Universita degli studi di Pisa

Facolta di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea

Enterprise Voice-over-IP Traffic Monitoring

Relatori:

Luca Deri

Marco Danelutto

Controrelatore:

Augusto Ciuffoletti

Candidato:

Fusco Francesco

Anno Accademico 2007/2008

Dedicated in loving memory to my grandfather, Giovanni Fusco and to my parents Antonella Rossi and Pier Luigi Fusco.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Current IP-based real-time communication services have put into trouble traditional network monitoring paradigms and have imposed some additional requirements to network monitoring applications. The aim of this thesis is to demonstrate that the increased complexity of network monitoring can be managed with relatively little effort if the appropriate software instruments are used. In particular, by using a proper software framework it is possible to produce complex and efficient monitoring applications that are not affected by common problems such as having a monolithic architecture or being difficult to extend.

This chapter introduces the issues addressed in this thesis and explains why the VoIP service has been chosen as the reference monitoring field. Additionally the requirements and the scope of this thesis are identified.

1.1 VoIP overview

The evolution of computer networks leaded to the deployment of broadband IP-based networks. The cheapness and the increasing performance of data transmissions through the Internet favoured the development of real-time communication services over this infrastructure. These services are often in substitution and enrichment of traditional ones, which are usually provided over dedicated circuits.

An example is Television-over-IP (or IpTV): the use of the Internet as transmission channel instead of the common broadcast channels(air, cable, satellite) added interactivity to the television. This means letting the user choose what to watch and when (video on demand), but also let the user influence the program (e.g voting).

Another example is Voice-over-IP (or VoIP). As its simplest, Voice over IP is the transport of voice using the Internet Protocol (IP). VoIP networks are attractive to telecom providers and enterprises as the same network can be used for both voice and data services, reducing equipment, operation and maintenance costs. Moreover the use of IP enables the creation of converging voice and video services not available on traditional *Public Switched Telephony Network (PSTN)* networks.

The usage of IP networks allows greater mobility of users: VoIP telephone number is not geographically bound, permitting a user to be reachable even in different countries under the same number. Moreover Internet connection can be obtained with wireless access networks, so VoIP users can even use WiFi devices like PDAs, or VoIP phones to have access to enriched telephony services.

1.2 Merging network monitoring with service monitoring

Offering real-time services able to meet users expectations on top of a best-effort IP network is a challenge for a variety of reasons. Moreover the ever increasing complexity of services whose quality is tightly coupled with the network performance parameters, makes standard monitoring paradigms unsuitable to understand how users perceive the service quality. As a consequence, service providers are moving from a network centric monitoring approach toward a service centric monitoring approach that helps them to meet customer's high expectation from service quality levels, while controlling their costs efficiently.

Network monitoring and service monitoring should be tightly related. In that sense we can reasonably say that network monitoring and service monitoring are going to become an unique and integrated task, that can be called *service oriented network monitoring*.

VoIP service is the perfect candidate to understand in a better way the needs of service oriented network monitoring for a variety of reasons. First, VoIP is a real-time communication service whose quality is highly dependant on the performance of the underlying network. Second, VoIP users have high expectations regarding VoIP service quality since they used to have reliable and high quality telephony services provided by the consolidated PSTN infrastructure. Third, VoIP service degradation can be caused by lot of factors including network congestion, VoIP servers overloading, security attacks, misconfiguration and by their combination.

1.3 Thesis Motivation

My interests in network monitoring has been gradually instilled in me by my supervisor Dr. Luca Deri. During the last few years we had several discussions on network monitoring and we have been involved together in several network monitoring research projects.

These precious experiences made me conscious that network monitoring is a niche sector, property of highly skilled network specialists. This because the implementation of network monitoring software requires the knowledge of many protocols and highly optimised code. In order to achieve high performance usually ad-hoc and highly optimised solutions are preferred to solve specific problems. As a consequence efficient monitoring software usually offers little flexibility, it is very difficult to extend and does not easily allow the code to be reused.

My thoughts were confirmed by some NEC Network Laboratories researchers during the last summer. At the time they were designing advanced VoIP monitoring architectures. During one of the earliest discussions, I discovered that they were surprised by the fact that, even if VoIP was popular and widespread, very few instruments were available at the time to reduce the development time of complex VoIP monitoring applications. For me, it was not a surprise. That was the beginning of my thesis work.

1.4 Thesis Scope

The objective of this thesis is to define RTC-Mon, a new framework for real-time communications monitoring systems which can be used to implement

complex and efficient service oriented monitoring applications with relatively little effort. RTC-Mon is a framework for extensible monitoring applications which need to introduce the computation of several service oriented metrics.

RTC-Mon defines a new software architecture taking profit by previous research experiences and it adopts a mixed kernel space user space approach to grant ease of usage while it keeps high the performance.

1.5 Thesis Requirements

In the course of problem analysis, many requirements have been identified and grouped by their relevance. In Chapter 5 the requirements listed below will be used to validate this thesis work.

	Requirements
1	Extensibility
2	Ease of use and development
3	Flexibility
4	Scalable and high-performance applications
5	Promotion of reuse
6	Efficient resource utilization and ability to run on environments with limited resources
7	Commodity hardware

Table 1.1: Thesis requirements

1. *Extensibility*

   The framework must support application extensibility. This is necessary in dynamic fields, such as service oriented network monitoring where new protocols or metrics have to be introduced frequently. For this reason it is mandatory to provide extensibility mechanisms.

2. *Ease of use and development*

   Writing complex and efficient monitoring applications is a niche field, property of highly skilled network specialists. The framework must provide a valuable environment for both software developers with little networking experience and for network specialists.

3. *Scalable and high-performance applications*

   The framework must grant both ease of usage and high performance. The framework must allow the development of complex and yet efficient applications with limited effort. Moreover, the framework must be scalable in terms of number of concurrent calls.

4. *Flexibility*

   The framework must be flexible enough to accommodate disparate monitoring tasks.

5. *Promotion of reuse*

   Reuse of code are becoming increasingly important in the software industry. The framework should promote the software reuse and should automate many monitoring tasks such as packet dissection and protocol parsing.

6. *Efficient resource utilization and ability to run on environments of limited resources*

   It is a common belief that network monitoring applications performing complex tasks need a significant amount of resources in order to run.

A modern framework should be able to run on high-end server class machines, but also on small embedded boxes with limited extensibility.

7. *Commodity hardware*

   The proposed solution must not rely on expensive/exotic hardware, such as specialised network monitoring hardware to improve the performance. This increases the flexibility and makes easier the deployment task.

1.6 Thesis Outline

This chapter has covered the basic concepts necessary to understand and evaluate this thesis, and defined the scope and the goal of this work in addition to having established requirements.

Chapter 2 covers relevant research effort undertaken in the areas covered by this thesis: passive monitoring technologies and previous VoIP monitoring efforts.

Chapter 3 analyses the VoIP monitoring task under a service oriented network monitoring perspective. Firstly a better description of the service oriented monitoring task is given. Then the most important VoIP metrics to be measured and computed by a VoIP monitoring application are identified.

Chapter 4 covers RTC-Mon in details. The first part describes the design of a modular kernel space infrastructure suited for application layer protocol analysis. In the second part instead, it is presented an user space library called LibVoIP.

Chapter 5 evaluates RTC-Mon and validates it. A RTC-Mon based VoIP monitoring application is covered. After that, the performance of the

solution are evaluated and the experiments results presented.

In Chapter 6 conclusions about this work will be drawn, and possible future development in this field will be considered.

Chapter 2

Related work

This chapter will cover the most relevant VoIP monitoring efforts. Moreover passive monitoring technologies will be described.

2.1 VoIP monitoring

2.1.1 Issues of VoIP

The adoption of IP for carrying both voice and data introduces some issues in terms of quality and service reliability that did not affect traditional telephony networks. This because IP networks work in best-effort mode and was not designed to transport voice which imposes some constraints in terms of network quality such as network latency and packet loss. Furthermore VoIP has a very different architecture than traditional circuit-based telephony and these differences have also some impacts in terms of security. Users expect reliable and high-quality telephony services, thus, service providers or network managers should constantly monitor their infrastructures in order to detect

service quality degradation and take corrective actions in real-time to ensure that the degradation perceived by users is minimal.

In a nutshell, VoIP is an evolution of PSTN based services carried over IP networks. The outcome is that VoIP users expect the same quality and reliability as in PSTN networks. Providing a VoIP solution that offers PSTN quality and availability is a significant challenge in many aspects:

- *Security*: PSTN networks have been resilient to security attacks for many reasons. First, they have have been maintained as closed networks, where access is limited to carriers and service providers. Second, entry to the PSTN has traditionally been protected by a price which can be more than 100.000 dollars per year. As VoIP often uses public networks, it is necessary to provide stronger security mechanisms in order to prevent and detect attacks such as denial of service and identity theft.

- *Quality of service*: The quality of a telephone call depends on both signalling performance and voice quality. With signalling performance we mean the time needed to establish a call and release it. Voice quality depends on the codecs being used and network infrastructure performance. While PSTN networks ensure fixed delay minimum-distortion services, this cannot be applied to IP-based networks.

- *Reliability*: Customers expect high service reliability regardless of the nature of the communication, either PSTN or VoIP. PSTN networks were designed to achieve 99.999% availability or carrier class reliability.

- *Billing*: PSTN world have based their entire infrastructures on switched networks, thus call prices are a function on the resources exclusively used

(i.e. the circuits). Call detail records (CDR) are produced in real time by telephony switches and are used by both call accounting systems and fraud detection systems. In the early days of telephony, CDR only included fields like caller/called party numbers, date and time, and call duration. Recently CDRs include new fields such as call route. Even if CDR became more complex call duration and parties identities are often the only metrics used for billing. Instead, on packet switched networks, such as IP, the concept of resource usage has a different meaning as various services can be provided simultaneously sharing the same physical resource. Internet Protocol Detail Record (IPDR) has been introduced to describe (and bill) next generation digital services including IpTV, VoIP, TV on demand. Billable attributes such as latency, bandwidth and quality of service are supported by IPDR documents.

2.1.2 VoIP protocols

VoIP has a general meaning, grouping all the technologies made to allow bidirectional audio communications over IP based networks. The earliest VoIP services were deployed by using proprietary protocols and even today many vendors use proprietary protocols.

Skype, introduced in 2002, is one of the most interesting proprietary protocols in the VoIP area and maybe it is the protocol that made VoIP used by the masses. Despite its popularity, Skype internals are mostly unknown. The details of the protocols used and protocol messages as well are not public and the encryption makes even harder to reverse engineering the protocol.

SCCP, is Cisco's proprietary VoIP protocol, used to connect Cisco VoIP

phones to the Cisco Call Manager server. SSCP has been introduced to reduce the processing load on hard phones.

IAX is the Inter-Asterisk Exchange protocol that establishes connections between clients and Asterisk servers.

There are some advantages to proprietary protocols. Vendors can build features to address specific problems, as IAX has done to make it easier for VoIP to work through firewall. Manufacturer can improve the performance, as Cisco has done with Skinny or use different communications models, as Skype has done with the peer to peer concept. However the adoption of proprietary protocols results in a confusing array of products that do not interoperate and a maze of protocols to choose from when planning a VoIP deployment.

To enable the cooperation between different vendors both ITU-T and IETF have been working on the standardisation of protocols to be used in IP telephony services. The first widely adopted standard is the H.323 (by ITU-T), an umbrella recommendations that defines the protocols to provide audio-visual communication sessions on any packet network. The first version of H.323 specifications was published in November 1996. During the years the initial version has been revised with enhancements to better enable both voice and video communications over packet switched networks. The current version of H.323, referred as H.323v6 was published in 2006. H.323 is a very complex specification that covers different facets of communications over packet switched networks such as call signalling, security management, media transmission and the provisioning of supplementary services needed to address business communications expectations.

On the other side, IETF worked on the specification of the *Session Initiation Protocol (SIP)*, an HTTP like signalling protocol designed with flexibility

and simplicity in mind. A signalling protocol, in the context of packet switched networks, is a protocol which allows the management of sessions between different entities. Once the session has been established a media transport protocol is used to carry multimedia content. Even if SIP was published later (1999) than H.323 it now reached a wider diffusion, mainly because it is simpler than H.323. This work focuses on SIP and the Appendix A is dedicated to it.

