution that has some common features with this work, but their focus is mostly on lower-layer protocols, and so it does not provide an ideal platform for quick development of VoIP monitoring applications. Hardware solutions exist but they are also hard to extend, are usually difficult to program or customize and are of course expensive compared to software-based platforms running on commodity hardware.

It is clear that there is a need for a framework that will allow for the development of VoIP monitoring applications while meeting certain crucial goals. First, it should provide basic VoIP monitoring mechanisms in order to minimize the development time of applications. Second, the framework should be easily extensible so that if this functionality proves insufficient, developers could add to it without lots of effort. Finally, the framework should be efficient in order to meet the demands of real-time VoIP traffic monitoring.

In order to meet these goals we implemented RTC-Mon, the Real-Time Communications Monitoring framework. We designed RTC-Mon with the aim of helping software architects develop VoIP monitoring applications scalable to ISP volume.

In addition, the framework grants high performance and usability, hiding the complexity of traffic capture and protocol analysis and thus reducing the time to market of complex applications.

RTC-Mon consists of an extensible, kernel-level plugin architecture, a SIP [20] plugin that performs high rate parsing and filtering, an RTP [21] plugin that supports in-kernel media analysis and a user-level VoIP library. In addition, we used the framework to implement VoIP Console, a proof-of-concept application that, despite being small (800 lines of code), can track a large number of per-user and per-call statistics such as the number of successful/attempted/rejected calls, call setup time, the peers involved, and RTP streaming information. Finally, we provide a testbed evaluation of RTC-Mon to show that it is able to support all of this functionality while yielding high performance.

II. RTC-Mon Framework Overview

As mentioned in the previous section, the characteristics of real-time communications monitoring dictate several requirements when designing a framework for application development, including flexibility, high performance, extensibility and short development time. Flexibility and high performance are necessary if we are to provide a framework that will allow for a wide range of high performance applications. We implemented RTC-Mon on Linux for a variety of reasons, including its open source paradigm, the availability of kernel packages capable of processing packets at high rates, and the fact that Linux is used in a large range of platforms, from powerful servers to less capable embedded devices.

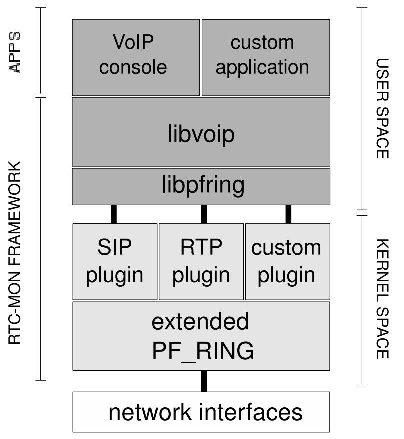

Fig. 1. RTC-Mon framework overview.

To achieve high performance, we relied on PF_RING [10], a Linux kernel-level network socket that dramatically improves packet capture speed. We expanded PF_RING by adding a plugin architecture to it, thus allowing developers to extend the framework by creating high performance, custom plugins (see Figure 1).

In order to allow for short application development time, we implemented plugins for SIP and RTP, two of the most commonly used protocols for VoIP communications. In addition, we implemented a user-level VoIP library called *libvoip* built on top of an extended *libpfring*, PF_RING's library. The *libvoip* library uses these two plugins and provides higher-layer functionality to applications. Further, libvoip can be easily extended to make use of custom plugins, as shown in the figure. It is important to note that while we designed the framework with VoIP monitoring in mind, it is flexible enough to accommodate any other real-time monitoring functions.

One final requirement was that the framework should allow for the quick development applications. To show that RTC-Mon meets this requirement, we implemented a proof-of-concept monitoring application called VoIP Console consisting of only 800 lines of code. Despite its small size, VoIP Console is quite capable of monitoring a large array of important VoIP statistics.

The rest of the paper is organized as follows. Section III discusses the extensions to PF_RING in detail, including the plugin architecture. Section IV describes the user-level libvoip library and Section V the proof-of-concept VoIP Console application. Finally, Section VI provides a performance evaluation of the various components of the framework and Section VIII concludes.

III. Extended PF_RING

PF_RING is a Linux kernel module providing a new type of socket that, coupled with device polling, allows packets to be captured at high rates. One of the advantages of the module is that it is independent of device drivers, and so can be used with any network card. In addition, PF_RING comes with *libpfring*, a user-level library that allows applications transparent access to the kernel-level sockets. Finally, we also chose it because it is a mature project that has a wide and active community of users.