Both H.323 and SIP employ the *Real-time Transport Protocol (RTP)* for media stream transport. RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio and video, over multicast or unicast network services. The Appendix B provides some additional information on RTP.

2.1.3 VoIP monitoring efforts

Much research has been carried in order to analyze the QoS network parameters for investigating the feasibility of VoIP services over current generation networks [59, 57, 8].

The passive analysis approach has been suggested in order to perform speech quality measurements.[25] performed several measurements to measure the voice service quality that current WiFi networks can offers. They performed the off-line analysis of media traffic in order to measure the voice quality.

Manosus et al [35] proposed a solution for the real-time measurement of voice quality. These measures are employed by their advanced PBX[1]. Spe-

[1]a PBX, or Private Branch Exchange is a business telephone system designed to deliver voice over a data network and interoperate with the normal Public Switched Telephone Network (PSTN)

cial agents were designed to perform the media analysis of VoIP calls. Some performance parameters, such as packet loss and jitter are measured in real-time so that the voice quality can be measured while a call is active. However the goal of the work is to provide monitoring facilities to their PBXs, which supports very few concurrent calls.

In [12] De Lima et al. propose a framework for voice quality monitoring. The main limitation of the framework is that it requires customized user agents in order to provide voice quality measurements. In their framework user agents are responsible to record and then analyze the media traffic. This means that the framework is practically useless, as in many of the existing VoIP networks hardphones (such as WiFi phones) are used.

All of the previously described efforts only take into account the measurement of voice quality. However voice quality is not the only one parameter that have to be considered in order to measure the quality of a telephony service. The aim of the work presented in [25] is to measure the call setup time, which is signalling performance indicator. However this work is limited to the call setup time and does not cover the measurement of network impairments.

The work in [3] present the design of passive and active probes capable to measure network impairments in order to compute the VoIP quality. One of the biggest advantage of this solution is that is allows signalling performance indicators to be computed. This work is targeted to enterprises and service providers who have to monitor in real-time their infrastructures in order to keep high the quality of VoIP services. However this work presents some limitations as very few signalling performance indicators are taken into account.

Unfortunately all of these efforts are in some sense limited, since only few indicators are taken into account. None of them offer extensibility mech-

anisms in order to accommodate the measurement of new indicators. Moreover none of the previous works explicitly cover the performance issues that the passive analysis of VoIP traffic imposes: the media traffic is carried over dynamically assigned ports and it is composed by small size packets. The adoption of capture cards, described in Section 2.4, is suggested in order to improve the performance. The drawback of this solution is the price which can be 10/100 times higher than commercial network adapters. Furthermore, expensive capture cards are capable to accelerate the capture phase, but do not provide effective and scalable filtering mechanisms. Moreover, it is worthwhile to note that some monitoring applications may need to run on small-embedded boxes with limited extensibility.

No relevant research has been carried on VoIP monitoring frameworks enabling fast development of complex and highly specialized passive VoIP monitoring systems. Furthermore, even if several SIP libraries already exists in the software scenario [24, 58], none of them has been developed to implement passive network probes. In fact the goal of those libraries is to enable faster development of SIP agents. In any case none of the library provides facilities to analyze the media traffic quality.

2.2 Packet capture

Packet capture is a commonly used passive monitoring technique which involves the real-time collection of packets as they travel over the networks. Packet capture probes are network probes that decode the captured traffic and perform some analysis on it. Passive monitoring accuracy and reliability depend on the captured traffic portion's over the total. More packet the probe

is able to analyze and more precise is the information it is able to gather. When the probe is not able to not to loose packets, a smart discarding mechanism can be adopted in order to have quantifiable accuracy. This mechanism is usually referred as packet sampling [16, 15]. However in case of service oriented network monitoring, such as VoIP monitoring, loosing packets is not acceptable. In fact packet losses can alter the values of some computed metrics (e.g. the stream quality). In the worst case, loosing packets can mean that a VoIP call is not even discovered.

The performance of a network monitor is most simply defined as its ability to not loose packets while still providing sufficient CPU to decode packets, analyze protocols and store or visualise the network traffic[37]. Packet handling can be characterized by the Maximum Loss Free packet reception Rate (MLFR) measured in packets per second for a fixed packet size. The packet reception rate is determined by several bottlenecks like interruptions handling, context switch and memory copies from kernel-space to kernel-space and from kernel-space to user space. To better understand the capture process, a brief overview of the journey of a captured packet inside the Linux Kernel is given in the following paragraph. This work focuses on Linux for its importance in research community especially in the field of network monitoring. Moreover, Linux is the fastest growing operating system in the embedded domain and supports a wide range of network interface cards and platforms.

The Linux networking stack is composed of different layers: the NIC driver, protocol processing and socket layer. After a packet arrives in the network interface card's FIFO receive buffer, the network interface card (NIC) transfers the packet by Direct Memory Access (DMA) to the kernel memory and interrupts the host processor. As a response to the interrupt the host

processor run an Interrupt service routine (ISR). The ISR moves the packet from the DMA memory region to a packet queue implemented in regular kernel memory and raises a *softirq* which is responsible to perform the protocol processing. After the fulfilment of this task, the softirq task inserts the packet in a socket buffer and eventually notifies the scheduler to wake up a blocked user process. After waking up, the user process completes the reception task using a *read()* system call.

The receiving mechanism uses three different buffers (DMA, the packet queue, the socket buffer) and employs three task threads which run on the host on three different contexts: ISR, softirq and user space.

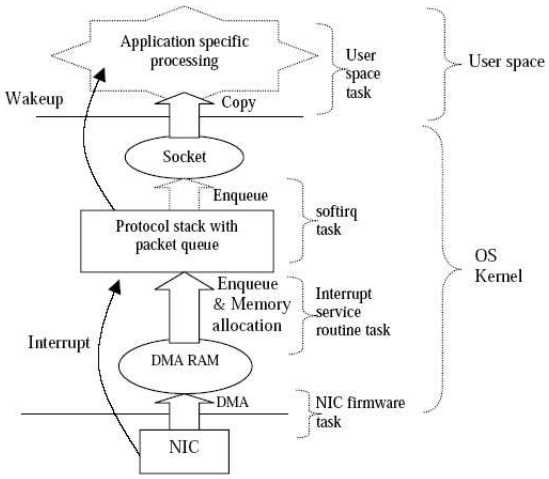


Figure 2.1: Linux 2.6 network stack overview

Given the described layered architecture (Figure 2.1) the issue that limits the packet handling rate are:

- **interrupt service overhead**: it includes the time consumed in context switching, memory cache and storing/retrieving process state. For ear-

lier Linux versions this cost was per packet since an interrupt is raised whenever a packet is received.

At high arrival packet rate servicing one interrupt for each received packets can lead to the "receive livelock"phenomena. The host is constantly overwhelmed by constantly servicing interrupts, having no more spare CPU cycles to perform any other useful operations on the received packets. As a result, the packet reception rate substantially decreases since only a subset of received packets can be processed and most discarded.

In order to mitigate the problem an hybrid interrupt-polling mechanism previously suggested by j.Mogul et al.[39] has been introduced in Linux since kernel 2.4.20 with the development the NAPI [54] driver interface. Thanks to NAPI the behaviour of NAPI aware network interface cards depends on the traffic load. Under low packet rate reception an interruption is raised for each received packets. However the number of interruptions are substantially reduced under high packet arrival rate conditions thanks to the adoption of a polling mechanism[49].

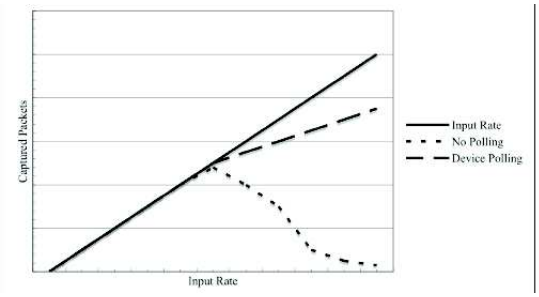


Figure 2.2: benefits of a polling mechanism (taken from [13])

As depicted in Figure 2.2, a polling scheme can increase the capture rate and mitigate the livelock phenomena.

- **data copies**: copying packets from the DMA region to the regular kernel memory and the subsequent copy of from kernel space to user space take significant CPU time. Some solutions, described later in this chapter have been developed in order to reduce the number of copies.

- **redundant protocol processing**:

- **kernel to user space boundaries crossing**: the received packets are consumed by a process in user space. Crossing the kernel space and user space boundary involves a data copy and the execution of a system call, a context switch. The response time for doing those operations is significant.

- **buffer overflow**: packets can be dropped because of buffer overflow. The buffers involved are the NIC FIFO, the DMA region and the packet queue. Those buffers are managed by three different producer-consumer pairs: NIC firmware-ISR task, ISR-softirq task and softirq-user space task.

  When a packet arrives in the NIC's FIFO buffer it needs to be transferred into the host's DMA buffer. If the buffer is full the NIC cannot offload its FIFO buffer and thus the FIFO buffer can be filled up. If the FIFO buffer is filled the NIC start dropping packets. Similarly user space task jitter may cause buffer overflow in the packet queue. If the packet queue if filled up it can cause buffer overflow in the DMA region.

The number of copies (kernel space to kernel space and kernel space to user space) can be minimised using kernel packet filters, which are responsible

to discard unwanted packets as soon as possible. More details regarding kernel packet filters are given in Section 2.3.

During the years, many research has been carried in order to improve the capture performance using custom hardware or even commodity hardware. The following sections give a brief overview of some of the most important contemporary solutions.

2.3 Packet filtering

Packet classification can be seen as the categorisation of incoming packets based on their content according to specific criteria that examine specific portion of a packet. Packets are classified into flows containing packets matching the same criteria. The criteria are comprised of a set of rules that specify the content of specific packet fields to result in a match. Fields may be header fields from layer 1 to layer 7.

The goodness of a packet classifier algorithm is usually evaluated on the following criteria[28]:

- *search speed*
- *storage requirements*
- *update time*: as classifier changes, data structures may need to be updated or reconstructed from scratch. Data structures that need a completed reconstruction can be called "pre-processing".
- *scalability in the number of fields used for classification and in the number of different rules*

```
black:/home/fuscof# tcpdump -d udp
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 4
(002) ldb      [20]
(003) jeq      #0x11            jt 7    jf 8
(004) jeq      #0x800           jt 5    jf 8
(005) ldb      [23]
(006) jeq      #0x11            jt 7    jf 8
(007) ret      #96
(008) ret      #0
black:/home/fuscof#
```

Figure 2.3: A sample BPF program

- *flexibility in specification*: rules can be complex (e.g. may allow wildcard, prefixes and so on) or be simple. Usually more complex rules involve the adoption of more complex data structures.

Firstly proposed by Mogul, Rashid and Accetta in 1987[38], a packet filter is a programmable abstraction for a boolean predicate applied to a stream of packets in order to select some specific subset of the stream. Packet filtering is a packet classification problem.

The BPF (Berkley Packet Filter) [36] is the most widely used solution to the problem. Every modern operating system provides the filtering mechanism. BPF includes a virtual machine capable to execute programs. Each program is an array of virtual machine instruction that sequentially execute some action over the virtual machine. The popular *tcpdump*[26] software, allows bpf programs to be easily inspected. Figure 2.3 shows how tcpdump compiles a filter to select only udp packets.

The execution of one of the simplest filter needs the execution of 8 instructions. More complex filters, such as filtering all http packets, take more than double the number of instructions.