While PF_RING has the basic mechanisms needed for packet capture, we had to extend so that it would meet the requirements of the RTC-Mon framework. PF_RING focuses on IP-layer packets, and so we added IP defragmentation to it in order to process higher layers in the kernel, thus achieving higher performance. To make use of this functionality while providing extensibility, we expanded PF_RING with a plugin architecture that enables easy implementation of higher-layer functions in the kernel. The rest of this section gives a detailed discussion of this architecture as well as the SIP and RTP plugins we implemented on it.

A. Plugin Architecture

We developed the plugin architecture so that developers would be able to perform a variety of crucial monitoring functions in the kernel, including packet payload parsing, packet content filtering and traffic statistics computation. Plugins are essentially kernel modules, providing a simple way for developers to add support for functions and protocols that the framework might not already come with. Further, the architecture allows for packets to be handled by one or many plugins before being discarded, thus enabling the development of applications that rely on several protocols or functions.

The process begins by creating a PF_RING socket and assigning it to an interface[1]. The socket has a set of rules associated with it that decide which plugins to send packets to. Each of these rules has three components: a filter, an ID identifying the plugin to send the packet to in case the filter matches, and an action ID that decides what happens to the packet in case of a match once the plugin has processed it.

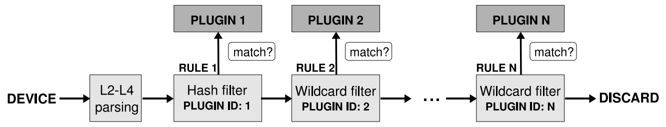

Fig. 2. Extended PF_RING overview with plugin architecture.

Figure 2 illustrates the basic architecture. First, PF_RING receives a packet on a device and parses its headers up to the transport layer, performing any IP defragmentation where needed. It then goes through the socket's set of rules one by one, applying a rule's associated plugin to a packet only if the rule's filter matches (for instance, a rule for an HTTP plugin could have a filter for TCP packets with port 80).

The filtering mechanism requires a closer look. As shown in the figure, filters can be of two types: hash or wildcard. Hash filters are used when it is necessary to track a six-tuple connection with the fields ⟨*vlan id, protocol, source IP, source port, destination IP, destination port*⟩ without incurring the linear evaluation costs of a rule list. The hash is managed by the plugin, thus giving it the power to decide what connections to track and what state to keep; as we will show later, this is used by the RTP plugin to track different calls.

Wildcard filters, on the other hand, are more flexible, allowing to match, for example, all UDP packets going to a specific port. These filters can also specify a higher-layer, plugin-specific filter. In this way, a user could instrument the system to process only INVITE messages (one of the types of messages that SIP has).

We mentioned earlier that rules also have an action associated with them. If a packet matches a rule's filter, the action determines what happens to a packet *after* it has gone through the rule and its plugin (if the packet does not match the rule it is evaluated against the next rule). In our architecture, there are three options for the action:

1) Continue rule evaluation.
2) Stop rule evaluation, send packet to user space.
3) Stop rule evaluation, do *not* send packet to user space.

The first option is straight-forward, allowing subsequent rules and plugins in a socket's set to also process packets (see Figure 3). The other two stop the rule evaluation: if a packet has already been handled by the appropriate plugin, a developer can use one of these two options to prevent any further and perhaps wasteful processing. Finally, a developer might need to pass some of the information gathered up to user space using option 2. Copying data to user space can be costly, however, and so option 3 is there to allow a developer to accumulate data in the plugin that an application can poll from time to time; this is the mechanism used by the RTP plugin described later in this section.

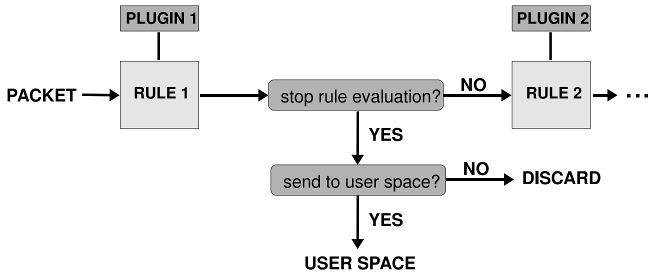

Fig. 3. Packet paths for all possible rule action types.

Creating a plugin for the architecture is simple and consists of essentially implementing functions for parsing and filtering traffic as well as for polling packet statistics from user space. We now turn our attention to the implementation of two such plugins for the SIP and RTP protocols.

B. Implemented Plugins

One of the most obvious features of VoIP monitoring applications is calls monitoring, and so we needed to make sure that the RTC-Mon framework supported this functionality. Calls monitoring consists of measuring both the signaling performance and analyzing the media traffic. For the former, the framework should be able to provide metrics such as the call setup time and the invite time (the time between the sending of an invitation and the receipt of a "ringing" response). In terms of media analysis, RTC-Mon should provide statistics on packet loss, jitter and the number of out-of-order packets for every active media stream.

In order to provide the basic kernel-level mechanisms needed to meet these demands we implemented SIP and RTP plugins. We chose these two since they are two of the most popular signaling and media transport protocols currently in use for VoIP communications. It is also important to note that these are essentially kernel-level modules: providing SIP filtering and RTP analysis at this level reduces the overall system load due to memory copies and system calls.

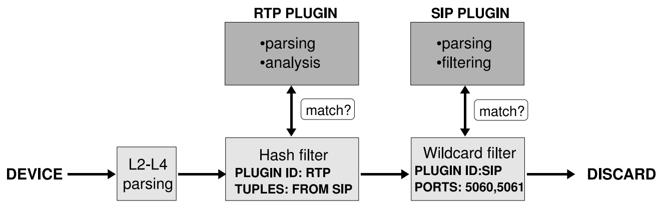

Fig. 4. Instance of PF_RING with SIP and RTP plugins.

[1]PF_RING supports the creation of several sockets per interface, thus allowing several independent applications to run on the same interface; throughout our discussion we use only one socket for simplicity's sake.

The SIP plugin takes care of parsing the most important parts of the message, including the start line and the To, From, Call-ID and Cseq header fields; this provides the basic information needed for applications to monitor signaling performance (we decided not to perform more in-depth SIP parsing in the kernel since not all applications may use the extra information). The plugin also gives offsets to some other information that might be useful in terms of monitoring, such as the message type, the Contact and User-Agent header fields, and the SIP message payload. Finally, the SIP plugin also provides wildcard filtering that can be applied to any of the parsed fields mentioned above.

Regarding RTP, we began with the observation that most of the bandwidth consumed by VoIP communications is the result of RTP streams. Consequently, a monitoring system will likely receive a large amount of media traffic, causing a significant number of context switches and system calls as it is passed to user level. To improve performance, we implemented an RTP plugin that performs the analysis of media traffic in the kernel and provides this information to applications. For each RTP stream the plugin computes the number of packets, the total bytes, the number of out-of-order packets and the medium, average and maximum jitter. In addition, the plugin stores the last sequence number and the synchronization source field (SSRC).

Figure 4 shows how the plugin architecture would look for an application using the SIP and RTP plugins together. The rule for RTP comes first, using a hash-based filter in order to track connections. No information is copied to user space at this stage, since the RTP plugin is designed to keep statistical information and pass it up to user space only upon request. The rule for RTP thus uses the "continue evaluation" action so that packets will reach the SIP rule. This rule matches all UDP traffic to ports 5060 and 5061 (the SIP well-known ports) and uses the "stop rule evaluation and send to user space" option to copy the parsed data and the packet up to user space. The setup shown in the figure is precisely the one we use for the VoIP console application we describe in Section V.

IV. User-Level VoIP Library

While PF_RING and the plugins give the high performance and basic analysis needed for monitoring, RTC-Mon provides higher level functionality in user space. To do so, we implemented *libvoip*, a C++ VoIP analysis library that exploits the features supplied by RTC-Mon's kernel infrastructure, thus allowing fast development of complex VoIP monitoring applications.

While the library is quite capable of performing a number of tasks including active call and user monitoring, our aim when designing it was to provide an extensible framework rather than a static library, since the functionality given will never be enough to satisfy all applications. As a result, the library takes care of carrying out standard tasks such as packet dissection, leaving developers free to concentrate on writing more complex library functions. With this in place, RTC-Mon should be able to accommodate both basic monitoring applications as well as those seeking to perform more involved tasks.

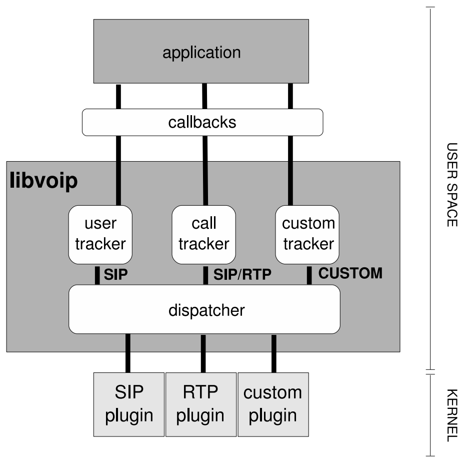

Fig. 5. Libvoip library overview.