After BPF, a large number of evolution have been produced by the research community. Some of them, such as BPF+[4] was developed in order to optimize the evaluation of complex filters. The Mach Packet Filter (MPF) [63] enhances the BPF model in order to optimize the case of multiple filters using similar patterns. The aim of DPF [20] is to improve the performance of BPF by the adoption of dynamic code generation techniques. Thus the filtering code is native and no more executed by a virtual machine. However all of these efforts basically present the same limitations:

- *low scalability*: only very few filters can be specified
- *changing the filter set need a reconfiguration, thus can lead to packet loss*
- *instruction proportional to the complexity of the filter*

Those limitations make the previous approaches unsuitable to effectively filter multimedia traffic, since the majority of multimedia applications, such VoIP, uses dynamically assigned UDP or TCP port for media transfers. If a multimedia application usually pick up a port from a small range of port numbers, it is possible to specify the entire port range. In any case this is only a partial solution to the problem, since non multimedia packets need to be discarded in user space.

To improve the filtering effectiveness of multimedia traffic *mmdump*[60], a multimedia oriented *tcpdump*, suggests the adoption of signalling aware session trackers whose task is to perform the signalling analysis in order to dynamically reconfigure the underlying BPF filter. Even if the advantages of performing application layer protocol analysis are clear, the work has some

limitations. First, *mmdump* suffers from the same filter reconfiguration problem of BPF: if new sessions are established at an high rate, the recompilation may cause packet losses. That's why the main purpose of the software is to analyze prerecorded traffic traces. The adoption of more advanced filtering techniques capable to reduce the reconfiguration time, such as the one proposed in [14] can only mitigate the problem. Second, it does not offer methods to select multimedia sessions by content (e.g. specify the caller of the VoIP calls to be monitored). This means that signalling traffic is always carried in user space regardless of its importance.

Content based filtering, which means filtering using fields from application layer protocols requires much more resources and it is much more difficult to implement. In fact, there are hundreds of application layer protocols currently used; there are multiple versions of the same protocol; some protocols, such as SIP, are text based whereas others are byte driven. The most common approach to deal with application layer protocol filtering is to use a *signature database* that is an ensemble of regular expressions for each text driven protocol and encapsulation description for byte driven protocols. The creation of this big database is not easy, thus there are some projects specialized in this task [33]. Signature based methods are useful to discover if a particular packet belongs to a certain application level protocol, but offer limited support for building more accurate filtering mechanisms (e.g. having only SIP packets coming from the user *"Francesco"*).

2.4 Monitoring hardware

Monitoring very high speed links using commodity hardware is difficult due to relatively slow buses and memories. The processing power of current general purpose systems is no more sufficient to passively monitor current high speed links (10 Gbps and above) and this trend is supposed to not change during the following years.

For these reasons the industrial research focused on specialised hardware capable to alleviate the burden on the resources used by monitoring stations. This section gives a brief overview of specialised hardware devices designed to overcome those limitations.

2.4.1 Capture cards

Are usually referred as capture cards some feature rich network interface cards (NICs) explicitly designed for passive monitoring purposes [18, 40]. The term capture cards is referred to the ability to offload the host system from the capture process. This increases the number of spare cpu cycles to be used by network monitoring applications running on the host system. As the *libpcap*[1] library became the "de facto" standard of network capturing on UN*X and Windows as well[62], capture cards usually provide enhanced libpcap capture libraries so that every libpcap based monitoring software can take advantage of the underlying specialised hardware without any porting efforts.

Beside accelerating the capture phase, capture cards provide some advanced features targeted at the passive monitoring domain such as high precision packet time stamping and hardware packet filtering. Hardware filtering allows wire speed filtering of packets matching a criteria. However most of

the cards allow to specify very few filters (8-64) as they are limited by the space available on the silicon/RAM used to store filters. Moreover most of the cards internally adopt Field Programmable Gate Array (FPGA), thus a filter reconfiguration usually require a general reconfiguration of the hardware. This operation can take seconds.

In a nutshell, capture cards offload the hosts from the capture process while allowing standard monitoring software to take advantage of the underlying hardware. However capture cards are expensive and do not provide filtering mechanisms to be used in a monitoring domain such as VoIP where several thousands of different filters have to be inserted and removed in real-time.

2.4.2 Network processors

Network processors are integrated circuit with specific features targeted to the networking applications domain[32]. Network processors (NP) are specialised to support the implementation of network applications at the highest possible speed. Network processor are more flexible and less expensive than custom *ASICs (Application-specific Integrated Circuit)* because of their programmability. In fact designing and manufacturing custom ASICs is very expensive. Therefore the ability to use a single device for various applications is an important factor. By using network processors the same physical device can be used while different software releases offer different functions.

Although the architectural design of the various network processors often differ significantly, all are optimized to exploit the inherent parallelism present in network workloads. Due to their highly specialised and unconventional architectures network processors create new challenges for the software

engineers. Network processors are usually programmed using low level assembler like languages. At best, higher level C-like languages are provided to reduce the development time.

It is worthwhile to note that being programmable do not necessary means that network processors are the best solution for every network workloads. Although they are programmable they do not offer the same flexibility of traditional general purpose CPUs.

2.4.3 Programmable cards

Network processors are usually multi-core processors, augmented with network specific instructions, hardware assists and memories. While these specialized NP features might improve performance, they come at a cost of reduced generality and familiarity. Programmable cards was introduced in order to overcome those limitations while keeping the performance high. This is possible due to the arrival of multi-core general purpose processors in the networking domain [7, 11].

The adoption of a multi-core general purpose processor in networking devices provides several advantages. The most obvious advantage is the programmer productivity improvement. Unlike NP based cards, programmable cards are usually programmed in C language and they are able to run customized version of general purpose operating systems (usually Linux). This means that it is possible at least in theory to build any standard Linux application for them. However porting applications on top of them is not so easy as it is supposed to be.

2.5 Monitoring frameworks and libraries

As stated in Section 2.1.3 little research has been carried on VoIP monitoring frameworks. Thus it is important to describe the most relevant general purpose passive monitoring frameworks and libraries as they would represent the starting point to design and implement a VoIP monitoring application.

2.5.1 PF_RING

PF_RING[13] is an high packet rate capturing solution that improves the standard Linux kernel capture performance using commodity hardware. PF_RING can take advantage of NAPI aware drivers and do not require any special hardware, thus it can be used with every network interface card supported by the standard Linux kernel.

PF_RING defines a new kind of socket explicitly designed for packet capture. The socket makes use of a memory mapped buffer, implemented as a circular FIFO, which is shared between the kernel and the user space application. The technique, initially proposed by P.Wood [47], allows the reduction of per packet costs by reducing the number of copies.

Moreover, the kernel networking core has been modified in order to completely bypass the standard protocol processing. As a consequence the journey of the packet inside the kernel is substantially reduced.

PF_RING comes with a kernel patch and an user space library. The patched kernel provides a loadable kernel module, the *ring* module, that allows the usage of the PF_RING socket type. The circular buffer size can be customized using two different module parameters: *bucket_len* and *num_slots*. The first parameter represents the circular buffer length whereas the second

represent the maximum length of the captured packet. If the packet size is greater than the bucket_len, the first bucket_len bytes of the packet are stored in the slot. Decreasing the bucket len below the MTU[2] leads to the reduction of time spent on copies for monitoring applications that need only few bytes for each packets (e.g. NetFlow probes just need the packet up to transport layer).

The user space library, called *libpfring*, enables fast development of packet capture applications. A PF_RING enhanced version of libpcap is provided so that every application written on top of libpcap can benefit from PF_RING.

PF_RING substantially increases the packet capture performance of the standard Linux operating system and it is considered both by the research community and by the monitoring industry as one of the most efficient packet capture solution that does not require expensive hardware to run.

However PF_RING still presents some limitations as the performance offered is largely dependant on the size of the captured packets. PF_RING really shines when the captured traffic is composed by large size packets but the performance rapidly decreases when the packets size gets smaller.

2.5.2 FFPF

Fairly Fast Packet Filter (FFPF)[6] is a network monitoring framework designed for speed, scalability (in terms of number of applications) and flexibility. One of the FFPF's main goal is to use commodity hardware.

Like *PF_RING* it employ shared memory buffers in order to reduce

[2]Maximum Transmission Unit

system load due to packet copying and context switching. Moreover, like *xPF* [29], allow the execution of monitoring programs inside the kernel. The performance of computationally intensive operations, such as content based filtering is high due to the adoption of *external functions*. External functions, implemented as Linux kernel modules, allow the framework to be extended and have been introduced for monitoring dynamic flows.

An implementation of the popular *pcap* packet capture library is provided to ensure backward compatibility with many existing tools. The framework supports several filtering languages including the popular BPF and two FFPF specific languages called *FPL-1* and *FPL-2*. However the goal of FFPF is not to optimize the filter expressions and it would not be simple to handle a large amount of different RTP streams.

2.5.3 SCAMPI

SCAMPI (A Scalable Monitoring Platform for the Internet) [30] is a scalable and programmable architecture for monitoring multigigabit networks. The main goals of the project are the following [10]:

- *definition of a common monitoring API*: SCAMPI based monitoring applications are written using the Monitoring API(MAPI) library. One of the goal of the project is to decouple the development of the monitoring applications from the monitoring environment. This allows the development of portable monitoring applications that can benefit from the features offered by different hardware devices.

- *expressive power*: the MAPI natively supports some advanced features, such as packet sampling, IP defragmentation and TCP reassembly, that

other monitoring libraries does not provide. Moreover it supports the same display filter language introduced by the popular Wireshark[27] free network protocol analyzer. Display filters support hundreds of application layer protocols, however they cannot be considered application layer packet filters, since the filtering is completely done in user space.

- *scalability through special purpose hardware*: the SCAMPI architecture can be used on top of commodity hardware. However the main goal of the project is to use specialised monitoring hardware and to provide an API taking benefits from heterogeneous specialised monitoring hardware.

If the monitoring station does not provide any specialised hardware the MAPI is implemented on top of the traditional *libpcap* library, so that SCAMPI architecture can run on top of commodity hardware, but with substantial performance impairments.

2.6 Why a new passive monitoring framework?

Monitoring VoIP in a passive way is a challenge since it imposes orthogonal requirements coming from network management and software engineering such as high performance, flexibility and extensibility.

Flexibility and extensibility requirements are needed to measure different kind of metrics, or indicators. Many of the VoIP works presented are capable to offer a very limited set of indicators. Some of them are only capable to provide voice quality measurements, while others measure very few signalling indicators such as the call setup time. However VoIP monitoring applications should be capable to provide a large set of indicators of which the

call setup time is just an example.

Performance requirements come from the VoIP traffic pattern. VoIP traffic represents one of the worst traffic pattern to capture since it is composed by a great number of small size packets. VoIP traffic is also very hard to filter since real-time data such as voice or video is carried over streams using dynamically assigned ports.

Capture and filtering challenges imposed by VoIP traffic are only partially solved by the adoption of specialized hardware for passive monitoring. Monitoring cards improve the capture phase but usually offer limited support for filtering. Moreover they are much more expensive than commodity network interfaces and due to their costs cannot always be considered a viable way.

The software solutions presented in this chapter are the most promising solutions for the design and the development of VoIP monitoring applications. However, from Table 2.1 it is possible to conclude that it would not be so easy to implement a VoIP monitoring application on top of them as:

- none of them natively provide any facility to analyze the VoIP traffic
- their filtering mechanisms offer very little scalability and this is a big limitation for a service using dynamically assigned ports
- they are general purpose passive monitoring frameworks that are suited for experienced network monitoring developers

The above limitations found in existing solutions, have been the driving force for the author for designing and implementing a new real-time communication monitoring framework described in the Chapter 4.

	PF_RING	**FFPF**	**SCAMPI**
Ease of usage	low	low	low
Commodity hw. support	Yes	Yes	Yes
Specialized hw. support	No	Yes (Intel IXP NPU)	Yes (DAG,SCAMPI, Intel IXP NPU)
Reference hw.	commodity	commodity	specialized
Filtering language	BPF	BPF FPL2	BPF wireshark display filters
Filtering scalability (commodity hw.)	low	low	low
Content based filtering	No	partially[a]	No
libpcap compatibility	Yes	Yes	Yes
Additional features	IP defragmentation IPv4 parsing	trusted compiler FPL2	packet sampling IP defragmentation TCP reassembly
kernel extensibility	No	Yes	NO[b]
VoIP monitoring facilities	**No**	**No**	**No**

Table 2.1: Monitoring technologies comparison

[a]can be implemented using *using FFPF's external functions*

[b]when SCAMPI is used with commodity hardware the MAPI is implemented on top of libpcap (user space)

Chapter 3

VoIP service monitoring

This chapter will describe the benefits offered by service network monitoring and analyze the VoIP monitoring task under this perspective.