Libvoip is an event-based library consisting mainly of a dispatcher and trackers (see figure 5). The dispatcher takes care of receiving packets and deciding which trackers to send them to. Trackers are in charge of combining information from different plugins and protocols in order to monitor a specific characteristic of the traffic, such as the number of calls a user has made. In addition, trackers are the mechanism by which developers extend libvoip: monitoring a new metric is as easy as implementing a new tracker and having the application make use of it.

In more detail, the dispatcher periodically checks a ring provided by PF_RING to see if a new packet (and its parsed information from layers 2 through 4) is available. To figure out which tracker to send a packet to, it first determines the packet type (e.g., SIP) by checking the ID of the plugin that processed it. With this information, the dispatcher then sends the packet to all trackers that at start-up had registered interest in this type of packet.

In order for the actual application to receive information, it registers a callback function with a tracker; in this way, whenever the tracker generates an event the function is called. It is entirely up to the implementer of a tracker to decide what types of events are generated, when they are generated, and what type of information is passed up to the application. This approach is flexible and simple: generating an event is as easy as implementing a single function and setting a variable to a value greater than 0.

In addition, trackers can perform further processing, such as more in-depth parsing, analysis or keeping state. The application could certainly take care of implementing these functions, but providing them in a tracker means that not only can other applications take advantage of them, but the developer profits from the event-based mechanisms provided by the library.

While the library is extensible, we have provided two trackers, *UserTracker* and *CallTracker* in order to speed up development time for applications that need basic VoIP communications monitoring. *UserTracker* uses SIP information to keep track of VoIP users. *CallTracker*, on the other hand, combines SIP and RTP data to monitor call information and can even handle monitoring behind NATs; it is this tracker that takes care of polling the RTP plugin for analysis data. In the next section we describe an application that makes use of trackers as well as the rest of the RTC-Mon framework to monitor VoIP communications.

V. VoIP Console Application

We designed the framework to, among other things, enable fast development of monitoring applications. To demonstrate this, we have built a proof-of-concept application on top of RTC-Mon called *VoIP Console*. Since the framework already provides most of the basic mechanisms needed to monitor traffic, the application consists of only 800 lines of code yet it is quite capable of tracking a large set of important statistics.

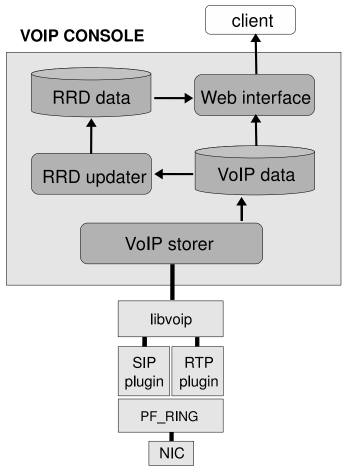

Fig. 6. VoIP Console application overview.

As shown in figure 6, VoIP Console is made up of several components. The VoIP storer component uses the framework to analyze the VoIP traffic. Since much of the complexity is encapsulated in *libvoip* and the rest of RTC-Mon, the storer is quite simple yet capable of providing a large set of information arising from the analysis of VoIP traffic. To accomplish this, we used the *UserTracker* and *CallTracker* described in the previous section.

The information generated by the VoIP storer is saved in a MySQL database and serves as the basis for the RRD Updater component. This component handles time-varying data such as the maximum number of concurrent calls. In order to achieve this, the RRD Updater retrieves relevant information from the VoIP database and stores it in a Round-Robin Database (RRD) using RDDtool [18]. We decided to use a RRD since it can store metrics in a constant and space-efficient manner.

The final component is the web interface, which compiles data from the RRD and VoIP databases and publishes information about users and calls to web clients. For each user, the web interface shows the display name, the SIP name, the number of successful/attempted/rejected calls, the number of SIP methods, the last successful registration time and the user agent. For each call, it gives the status (updated in real-time), the peers involved, the call setup time, the invite time and RTP streaming information (if the call has been successfully established). As can be seen from the long list above, VoIP Console provides, thanks to the RTC-Mon framework, quite a lot of functionality for only a few hundred lines of code.

VI. Evaluation

While the RTC-Mon framework provides flexibility and reduces development time, we still need to show that it is able to cope with high data rates when using a general purpose computer to monitor traffic. To do so, we built a small testbed and implemented all of the components of the framework, testing them to see how well they perform.

Before discussing the evaluation results, it is worth mentioning two relevant parameters: packet size and the maximum number of concurrent flows that a link can accommodate. Packet size is important because smaller packets put higher strain on the monitoring system. The number of flows, on the other hand, gives a good idea of the maximum amount of state that the system might need to keep in order to monitor all calls currently active.

Both of these factors depend on the codec used. Different phones (be them hardware or software-based) support different codecs, and so there is a variety of them used in VoIP communications. Figure 7 lists relevant information for some of the most common codecs. To calculate these numbers we assumed Ethernet Gigabit links and IP/UDP/RTP packets, since this is the most common scenario. As can be seen, packet sizes range from 78 to 218 bytes: it is important that our experiments cover this entire range, since it is bounded by the two most supported codecs, G.729 and G.711 (we arrived at this conclusion by tallying up the supported codecs of 43 hardware and software phones from companies like Cisco, Grandstream, Linksys, Siemens and Snom listed in [19]).

The figure further shows that the maximum number of concurrent RTP flows for any of the codes is at most 48,000 or so. This latter is a theoretical number, since it assumes perfect conditions and no other traffic on the link, but it gives a worst-case figure. As a result, in the rest of the section we will focus on number of flows from thousands up to 50,000.

One final factor worth keeping in mind is the maximum theoretical rate for Gigabit Ethernet. Depending on packet size, the actual rate on such a link is less than 1Gb, as a result of header overheads; figure 8 shows the maximum theoretical rates for the small packet sizes we are interested in. Please note that the rates presented in the results are loss free (no packets dropped).

A. Testbed

The testbed used for the experiments consists of an IXIA 400 traffic generator [15], two computers and an HP Procurve 1800 switch connecting them all (see figure 9). We used the

Codec	Sample Size (bytes)	Sample Rate (ms)	Bit Rate (Kbps)	Packet Size (bytes)	Eth. Bandwidth (kbps)	Max Num. Flows (Gb link)
G.711	80	10	64	218	87.2	11,468
G.726	20	5	32	138	55.2	18,116
G.726	15	5	24	118	47.2	21,186
G.728	10	5	16	118	31.5	31,780
G.729	10	10	8	78	31.2	32,051
iLBC	38	20	15.2	96	27.7	36,101
G.723.1	24	30	6.4	82	21.9	45,732
G.723.1	20	30	5.3	78	20.8	48,077

Fig. 7. Rate information for various common VoIP codecs. The figures assume Ethernet/IP/UDP/RTP headers.

Packet size (bytes)	Size on wire (bytes)	Theo. Max (in Kpkts/s)	Theo. Max (in Mb/s)
64	84	1488	762
100	120	1042	833
150	170	735	882
200	220	568	909
250	270	463	926

Fig. 8. Maximum theoretical rates for 1Gb Ethernet for small packet sizes.

IXIA 400 to generate trash UDP traffic, a computer to generate VoIP traffic and another one as the monitoring system.

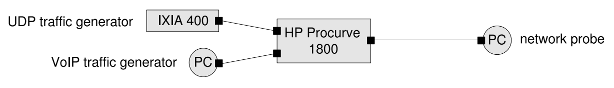

Fig. 9. Experiment network topology showing the UDP traffic generator, the VoIP generator and the network probe.

The VoIP traffic generator consists of an Intel Centrino CPU at 1.86Ghz with 512MB of memory running a Linux 2.6.24 kernel. The computer injects VoIP traffic by replaying a packet trace with *tcpreplay*. The trace contained about 1,000 calls each lasting 30 seconds: from [4] we know that calls typically last about 100 seconds, and so we picked 30 seconds as a worst-case scenario, since we could produce a higher call rate with short calls, thus putting more load on the system. In addition, the maximum number of concurrent calls in the trace was about 200; while this may seem small, the tests were run with a mixture of calls and other non-VoIP traffic, adding up to as many as 50,000 concurrent flows.

The monitoring system has a Celeron processor running at 3.2Ghz, 4GB of memory, an Intel 1000 NIC and also runs a Linux 2.6.24 kernel. It receives the combined traffic from the UDP and the VoIP generators.

B. RTC-Mon Performance

In order to test the performance of the RTC-Mon framework we decided to focus on RTP traffic. The reason for this is that control traffic (for example SIP) represents only a small fraction of all traffic of a VoIP call, and so it does not tax the system nearly as much as the RTP traffic does. In more detail, we mentioned earlier that calls typically last about 100 seconds. We ran a quick test capturing 100-second calls using different codecs and found that SIP traffic was only 1% of the total traffic. In addition, the RTP plugin puts further strain on the system since it keeps state for each ongoing call. While we also processed and analyzed SIP traffic, the tests are designed to stress the RTP analysis, since we feel this dominates the overall system performance.