Then, it will show why this VoIP service oriented monitoring is helpful for network manager in order to perform an effective VoIP service management.

At the end, the major challenges to solve will be presented.

3.1 Service oriented monitoring

Service oriented network monitoring goal is to define and measure the Quality of Experience (QoE) of the provided service. QoE is a collective term to form a measure of the quality of a service and include all aspects of service: its performance, level of customer satisfaction over the total and so on. Determining the QoE of a service provides a discriminator between various type of services and leads opportunities to balance the level of quality offered against price and customer expectations. QoE itself it is not measurable. Thus, an external methods is needed in order to have objective measure of the service.

Service Level Agreements (SLAs), which have been widely used by telecommunication service providers, are now being considered for non communication network services and are being adopted to define the agreed performance and quality of the service. Unlike QoE, which is a perceptive measure of the service, SLAs refers to the definition, measurement and reporting of objective measures. Thus, the key concept is to map perceptive measures from QoE into objective measures for SLA.

There is a difficulty in mapping service specific measure to technology specific parameters that are more easily measured and reported. As a consequence, traditional SLAs have focused, almost solely, on the performance of the supportive service.

However, the growth of service-oriented management leaded to the requirements of new indicators that focus on service quality rather than network performance. These concepts were introduced by the Wireless Services Handbook(GB 932). These new indicators *Key Quality Indicator(KQI)* and *Key Performance Indicator(KPI)*, provide a measurement of specific aspect of the service performance leading to a more complex and more precise SLAs definition.

Defining the key quality indicator and performance indicator is one of the most important aspect of service oriented monitoring as they are the metrics used to model the user's quality of experience and to perform conformance test against SLAs. KQIs derive from a number of sources, including the performance metric of the service or underlying support service as KPI. KQI and KPI may have an upper and a lower error threshold and an upper and a lower warning threshold. The mapping between the KPI and the KQI may be empirical or formal. In order to better understand the indicators and how they

are related we can consider a simple example. Suppose you have to provide a typical client/server service. The service performance is measured by the application response time (ART) key performance indicator. ART is measured that the moment from the user enters an application query, command or so requiring a server response to the moment the user receives the response and can proceed. The ART depends on servers load and on network bandwidth, since multiple applications compete for network resources. However in order to define the overall service quality one of the KQIs listed in Table 3.1 can be used:

KQI	**Description**
AART	average ART
MART	maximum ART
SARTP	percentile of request with the ART below the threshold

Table 3.1: sample key quality indicators

Given KQIs, KPIs and SLAs, the monitoring task can be defined as the continuous process of measuring indicators in order to check the SLA conformance. The measurement activity involves the measure of KPIs which are collated and combined in order to have the required KQIs. The relationship between KQIs, KPIs and SLAs are depicted in Figure 3.1.

The following Section analyses the VoIP monitoring task under a service oriented network monitoring perspective.

3.2 VoIP service oriented monitoring

We usually tend to associate the quality of a voice service, like VoIP, to the voice quality that the service is capable to offer. However this simplistic view

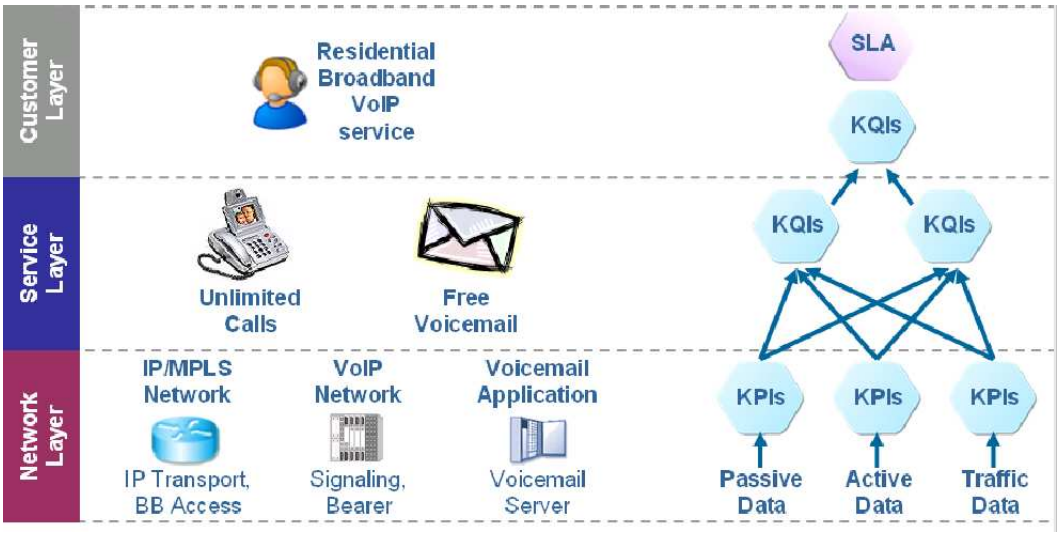


Figure 3.1: service oriented network monitoring overview

of service quality does not take into account other important indicators that make a voice service an high quality service. We use to measure the service quality with voice quality simply because traditional telephony services are consolidated and capable to offer highly reliable, high accuracy and high speed services.

Quality of Service comprises requirements on all the aspects of a connection, such as service response time, reliability, availability and so on. The quality of a voice service needs to be evaluated from the call attempt to the call termination, as depicted in Figure 3.2. This obviously includes also the measurement of voice quality, but it is not limited to it.

Signalling indicators are covered in Section 3.2.1 whereas the voice quality indicators are covered in Section 3.2.2.

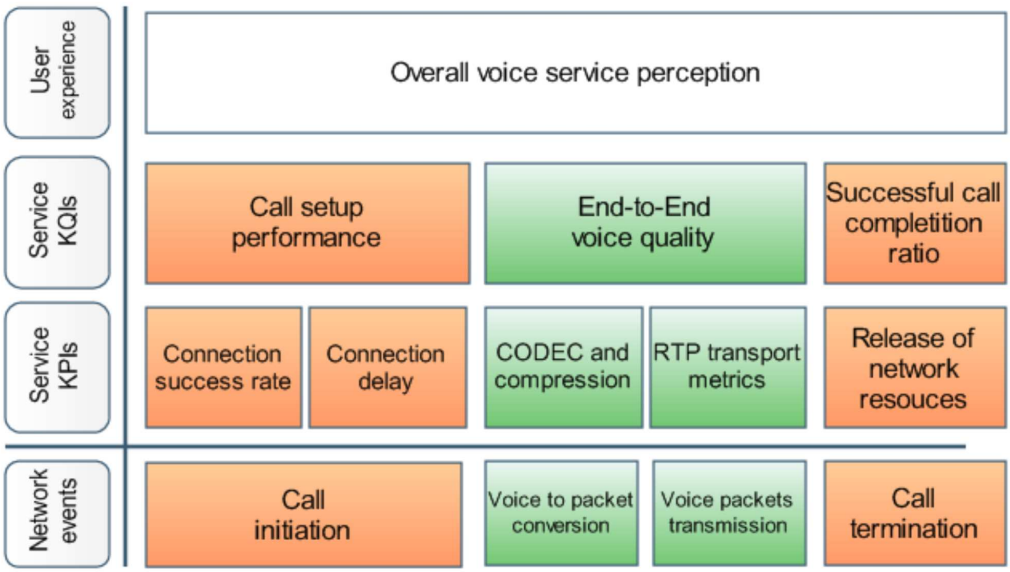


Figure 3.2: VoIP service monitoring overview

3.2.1 Signalling indicators

Traditional telephony services based on PSTN networks have already a consolidated of standard signalling performance metrics[17]. During the last few years the signalling performance of SIP based VoIP networks has been measured using non standardized metrics. At best, metrics coming from PSTN based services were adapted even for VoIP in order to have comparable results between traditional voice services and next generation IP based voice services.

More recently, D. Malas proposed a definition of a standard set of SIP signalling metrics [34]. The metrics, listed in Table 3.2, introduce a common foundation for understanding and quantifying performance expectations between service providers, vendors and the users of services based on SIP. It is worthwhile to note that the measurements are affected by variables external to SIP since their scope is to catch an end-to-end performance. The external

variables may include network connectivity, router and switch performance and server and hardware performance.

Those metrics are briefly described below:

- **Registration Request Delay (RRD)**: the registration request delay is used to detect impairments causing delay in responding to an user agent register request. The output of this metric is a numerical value and indicate milliseconds. This metric measures the performance of the Registrar server and includes the delay caused by user location database access.

$$RRD = Timeof FinalResponse - Timeof REGISTERRequest$$

RRD can be averaged using the following formula:

$$ARRD = \frac{\sum_{i=1}^{numregister} RRD_i}{numregister}$$

- **Session Request Delay (SRD)**: this metric is similar to the Post Dial Delay (PDD), which is used by traditional telephony services. The output is a numerical value representing milliseconds.

$$SRD = Timeof StatusIndicativeResponse - Timeof INVITE$$

- **Session Disconnect Delay(SDD)**: the session disconnect delay is utilized to detect impairments delaying the time needed to end a session.

$$SDD = Timeof 2XX or Timeout - Timeof CompletionMessage(BYE)$$

- **Session Duration Time (SDT)**: the metric is used to measure the duration of a session. In telephony services represent the Call Hold Time (CHT). Measuring the session duration time and averaging it is useful since short duration sessions can be caused by poor audio quality calls.

$$SDT = Time of BYE or Timeout - Time of 200 OK response to INVITE$$

- **Average Hop per Request (AHR)**: AHR is defined as the number of hops traversed by and INVITE or MESSAGE request and it is measured in number of hop. An high AHR can be a symptom of inefficient routing or misconfiguration.

- **Session Establishment Rate (SER)**: this metric is used to detect the ability of an user agent or a proxy to successfully establish new sessions. This metric is similar to the Answer Seizure Ratio (ASR).

$$SER = \frac{\#of INVITE Requests w/associated 200 ok}{(Total \#of INVITE Requests) - (\#of INVITE Requests w/3XX Response)}$$

- **Session Establishment Efficiency Rate(SEER)**: this metric is similar to the SER and it is computed in the same way. The only difference is that the numerator represents the number of INVITE requests resulting in a 200 OK, 480, 486 or 600.

- **Session Defects (SD)**: it is a measure of failures in dialogue processing. These failures response are in response to initial session setup requests, such as INVITE. The draft suggests the usage of the following SIP error responses to mark a session as defective:

- 500 Internal Server Error
- 503 Service Unavailable
- 504 Service Timeout

The output of this metric is a numerical value representing the percentile of session defects.

- **Ineffective Session Attempts (ISA)**: ineffective session attempts occur when a proxy or an agent internally releases a setup request with one of the following response codes:

  1. 408 Request Timeout
  2. 500 Server Internal Error
  3. 503 Service Unavailable

  ISAs can be caused for example by congestion. The metric has to be calculated as a percentage of the total session setup requests.

$$ISA_{\%} = \frac{\#ofISA}{Total\#ofSessionRequests}$$

- **Session Disconnect Failures (SDF)**: session disconnect failures occur when an already established session is terminated in presence of a failure condition. A typical failure condition is the loss of media related to an active session which is reported by media gateways to user agents. The failure condition causes the early termination of the session with a special BYE message that indicates the abnormal condition in the `Reason` header field[56].

The SDF is a numerical value, so the metric is computed as a percentage of total session completed successfully.

$$SDF_{\%} = \frac{\#ofSDF}{Total\#ofSessionRequests}$$

- **Session Completion Rate (SCR)**: a session completion is a SIP dialogue that ends without failing due to lack of response from a proxy or UA. For example a session completion fails when an INVITE is sent from a UAC, but the related UAS does not respond to the UAC.