As a first test, we wanted to see the system's performance when dealing with a mix of VoIP traffic and other UDP traffic. More specifically, we were interested in the improvement arising from analyzing traffic in the kernel plugins rather than in user space. To do so, we first measured the load that the RTC-Mon framework put on the CPU of the monitoring computer. We then implemented a special version of *libvoip* that performs the exact same analysis but does so in user level rather than rely on kernel-level plugins. The monitoring was driven by *voipcapture*, a minimal RTC-Mon application that forces analysis of both signalling and media traffic, but does no further processing, ignoring any events it receives. The results in Figure 10 show that performing the analysis in the kernel yields clear improvement regardless of the incoming packet rate. Further, the figure demonstrates that the framework can cope with large packet rates while keeping the CPU relatively idle (between 80% and 20% for the whole Gigabit range). It is also worth noting that the user-level analysis begins to drop packets when the CPU idle percentage reaches 0.

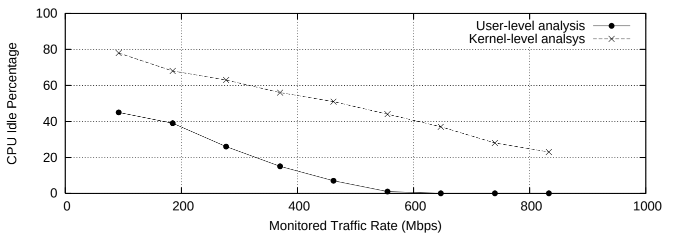

Fig. 10. Performance when filtering trash UDP traffic from VoIP traffic (250-byte packets).

So far we have shown that RTC-Mon is quite capable of picking out VoIP traffic from a loaded link and analyzing it; we now turn our attention to how well the framework performs when it has large amounts of traffic to analyze. In order to help with this we wrote a small program, *rtpcontrol*, that provides command-line control of the capture and analysis of RTP traffic by allowing insertion of a configurable number of RTP rules, each representing a monitored stream; packets belonging to a stream are used to update the stream's statistics.

Since we did not have a powerful VoIP traffic generator handy, we used the IXIA 400 to generate UDP traffic and configured the RTP plugin (using the *rtpcontrol* program) so that it would consider these packets as malformed RTP packets, thus forcing them to be analyzed. As mentioned at the beginning of the section, a Gigabit Ethernet link can carry at most about 50,000 RTP flows. To test this limit, we configured the IXIA to generate up to this many flows, while setting the VoIP traffic generator (a computer) to replay the VoIP packet trace; the monitoring system tracked every single one of these flows and kept statistics for them.

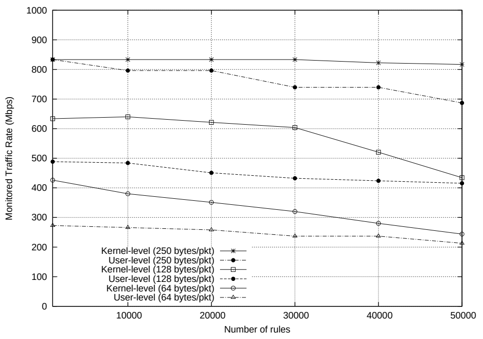

Fig. 11. RTC-Mon performance when tracking large numbers of RTP flows. User-level means that packets are copied to user space for payload analysis.

Figure 11 shows the results of these tests. It contains two graphs per packet size, one representing the performance when all of the analysis is done in the kernel, and another one when the basic RTP analysis is done in the kernel but then the packet is copied to user space (perhaps to analyze its payload). As can be seen, RTC-Mon yields high rates when monitoring even small packet sizes. For 128-byte packets and 30,000 RTP rules, for instance, it can process traffic at about 600Mbps, 70% of the theoretical maximum; for 250-byte packets the rate jumps to about 830Mbps, 90% of the maximum. As expected, copying packets to user space results in a performance hit, but even in this scenario RTC-Mon is able to process packets at a very respectable 500Mbps for 128-byte packets and 50,000 RTP rules. It is worth noting that the kernel and user-level curves tend to merge near the 50,000 limit because filter processing begins to dominate the performance and in both curves this is done in the kernel.

Another important factor concerning a monitoring system is how quickly it can reconfigure the rules that determine what traffic to track. To give a baseline number to compare to, we decided to test the Berkeley Packet Filter [6] (BPF), since it is the de-facto standard filtering mechanism for Unix-like systems. We began by writing a simple C program that measures the time needed to compile a complex filter containing many expressions (monitored RTP streams). For a filter with 200 expressions, the compile time was 800 milliseconds (we tried filters with more expressions but the kernel refused them, returning an error). This means that if we were monitoring 199 streams and wanted to monitor an extra one, it would take at least this time before the system could track the new stream. To put this into perspective, G.711 and G.729 generate a voice packet every 20ms, so as many as 40 packets could go untracked before the change takes place.

To test the time it takes to change filters in RTC-Mon, we inserted (and removed) a single rule 500 times for each run, and we repeated the experiment with a varying number of rules installed in the monitoring system (see Figure 12). The results clearly show that it is possible to insert rules much faster than with BPF filters, and that removing them is even quicker. In the worst case (inserting a rule with 50,000 rules loaded), the total time is about 2 milliseconds, meaning that at most a single G.711 or G.729 packet would go untracked.

# Rules	**Avg. Ins. (usec)**	**Max. Ins. (usec)**	**Avg. Del. (usec)**	**Max. Del. (usec)**
10,000	20.9	98	4.7	81
20,000	22.3	130	5.3	95
30,000	25.2	210	7.9	150
40,000	27.9	379	12.7	225
50,000	47.1	2037	19.1	52 7

Fig. 12. Time needed to change a rule (a monitored stream) in RTC-Mon. Ins stands for insertion, del for deletion and usec for microseconds.

VII. Related Work

Much research has been done analyzing the QoS network parameters in order to test the feasibility of VoIP services over current generation networks [23], [22], [7]. The passive analysis approach has been used to perform speech quality [8], [16] and signalling performance measurements [1]. While these projects highlight the importance of VoIP monitoring, they do not cover the need of software frameworks allowing fast development of complex monitoring applications. Further, they support a limited number of performance metrics and they do not explicitly analyze the performance requirements that a monitoring application may have, but limit themselves to mentioning hardware cards as a solution.

Hardware cards [12] improve the performance when capturing packets, leaving more spare CPU cycles to perform monitoring tasks. However, they are expensive and offer limited flexibility, since each card can usually only serve a single monitoring application. Programmable network cards such as TILExpress-64 [9] are capable of accelerating not only packet capture but also their analysis, since they allow the execution on the card of monitoring programs written in C. They are easier to program than a network processor, but porting C programs to them is not always an easy task.

The Click modular router [17] allows users to build efficient network devices out of general-purpose computers by providing a set of processing elements that are then connected in various ways to create the desired device. While Click yields very good performance, it is mostly aimed at lower-layer functionality, and so it comes with very little application layer elements. In addition, it does not provide a user-level library, and so developing full applications with it cannot be done as quickly as with RTC-Mon.

Mmdump[25] is a real-time version of the popular tcpdump tool and it is implemented on top of *libpcap*. It parses session control protocols like RTSP in order to discover the dynamically negotiated ports and change the underlying BPF filter. The dynamic filter reconfiguration needs to be quick in order to not lose packets during the update. Unfortunately, mmdump is slow due to its use of BPF. As a result, the aim of BPF successors such as *BPF+* [3], DPF [13] and PathFinder [2], has been to optimize the time to evaluate complex filtering expressions.

The work in [11] suggests the adoption of Bloom filters to overcome some of the limitations of BPF and its successors; this technique gives support for a large number of different filters. While it is not able to perform content-based filtering, it can still be used to filter RTP streams.

Content-based filtering, and in particular using information from application layer protocols to filter at the kernel level is one of the aims of Fairly Fast Packet Filter (FFPF) [5]. FFPF is a high-performance packet capture and filtering architecture. Like *PF_RING*, it employs shared memory buffers, reducing system load due to packet copying and context switching. Moreover, like *xPF* [14], it allows the execution of monitoring programs inside the kernel. Unlike RTC-Mon, FFPF does not provide SIP or RTP analysis, nor would implementing this functionality or that of other protocols be simple since IP defragmentation is not supported. The SCAMPI project [24] does provide a powerful API, but it was designed to work with specialized monitoring hardware and defaults to using *libpcap* when run on commodity hardware, yielding poor performance.

VIII. Conclusions

It is clear that monitoring is needed in order to ensure the quality of real-time communications over the Internet. In this paper we introduced RTC-Mon, the Real-Time Communications Monitoring framework, which provides an architecture for quickly developing efficient and powerful monitoring applications. More importantly, RTC-Mon is extensible, making it easy for a developer to add any advanced functionality needed.