  SCR is defined as a percentage that can be computed using the following formula.

$$SCR_{\%} = \frac{\#ofSuccessfullyCompletedSessions}{Total\#ofSessionRequests}$$

- **Session Success Rate (SSR)**: sessions can fail due to ISA or SDF. The session success rate, most commonly known as Call Success Rate(CSR) in telephony applications, is defined as the percentage of successfully completed sessions and can be computed using the ISA and SDF percentage.

$$SSR = 100\% - (ISA_{\%} + SDF_{\%})$$

3.2.2 Voice quality indicators

The voice quality of a telephony call depends on many factors including user equipments, adopted codecs and network performance. The *Mean Option Score(MOS)* is one of the most commonly used voice quality indicator. It pro-

KPI	Name
RRD	Registration Request Delay
SRD	Session Request Delay
SDD	Session Disconnect Delay
SDT	Session Duration Time
KQI	**Name**
AHR	Average Hop per Request
SER	Session Establishment Rate
SEER	Session Establishment Efficiency Rate
SDM	Session Defects per Million
ISA	Ineffective Session Attempts percentile
SCR	Session Completion Rate
SSR	Session Success Rate

Table 3.2: SIP End-to-End Key Quality Indicators (KQI) and Key Performance Indicators (KPI)

vides a numerical indication of the perceived quality of received audio streams.

Several methods were introduced in order to derive the MOS value from objective measurement. Some measurement techniques are intrusive [46, 45] whereas others allows to compute the MOS in a totally passive way [44].

Some non-intrusive measurement techniques, such as the E-model, standardized by the ITU as G.107[23], allow to compute the MOS with simple formulae rather than with the analysis of voice signals, so that the MOS computation requires very few resources. Those formulae allow to compute the MOS using the codec, the end-to-end delay and the packet loss.

Beside packet loss and end-to-end delay there are also some other performance network parameters that impact on voice quality. They are the bandwidth and the jitter. Those network performance parameters are described below.

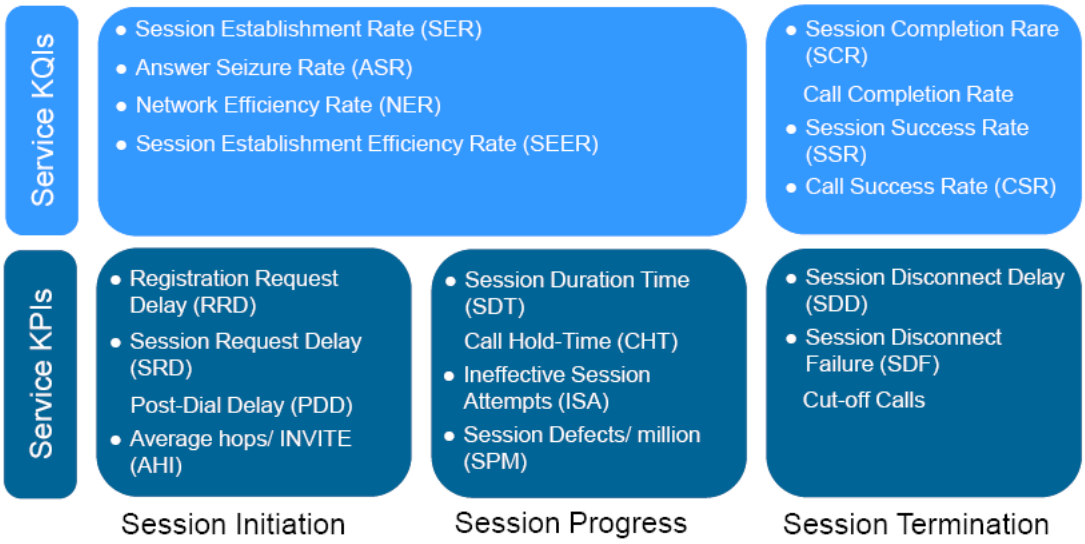


Figure 3.3: Signalling indicators

Delay

In real-time bidirectional communications keeping the end to end delay low is very important. Excessive end to end delay in voice communication have two side-effects:

- *Echo*: it is caused by the signal reflections of the speaker's voice from the far-end telephone equipment back into the speaker's ear.
- *Talk overlap or Hello effect*: it is the problem of one talker stepping on the other talker's speech

The end to end delay is the sum of delays derived by multiple sources:

- *Accumulation delay*: it is caused by the need to collect a frame of voice samples to be processed by the voice codec. It is related to the type of codec used.

- *Processing delay*: is a function of both processing power and codec used. It is the time needed to encode and collect the encoded frames into a single network packet. Often multiple multiple encoded frames are collected in a single packet to reduce the packet network overhead.

- *Network delay*: it is caused by the physical medium used to transport the voice data and by the protocols used. It is a function of link capacity and the processing that occurs as the packet transit the network.

- *Jitter reduction delay*: it is introduced by the procedure used to reduce the effect of jitter, described later.

Jitter

Jitter is defined as the variance of the one-way delay. When jitter is high, packets arrive in chunks. A jitter buffer is usually used by the receiver to reduce delay variations. Call quality is not affected by jitter fluctuations as long as the jitter buffer can mask fluctuations. Latency constraints, which depend on the codec being used, impose a buffer flush at least 150 ms that usually corresponds to a few packets. Jitter can be controlled by network traffic engineering on routers and firewall, so that a preference path is reserved to voice packets. Nevertheless the de-jittering process, that also includes packet reordering, is usually performed on VoIP terminals.

Bandwidth

Bandwidth requirements depend mostly on the codec. A *codec (COder/ DE-Coder)* is an algorithm used encode audio or video content before sending it on the network. Codecs are used to represent the original data with less bits

while keeping the quality high. The effective bandwidth requirement for a particular codec is higher than the bit rate of the codec as the overhead of all network protocols (RTP, UDP, IP, ethernet) should be taken into account.

Packet loss

Another parameter that influences the quality of the communication is the packet loss percentage. Loss may be caused by discarding packets in IP networks (network loss) or by dropping the packets at the terminal due to late arrival as they do not fit inside the current jitter buffer hence need to be discarded. Network loss is normally caused by large buffers, network congestion, route instability such route change and link failure. Congestion is the most common cause of loss.

3.3 VoIP monitoring requirements

Traditional network management identified five different management categories that can be expressed with the acronym *FCAPS*. FCAPS is the acronym of Fault, Configuration, Accounting, Performance, Security. Due to its importance FCAPS describes the different tasks that a *Network Management System(NMS)* should be able to perform. Even if FCAPS has a general meaning since it is not bound to specific service monitoring, FCAPS reasoning is still useful to analyze the monitoring requirements in a certain field, like VoIP monitoring.

3.3.1 Fault management

The goal of *fault management* is to recognize, isolate and then correct faults that occur in the network. Failures include hard failures, soft failures, misconfiguration, performance bottlenecks, loss of resilience and more.

VoIP infrastructure is quite different from the traditional PSTN infrastructure. This impacts on fault management for a variety of reasons.

First, VoIP user equipments are much more complex than traditional PSTN user equipments. Traditional telephony devices do not need to be configured whereas VoIP user agent needs to be configured using an appropriate method (web interface or telnet). The configuration task is no more allocated on a single entity and sometimes VoIP users need to manually adapt their VoIP configuration by themselves. The lack of a single configuration management domain makes misconfiguration more frequent.

Second, PSTN is composed by large islands owned and individually maintained by different organisations or service providers. As a consequence each island employs an homogeneous hardware and software infrastructure provided by the same vendor. This is not always true for VoIP networks where each user is allowed to choose its preferred VoIP device. Having an homogeneous infrastructure substantially reduces the interoperability issues.

A VoIP monitoring application should offer an easy to use and powerful interface to help network administrators to detect and resolve interoperability issues or misconfiguration. The most simple example of interoperability issue is the lack of a common codec to be used for a conversation. This issue can be caused by misconfiguration (e.g. some codec are explicitly disabled by user).

Thus, it is important to know what are the most widely used user

agents and correlate unsuccessful calls to user agents. Each user agent can be identified by the couple *(vendor, model, firmware)*.

3.3.2 Configuration management

Configuration management goal is to improve the robustness of the network, shielding the network resources from human mistake through automation. This work does not cover this aspect of VoIP management.

3.3.3 Account management

Account management goal is to gather usage statistics for users. The statistics are used for billing purpose or simply to better understand the service usage. In this case Account Management is replaced by Administration Management. Keeping track of users is useful for VoIP. Information like the number of registrations, the user agent and the different IP address used can be useful to detect faults or security attacks.

3.3.4 Performance management

Performance management offers a foundation for pro-active management of efficient network resource utilization, capacity planning and impact analysis. It can optimize the Return of Investment (ROI) of a network infrastructure by providing a deep insight into cost/performance tradeoffs at various levels of network resources. In case of VoIP the resources are the IP network and the VoIP entities. So the performance which has to be measured, can be grouped in network performance, that impacts on voice quality and signalling performance, that impacts on the service quality. The network performance

should be measured in order to understand if and when the network is capable to support the required number of concurrent audio or video calls. Signalling performance measurement involves the analysis of signalling protocol in order to discover performance bottlenecks in the VoIP infrastructure.

3.3.5 Security management

Security management has a central role in VoIP monitoring. The adoption of Internet for carrying both voice and signalling traffic offers new opportunities but also introduces security risks. First of all, on the internet the most widely used applications are usually the preferred victims of attackers. Due to its growing popularity, VoIP is going to become the next likely target. Second, even if VoIP reached a wide diffusion and the market of SIP devices is growing fast, vendors do not seems to pay enough attention to security. A VoIP phone can be a victim of attacks like any other internet host, with the difference that it has less resources and usually receives less security updates than soft-phones. As a consequence hard-phones are usually the perfect candidates to perform denial of service (DoS) attacks. SIP active fingerprinting tools, such as *smap*[19] were developed in order to discover the model and the firmware of the most widely used hard phones. Moreover very few devices support advanced security features (such as SRTP, SIPS or TLS). Thus, hacking the current generation of VoIP network is very easy. Third, the vulnerabilities of VoIP encompass not only the flaws inherent within the VoIP application itself, but also in the underlying operating systems, applications and protocols that VoIP depends on. The complexity of VoIP creates an high number of vulnerabilities that affect the three classic areas of information security:

confidentiality, integrity and authentication. Some of the most basic security attacks are presented in the following list:

- **session teardown**: an attacker could cause the forced termination of SIP sessions sending crafted BYE or CANCEL SIP messages to one of the call endpoints.

- **media hijacking**: Since RTP does not provide neither confidentiality or message integrity, RTP streams are susceptible to man in the middle attacks, such as RTP injection. An attacker injects prerecorded media streams into the outgoing connection. The injected data needs to be encoded with the same codec used by the eavesdropped data and must be placed slightly ahead of the eavesdropped data. This should always require the manipulation of some RTP header fields (SSRC, timestamp, sequence number) and UDP ports. In practice, some widely used RTP implementations do not evaluate those fields making the attack even easier[61].

- **registration hijacking**: an attacker impersonates a valid user agent and replaces the legitimate registration with its own address. As a consequence, all the incoming calls are forwarded toward the attacker user agent.

Detecting service usage deviations from the normal service usage pattern can help network administrator to discover security attacks. For example having subsequent short duration calls between the same endpoints is a very suspicious service usage pattern and can be caused by a teardrop attack or by poor audio quality. Further common service anomalies are listed in Table 3.3.

Anomaly detection requires the definition and the measurement of indicators used to distinguish anomalies from normal usage patterns.

Anomaly	**DoS**	**password cracking attempt**	**active fingerprinting**	**RTP injection**	**registration hijacking**	**teardrop attack**
Frequent successful registrations of an user	X					
Too many unauthorized registrations of an user		X				
Unknown SIP method	X		X			
SIP OPTIONS			X			
Malformed RTP packets	X			X		
Subsequent short duration calls between the same endpoints	X					X
Concurrent (or nearly concurrent) proxy registrations of an user from different IP addresses					X	

Table 3.3: Service anomalies and possible attacks

3.4 Why passive monitoring?

There are several approaches to network monitoring. The two common approaches are the passive and the active one. *Active monitoring* relies on the capability to inject test traffic into the network or send packets to servers and

applications. Automated VoIP agents are responsible to establish dummy sessions between fake users. Then the system response is analyzed to measure some parameters such as the end-to-end delay.