We presented testbed results showing that the framework is quite efficient, tracking information for large amounts of RTP flows even for small packet sizes while keeping the CPU relatively idle. Further, we implemented a proof-of-concept application on top of RTC-Mon that, despite consisting of only 800 lines of code, can efficiently track a large set of VoIP quality metrics.

As future work we are looking into distributed monitoring solutions based on RTC-Mon, since SIP and RTP flows belonging to a call do not always follow similar paths. Further, we are also in the process of implementing plugins for other protocols such as RTCP in order to support other applications like IPTV.

References

[1] S. Agrawal, J. Ramamirtham, and R. Rastogi. Design of active and passive probes for VoIP service quality monitoring. In *Telecommunications Network Strategy and Planning Symposium*, 2006.

[2] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. Pathfinder: A pattern-based packet classifier. In *Operating Systems Design and Implementation*, pages 115–123, 1994.

[3] Andrew Begel, Steven McCanne, and Susan L. Graham. Bpf+: exploiting global data-flow optimization in a generalized packet filter architecture. *SIGCOMM Comput. Commun. Rev.*, 29(4):123–134, 1999.

[4] Robert Birke, Marco Mellia, Michael Petracca, and Dario Rossi. Understanding voip from backbone measurements. In *INFOCOM*, pages 2027–2035, 2007.

[5] Herbert Bos, Willem de Bruijn, Mihai Cristea, Trung Nguyen, and Georgios Portokalidis. FFPF: fairly fast packet filters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 24–24, Berkeley, CA, USA, 2004. USENIX Association.

[6] S. Mc Canne and V. Jacobson. The BSD packet filter: A new framework for user-level packet capture. In *USENIX Conference*, 1993.

[7] Xiuzhong Chen, Chunfeng Wang, Dong Xuan, Zhongcheng Li, Yinghua Min, and Wei Zhao. Survey on QoS Management of VoIP. In *ICCNMC '03: Proceedings of the 2003 International Conference on Computer Networks and Mobile Computing*, page 69, Washington, DC, USA, 2003. IEEE Computer Society.

[8] A. Conway. A passive method for monitoring voice-over-ip call quality with itu-t objective speech quality measurement methods. In *ICC*, 2002.

[9] Tilera Corporation. TILExpress-64 Card. http://www.tilera.com/products/tilexpress64.php.

[10] Luca Deri. Improving Passive Packet Capture:Beyond Device Polling. In *System Administration and Network Engineering Conference (SANE)*, 2004.

[11] Luca Deri. High-speed dynamic packet filtering. *J. Netw. Syst. Manage.*, 15(3):401–415, 2007.

[12] Endace. DAG network monitoring cards. http://www.endace.com/our-products/dag-network-monitoring-cards/.

[13] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM*, pages 53–59, 1996.

[14] S. Ioannidis, K. Anagnostakis, J. Ioannidis, and A. Keromytis. xpf: packet filtering for lowcost network monitoring, 2002.

[15] IXIA. IXIA - Leader in IP Performance Testing. http://www.ixiacom.com/.

[16] Michael Manousos, Spyros Apostolacos, Ioannis Grammatikakis, Dimitrios Mexis, Dimitrios Kagklis, and Efstathios Sykas. Voice-Quality Monitoring and Control for VoIP. *IEEE Internet Computing*, 9(4):35–42, 2005.

[17] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. *SIGOPS Oper. Syst. Rev.*, 33(5):217–231, 1999.

[18] T. Oetiker. RRDTool - About RDDTool. http://oss.oetiker.ch/rrdtool/.

[19] Ozvoip.com. Codec Support in VoIP Devices. http://www.ozvoip.com/voip-codecs/devices/.

[20] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916.

[21] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003.

[22] C. So-In. A Survey of Network Traffic Monitoring and Analysis Tools. Technical report, Washington University, 2008.

[23] A. Takahashi, H. Yoshino, and N. Kitawaki. Perceptual QoS assessment technologies for VoIP. *IEEE Communications Magazine*, July 2004.

[24] Information Society Technologies. IST-SCAMPI. http://www.ist-scampi.org/.

[25] Jacobus van der Merwe, Ramón Cáceres, Yang hua Chu, and Cormac Sreenan. mmdump: a tool for monitoring internet multimedia traffic. *SIGCOMM Comput. Commun. Rev.*, 30(5):48–59, 2000.