Passive monitoring relies on the capability to capture the traffic flowing across the network. The captured traffic is analyzed in real-time by network probes.

Active and passive monitoring can be considered as complementary and have their values and drawbacks. From the performance point of view both approaches require resources. Passive monitoring requires standalone equipments, the probes, that perform the analysis of the captured traffic. This means that the service itself is not penalized by the monitoring task. Instead active monitoring can waste resources (e.g. network bandwidth and processing power) that are otherwise needed to provide service to real users. Since active monitoring do not require real traffic it is useful to analyze the network performance when the service is not yet provided such as during the VoIP network assessment phase. Even if active monitoring is capable to offer meaningful results we should always remember that what is obtained through the synthetic traffic analysis is an estimate of the traffic perceived by users. On the other hand results coming from passive monitoring correspond to the quality perceived by the users, especially when the service usage is high. Moreover, service usage anomalies can only be discovered using a passive approach. This is the main reason why this work focuses on VoIP passive monitoring.

Furthermore, active monitoring is better suited for end-to-end measurements whereas with passive monitoring is possible to segment the network so that administrators can have a per network view of network performance metrics (such as packet loss and jitter).

Having this perspective is useful on large networks, since network bottlenecks or congested links can be easily identified. The view is especially useful when *voWiFi (VoIP over Wifi)* networks are used as VoIP access networks.

Through boundary monitoring, network operators can detect and act upon problems occurring both within and outside of their networks. Complaints received regarding poor quality calls can be checked against the monitoring system.

3.5 Challenges

This chapter has shown that monitoring VoIP services under a service oriented perspective allows to define the quality of experience of the VoIP service usage using objective measurements, called indicators. Several key performance indicators and key quality indicators, coming from the analysis of both the signalling and the media transport protocol, have been identified.

Furthermore the chapter has shown that service oriented network monitoring offers meaningful information for network managers, as some indicators are useful in several aspect of network management, including performance and security management.

The passive monitoring approach has been suggested as it allows to distinguish anomalies from normal usage patterns. However performing service oriented network monitoring in a passive way is not an easy task for a variety of reasons.

First, service oriented monitoring requires the measurements of different application level metrics, or indicators. Key quality and key performance indicators should be defined before service deployment since they represent

what need to be measured or computed by the service monitoring application. However this is not always true in practice. Usually new indicators may need to be introduced in order to provide a more accurate measure of service quality perception. Moreover some indicators may be defined in order to detect service usage or protocol anomalies. Thus service monitoring applications need to be flexible and extensible enough to measure indicators that were not taken into account during the design phase. New indicators should be introduced with little programming effort.

Second, measuring those indicators in a passive way usually requires the complete dissection and analysis of one or more application layer protocols. Adding application level support to network probes increase the complexity of monitoring software, since application level protocols are usually more complex and more often updated than lower layer protocols. Protocol complexity impact on monitoring requirements in terms of size of information set required and in terms of performance.

In a nutshell, service oriented network monitoring is an enrichment of traditional network monitoring. Service oriented network monitoring applications require additional design effort in order to manage the complexity and achieve high performance at the same time.

Chapter 4

RTC-Mon framework

This chapter presents RTC-Mon: a framework designed in order to reduce the time needed to implement complex service monitoring applications such as VoIP monitoring applications.

4.1 Design goals

We identified the following design goals in order to design and implement a framework capable to provide a ready to use infrastructure to speed up the development of complex VoIP monitoring applications.

4.1.1 Functional design goals

The design has been done in order to provide all the information that a passive VoIP monitoring application may need to have in order to provide a comprehensive view of the VoIP service status. This means that the framework must be capable to analyze both the signaling and the media protocols in order to

gather the information identified in Chapter 3.

4.1.2 Performance

The framework should be able to handle several hundred calls on a desktop class computer and should overcome some of the performance issue related to VoIP traffic filtering. In fact the adoption of dynamic ports makes current packet filtering technologies ineffective. VoIP traffic filtering using the standard BPF filtering mechanism gives to the developers two different choices:

1. use a single flat *"udp"* filter
2. use a single *"udp and port 5060"* to have the signaling traffic and a new filter for every RTP stream

The first solution practically makes the filtering ineffective and thus result in wasting system resource. In fact, depending on the traffic behaviour a large number of non VoIP packets can be forwarded to the user-space application and then later on discarded.

The second solution, which seems to be the best one, is not suitable to handle a great amount of different RTP streams and it is practically unusable with just few hundreds of VoIP calls. In fact BPF filtering is done at the socket layer and each socket can accommodate only one BPF filter, so that multiple BPF filters will require multiple sockets[1]. This strategy is not scalable and not efficient, since each packet has to be parsed by every socket.

So it is clear that to improve the performance of BPF we need to have a filtering mechanism that must be:

[1] libpcap uses the PF_PACKET socket family to perform the packet capture

- *effective*: only the required VoIP traffic should be forwarded to the user-space
- *scalable*: the system load due to filtering must not grow up linearly with the number of filters.

4.1.3 Usability design goals

The framework should be easy to use, flexible and extensible. To be easy to use, it should hide the complexity of traffic capture, protocol parsing and thread management.

Flexibility means that the framework should allow users to have access to application level information, but also to raw packets.

In order to allow users to add support for different kind of protocols the architecture should be easy to extend and should be modular.

4.2 Rationale

Before starting the design an analysis of the VoIP traffic patterns and protocols has been done in order to optimize the most costly task while preserving usability. The key to improve the performance of the framework is to provide an effective filtering mechanism capable to discard non VoIP traffic at the kernel level. This can be accomplished exploiting the same approach adopted by *mmap*, but improved in order to overcome the BPF limitations. Thus, the signalling traffic needs to be analyzed in order to dynamically manage a set of filters.

Moreover in order to improve the performance an analysis of VoIP traffic

pattern has been carried on. The Table 4.1 shows the traffic produced by two VoIP calls using two different audio codecs. The duration of each call is 3 minutes, which is lower than the average call duration time reported by several telecom providers. Since the VoIP traffic is almost composed by RTP packets, the RTP analysis is one of the most resource consuming task; thus optimizing it is the second key to improve the performance.

The RTP analysis does not require the RTP payload inspection. Jitter, packet loss and out of order packets can be computed using only the RTP header which is simple to parse. Moreover we can assume that most of the monitoring applications do not need to inspect the RTP payload. Furthermore we can reasonably assume that stream analysis results are not often needed: most of the VoIP monitoring applications need the results only when the call ends. In the worse case, the analysis results are needed every few seconds. For that reasons, doing the RTP analysis inside the kernel is feasible and can offer benefits in terms of performance, if RTP packets are not forwarded in user space.

The signalling traffic is carried over well known ports and it is easy to parse. Even if the signalling traffic is only a fraction over the total, performing the SIP filtering at the kernel level using SIP fields can be a benefit in terms of performance.

4.3 Framework overview and design choices

Those considerations lead to the design of a mixed user-space kernel-space architecture depicted in Figure 4.10. Having a mixed kernel userspace approach is the best compromise between performance and flexibility. Performing the

GSM	pkts	bytes	mean pkt size	bandwidth percentile
RTP	19410	1690825	87	99.52%
SIP	22	8161	370	0.48%
G.711	pkts	bytes	mean pkt size	bandwidth percentile
RTP	26515	4262450	160	99.64%
SIP	20	15187	759	0.36%

Table 4.1: Typical VoIP traffic pattern

RTP analysis and SIP filtering at kernel level results in a reduction of the overall system load due to memory copies and system calls. The SIP filter can be configured to forward only packets matching content based filters. Thus, it is possible to filter calls using *From*, *To* and some other SIP fields.

The RTP analyzer performs the RTP analysis without forwarding packets in user space to reduce the number of packet copies. However, if the userspace application needs to inspect the RTP payload, the RTP analyzer can be configured in order to forward each analyzed packet to userspace.

An user space library can benefit from these primitives and it is the most convenient way to keep the state of calls and active users. The library, called *LibVoIP*, analyze the SIP filtered traffic in order to instrument the RTP analyzer at kernel level. LibVoIP is an event based library. This paradigm is commonly used by several network libraries, such as [58]. The paradigm has been chosen for a variety of reasons. It allows developers to concentrate on the implementation of event handler. Moreover it do not require complex message exchanges that can cause performance impairments.

The library is implemented in C++, which is, unlike C, object oriented. C++ libraries, such as *Standard Template Library(STL)* and *Boost*, allows the development productivity to be increased. We did not considered interpreted

languages with automatic memory management such as Java, C# and Python, since they usually offer lower performance and requires more resources. Moreover we preferred C++ since programs implemented with this language are very portable, even if at the source code level; C++ software can be compiled on almost every platform and architecture, including small embedded boxes, where Java Runtime Environment and .Net Framework runtime are usually unavailable.

The RTC-Mon *libpfring* library represents an intermediate layer between LibVoIP and the kernel infrastructure. The layer provides a packet oriented capture library and allows the kernel to be instrumented. Having this layer important because it allows the kernel infrastructure to be reused in different contexts.

The framework architecture was designed with VoIP monitoring in mind, but it can be adapted or extended in order to perform other monitoring tasks. Since the extensibility was one of the design goals the framework can be enhanced and extended at different layers. The framework is capable to perform VoIP analysis, but it provides a set of features that can be reasonably exploited in order to allow the monitoring of several real-time services.

In particular the kernel part of the architecture has been designed in order to provide a deep packet inspection infrastructure rather than a custom solution that covers VoIP monitoring needs. Thus, the SIP filter and the RTP analyzer are just examples of the infrastructure usage. A detailed description of the infrastructure is given in 4.4.

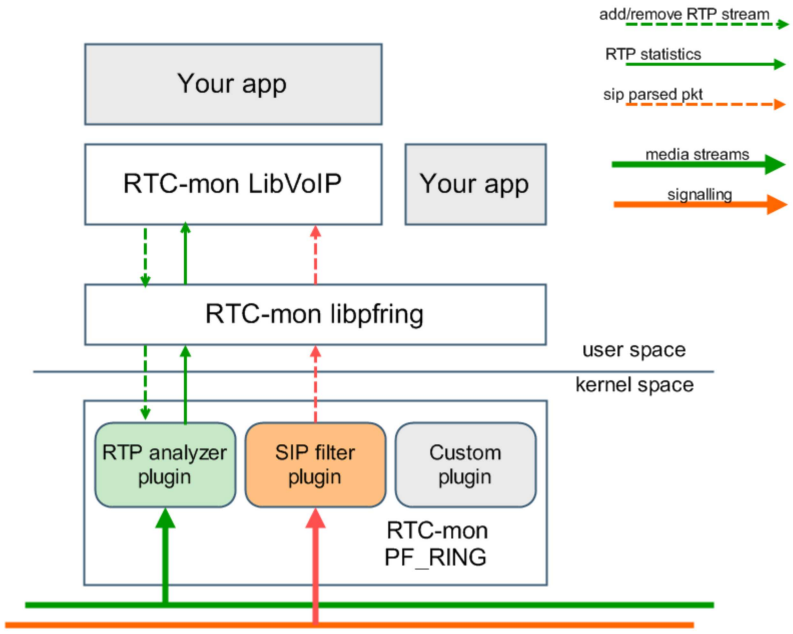


Figure 4.1: RTC-Mon overview

4.4 Kernel enhancements

In order to increase the VoIP capture performance the kernel should provide the following features:

- *provide SIP filtering at kernel level*: this means that the kernel should be able to parse some of the SIP header field in order to let users select which SIP packets should be forwarded to user space.

- *be capable to analyze a set of RTP streams*: RTP streams analysis involves the ability to compute jitter, packet loss and out of order packets over an RTP stream. The kernel should provide some primitives used to manage the set of streams to be monitored. Moreover, it should be able to know if a captured packet belong to one of the streams in the mon-

itoring set. Packets belonging to an RTP stream are not forwarded to userspace. A polling mechanism in needed in order to read RTP analysis result.

In order to implement the previously described features we designed an extensible infrastructure that is not bound to a specific analysis domain. The infrastructure provides a set of capabilities that can be exploited by developers in order to implement the filtering and the analysis of disparate application layer protocols. A specific application layer protocol support can be introduced by means of *plugins* taking advantages of infrastructure primitives. Each plugin is a kernel module providing the implementation of a common interface.

These are the primitives offered by the kernel infrastructure:

1. *a mechanism used to forward both packets and parsing information to userspace*: one of the tasks that can be performed with plugins is the filtering of packets using application layer protocol information. To perform the filtering, plugins must perform the parsing. The parsing information collected by plugins can be forwarded to the userspace instead of being discarded. This improves the performance and reduce the parsing effort at the user space level.

2. *IP defragmentation*: passive monitoring systems usually perform IP defragmentation in user space. However, IP fragments should be reassembled in order to perform application layer protocol analysis in kernel space.

3. *packet header parsing up to transport layer*: to perform the parsing of an application layer protocol is necessary to know the packet payload

offset which indicate when the payload begins. The payload offset is not constant due to the usage of variable length headers and it is known only after packet parsing up to transport layer.

4. *a polling mechanism*: userspace applications should be able to poll the plugins in order to have information regarding the analyzed traffic. The polling mechanism is especially needed for plugins that perform the analysis without forwarding packets in userspace. In that case, the polling mechanism is the only interaction between the userspace and the kernel space.

5. *a connection tracker with persistent memory storage*: The analysis of some protocols can be performed just over each packet. However in some case, such as RTP, is necessary to perform some computation over flows more than over packets. Some information, such as packet loss, must be kept in a persistent memory that is freed only when the connection tracking of a particular stream is disabled.

6. *extensibility by means of plugins*: Plugins can take advantage from the previously listed primitives in order to perform upper layer protocol analysis. A detailed description of plugin interface is given in section 4.4.2.

7. *a mechanism used to associate a plugin to a captured packet*: plugins are enabled on packets matching rules. The rule mechanism is described in section 4.4.1.

In order to implement the infrastructure we decided to enhance PF_RING rather than writing something from scratch for a variety of reasons. First of all PF_RING already provides some of the needed features. Packets captured

width PF_RING are enriched with a memory area containing transport layer parsing information. The mechanism was easily customised in order to carry also application level parsing information (capability 1). IP defragmentation was previously implemented by the author (capability 2). PF_RING parses packets up to transport layer (capability 3). PF_RING is a special socket and supports the *getsockopt()* and *setsockopt()* system calls. Those system calls are suited to implement a polling mechanism (capability 4).

Secondly PF_RING is a mature project with a large community of users. Thus it has been deeply tested. It already provides an user space library to enable faster development of capture applications.

4.4.1 Plugins 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[2]. 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,

[2]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.

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.

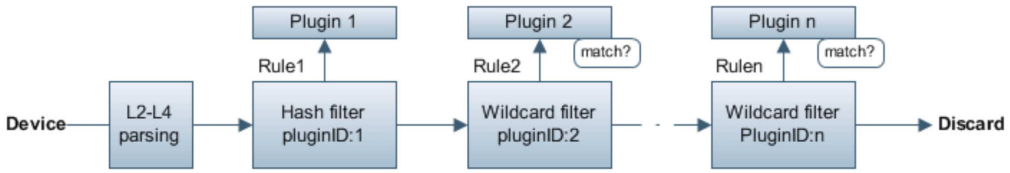


Figure 4.2: Extended PF_RING overview with plugin architecture.

Figure 4.2 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 rule mechanism requires a closer look. As shown in Figure 4.2, rules can be of two types: hash or wildcard. Hash rules 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 media streams.

Wildcard rules, on the other hand, are more flexible, allowing to match, for example, all UDP packets going to a specific port. These rules can also specify a higher-layer, plugin-specific filter. In this way, a user could instru-

ment the system to process only INVITE messages.

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 4.3). 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 chapter.

There are two different memory areas that plugins are allowed to manage: the parse memory buffer and the hash memory. The *parse memory buffer*, depicted in Figure 4.4 contains pointers to plugin parse buffers. A parse buffer is a contiguous memory area allocated by plugins whenever a packet needs to be processed and deallocated by the PF_RING core when the packet has

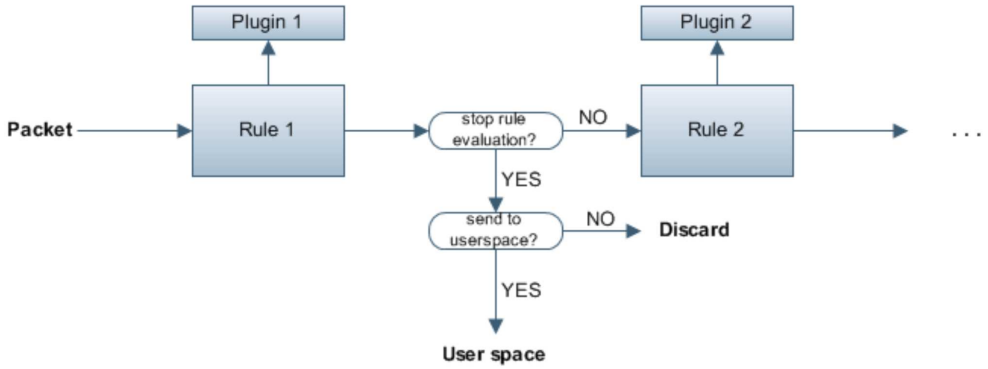


Figure 4.3: Packet paths for all possible rule action types.

been handled. Since plugins have been introduced to allow the analysis and the parsing of different application layer protocols the plugin architecture does not make assumptions regarding the parse buffer's content. Plugin developers can use the parse buffer to allocate plugin dependent parsing information. The role of the parse buffer is crucial for two reasons. First, the parsing information can be used to perform packet filtering using application layer protocols. Second, the parse buffer of the plugin associated to the latest matching rule is copied to userspace together with the packet if the action 2 is specified.

Figure 4.5 shows the memory layout of a PF_RING slot containing a successfully parsed packet that needs to be forwarded to userspace. The slot is composed by three different memory regions. The first region contains packet parsing information up to layer 4 and includes the length of the captured packets and the length of the second memory region. It is worthwhile to note that the PF_RING core is responsible to perform packet parsing up to layer 4 so that each packet is parsed up to transport protocol exactly once regardless of the number of sockets and plugins enabled. This parsing information is

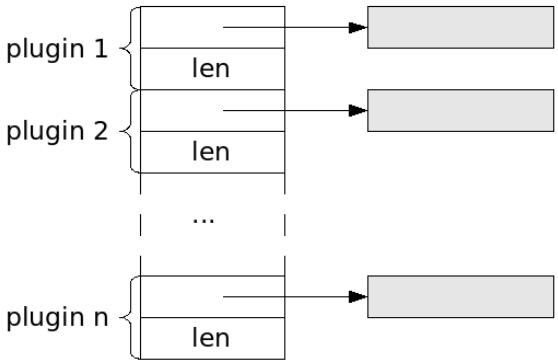


Figure 4.4: The parse memory buffer

forwarded to plugins so that plugins developers do not have to take care of lower layer protocol header dissection.

The second region contains parsing information coming from higher layer protocol parsing. That's where plugins come into place. This region contains the parse buffer of the latest plugin having a match. The third and last region contains the captured packet.

The second memory area that plugins are allowed to manage is the *hash memory*. The hash memory is a traditional hash composed by singly linked bucket chains. Each bucket represents a single hash rule and contains:

- the pointer to the next bucket
- a void pointer to a plugin dependent memory area
- the length of the memory pointed by the previous pointer
- the hash filtering rule

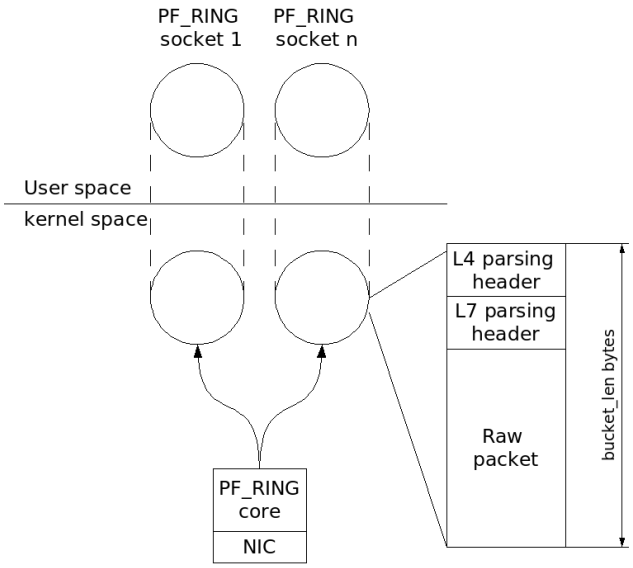


Figure 4.5: PF_RING slot layout with plugin parsing information.

The main difference between the parse memory buffer and the hash memory is that the memory allocated by the hash is not freed until the hash filtering rule is deleted so that plugins are allowed to keep flow based information such as the number of packets of a TCP connection. Another difference is the mechanism adopted to grant to user space applications the access to that memory. Parse buffers can be copied to userspace together with the packet. On the other hand the plugin dependent data pointed by each hash bucket can be read using a polling mechanism. The RTP analyzer, implemented as a PF_RING plugin makes use of the hash memory to store RTP analysis information read by user space applications using the polling mechanism.

So far we said that the RTC-Mon PF_RING core takes care of many tasks including IP defragmentation, packet parsing up to layer 4, rule and plugin management including parse buffer and hash bucket deallocation. On

the other hand plugins can introduce the support for upper layer protocol parsing and analysis. Filtering packets using fields from application layer protocol is possible and will be described in details in the Section 4.4.2.

Plugins are allowed to allocate their own parse buffer that can eventually be forwarded to the userspace together with the packet. Moreover they are allowed to manage hash buckets corresponding to hash rules. The hash bucket content can be copied on request to userspace using the polling mechanism.

4.4.2 Plugins developement

Plugins are kernel modules providing two entry points *module_init* and *module_exit* that are called when the plugin is inserted and removed. The module_init function is responsible for registering the plugin using a PF_RING function that takes as parameter a struct containing:

- *a plugin identifier*: an integer used to univocally identify the plugin
- *plugin_filter_skb*: a pointer to a function called when a packet needs to be filtered
- *plugin_handle_skb*: a pointer to a function called whenever a packet is received
- *plugin_get_stats*: a pointer to a function called whenever an user wants to read statistics from a filtering rule that has set this plugin as a filtering action

Plugin developers can provide the implementation of one or more of the previously mentioned functions. The following sections describe the defined pointer to functions in detail.

plugin_handle_skb

```
typedef int (*plugin_handle_skb)(
    filtering_rule_element *rule,       /* In case the match is on the list */
    filtering_hash_bucket *hash_bucket, /* In case the match is on the hash */
    struct pfring_pkthdr *hdr,
    struct sk_buff *skb,
    u_int16_t filter_plugin_id,
    struct parse_buffer *filter_rule_memory_storage);
```

plugin_filter_skb

```
/* Return 1/0 in case of match/no match for the given skb */
typedef int (*plugin_filter_skb)(
    filtering_rule_element *rule,
    struct pfring_pkthdr *hdr,
    struct sk_buff *skb,
    struct parse_buffer **filter_rule_memory_storage);
```

plugin_get_stats

```
/* Get stats about the rule */
typedef int (*plugin_get_stats)(
    filtering_rule_element *rule,
    filtering_hash_bucket *hash_bucket,
    u_char* stats_buffer,
    u_int stats_buffer_len);
```

4.4.3 Implemented Plugins

SIP plugin

The SIP filter depicted in Figure 4.10 was implemented as a PF_RING plugin called *sip_plugin*. The plugin is meant to be enabled using a wildcard rule that matches all UDP packets with the source or destination port set to the standard SIP port (5060 see Appendix A).

The main purpose of the SIP filter is to discard unnecessary signalling traffic at the kernel level. The following filtering structure is defined for that purpose.

```
struct sip_filter {
  u_char swap_peers;
  /* 1 = match also when swapping caller with called
     Used only for filtering and not packet
     parsing */
  sip_method method;
  u_char caller[PEER_LEN]; /* Empty string means 'any' */
  u_char called[PEER_LEN]; /* Empty string means 'any' */
  u_char call_id[CALL_ID_LEN]; /* Empty string means 'any' */
};
```

Although the parsing structure is simple it is powerful enough to select only packets coming from a specific user user or having the specified SIP method. For example it is possible to select only SIP INVITE packets coming from user *fusco@sip.com* or every REGISTER packet regardless of the user.

In order to perform the filtering using the previously described structure the plugin has to parse SIP packets. The following C structure is the parsing

structure defined by the plugin. The structure is copied to user space together with the packet when the action *forward* is selected.

```
struct sip_parse {
  sip_method method;
  u_char caller[PEER_LEN];
  u_char caller_name[PEER_LEN];
  u_char called[PEER_LEN];
  u_char called_name[PEER_LEN];
  u_char call_id[CALL_ID_LEN];
  u_int16_t cseq, status_code;
  sip_method cseq_method;
  /* Offsets with respect to the SIP payload */
  u_int16_t user_agent_offset,
    sdp_offset, contact_offset,
    record_route_offset;

};
```

The majority of field names are self-explanatory. Some fields such as *caller* and *called* are completely parsed (a char array is given). For other fields, such as the *user agent*, it is given only an offset. Moreover, only a small subset of SIP fields is parsed, thus the plugin is only responsible to perform the first stage parsing. The parsing of SIP packets needs to be completed by an userspace analysis library. Parsing very few SIP fields has been a design choice. We decided not to implement an heavy parsing at the kernel level for many reasons. First, we wanted to keep the parsing information as small as possible.

The parsing structure has to be copied from userspace to kernel space together with the packet; having a very large parsing struct requires bigger PF_RING slots. Second, the SIP component may need to be reusable by applications having different requirements. Some applications just may need to have the basic fields (such as `From` and `To`), thus parsing more fields will be useless. Third, having a bug in kernel can cause a system hang or crash.

The sip_plugin also provides an implementation of the plugin_get_stats function so that the plugin can be used for monitoring purposes without copying SIP packets in user space. Using the polling mechanism the following C structure can be read from user space in order to have statistics regarding the analyzed traffic.

```
struct sip_stats {
  u_int32_t num_register_pkts, num_options_pkts,
    num_invite_pkts, num_ack_pkts,
    num_notify_pkts, num_bye_pkts,
    num_cancel_pkts, num_sip_pkts,
    num_publish_pkts, num_subscribe_pkts,
    num_unknown_pkts;
};
```

The structure contains a counter for each SIP packet type, including malformed packets. Providing these counters can help network administrator to discover protocol usage anomalies. For example having a large number of malformed packets can be a symptom of a denial of service attack.

RTP plugin

The RTP analyzer component was implemented as a plugin called *rtp_plugin*. The plugin performs the RTP analysis of a managed set of RTP streams inside the kernel and has to be enabled using hash rules, each representing a monitored RTP flow. The rtp_plugin performs the RTP analysis using fields from the RTP header so that applications can use the polling mechanism to read RTP analysis results without copying RTP packets to userspace.

For each monitored RTP stream the RTP analyzer keeps the following information:

- *payload type*(8 bits): defines the format of the RTP payload and determines its interpretation by the application
- *total packets*(32 bits): this is the number of captured packets belonging to the monitored stream
- *total bytes*(32 bits): this quantity represents the total bandwidth consumed by the RTP stream.
- *number of malformed packets*: an RTP packet is considered malformed if does not carry any payload
- *number of out of order packets*: the expected sequence number for a RTP packet is the sequence number of the previous captured packet plus one. If the sequence number for a captured packet is different from the expected sequence number the packet is considered out of order. The RTP plugin keeps track of the number of malformed packets, so that RTP injection attacks can be easily discovered.

- *current, mean and max jitter*: three different values for the jitter are kept. The maximum, the mean and the actual jitter value.
- *initial sequence number*: corresponds to the lower sequence number observed
- *last sequence number*: the field represents the highest sequence number seen.
- *initial timestamp*: corresponds to the RTP timestamp of the first captured packet belonging to the monitored RTP stream
- *last timestamp*: is the RTP timestamp of the latest captured RTP packet
- *SSRC*: this is the synchronization source identifier for the monitored RTP stream

With this information is quite easy to compute some of the most important key performance indicators. The packet loss can be easily computed using the initial and the last sequence number. The bandwidth consumed by the monitored RTP stream is kept and can be used together with the number of packets to compute the mean packet size.

4.5 LibVoIP

LibVoIP is a C++ VoIP analysis library that exploits the features provided by the kernel infrastructure in order to allow fast developments of complex VoIP monitoring applications. The library is already capable to perform active calls and users monitoring and provide a large set of information regarding the analyzed traffic that cover much of the monitoring requirements. However one of

the most important library's design goal is to provide an extensible framework designed to manage the complexity of VoIP traffic analysis rather than a fixed and static solution. Thus, the library automates many of the standard tasks, such as protocol parsing, leaving the developers free to concentrate on writing more complex library extensions.

4.5.1 Overview

Every passive probe should perform the following activities: packet capturing, packet parsing and protocol analysis. LibVoIP is an event based library that covers all of these activities and provides an easy to use interface to export analysis results to library users. Figure 4.6 provides an overview of the library and shows the most important LibVoIP components.

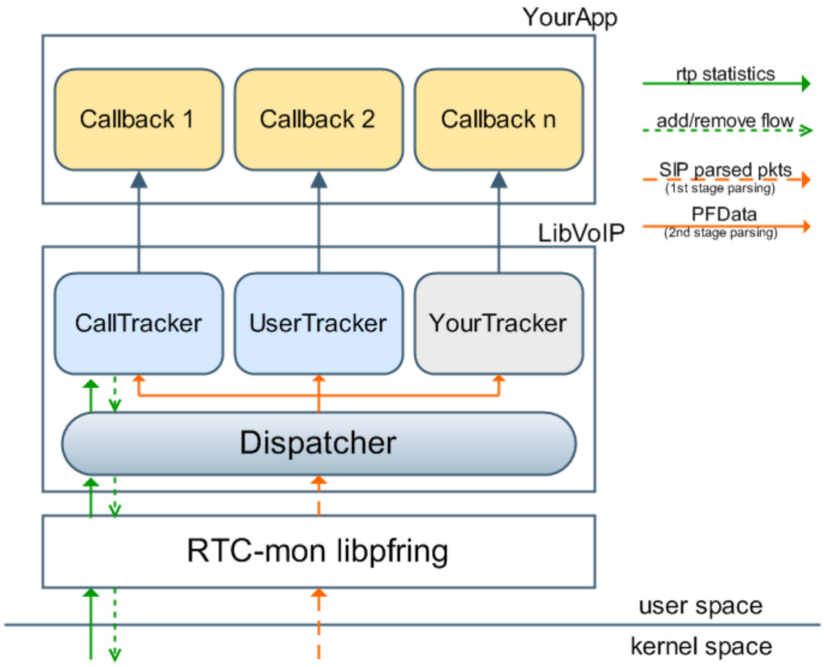


Figure 4.6: LibVoIP overview

Packet capturing and parsing is performed by the *Dispatcher* component. The class allows to start the capture process from a specified network interface. The Dispatcher's `start` method creates a configurable number of capture threads. Each thread gets a captured packet using the underlying *libpfring*, completes the parsing of signalling packets and then dispatches the packet together with the parsing information to a set of *Tracker* components, described later. Packets, with the corresponding parsing structures are encapsulated in *PFData* class instances. *PFData* class is substantially a packet container with the additional methods `parse` and `get_parsed_data`. The `parse` method is called by capture threads to perform the second stage SIP parsing.

The parsing structures are substantially three:

- the layer 4 parsing structure coming from the PF_RING core
- the SIP parsing structure from the SIP plugin
- another SIP parsing structure which is filled by the PFData

The capture process can be stopped using the Dispatcher's `stop` method. The *Dispatcher* is the interface between trackers and the underlying libpfring library, so it provides some methods to manage the RTP analyzer's monitoring set.

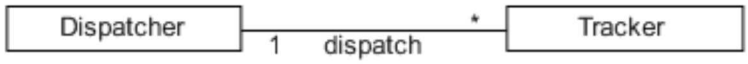


Figure 4.7: Trackers and dispatcher relationship

Trackers, such as the *CallTracker* and the *UserTracker* are subclasses of the *Tracker* class (see Figure 4.8). They are responsible to perform the

signalling analysis and to keep the state of something which is relevant for monitoring purposes. Tracker are organized as hash data structures composed by singly linked chains of *Bucket* subclasses instances. Each tracker provides its own *Bucket* subclass. Bucket subclasses are used by trackers to keep information regarding the analyzed traffic; as they are used by event handlers they must provide accessors methods.

The library comes with two different trackers, but the library design allows and encourages the development of custom tracker components. In Section 4.5.2 the extensibility through tracker is better described whereas Section 4.5.3 and 4.5.4 describe the standard trackers provided by the library.

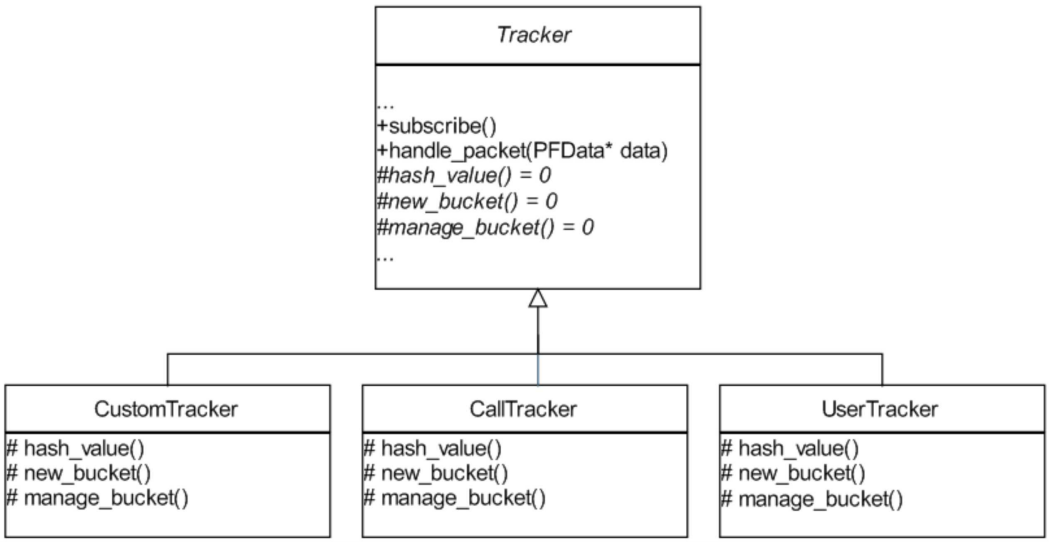


Figure 4.8: Trackers

Trackers emit events when the state they keep changes (e.g. a new user has been discovered). The library defines three different event types: `NEW_BKT_EVT`, `UPD_BKT_EVT` and `DEL_BKT_EVT`.

Events are handled by event handlers provided by library users. Each

tracker is responsible to execute the set of event handlers associated with the generated event type. Event handlers are implemented as *Callback* subclasses (see Figure 4.9). Callback subclasses must provide an implementation of the `execute` method. The method is executed by trackers and takes as argument: a reference to a Bucket subclass instance and a reference to the PFData instance that caused the event generation.

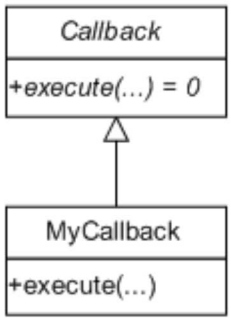


Figure 4.9: Callback

The main function of a simple monitoring application written on top of LibVoIP usually consists of just few lines of code. The main has to perform to following activities:

1. creation of a new Dispatcher object
2. creation of a new Tracker subclass object
3. creation of a new Callback subclass object
4. subscription of Callback to the tracker (using the Tracker's `subscribe` method
5. registration of the tracker (using the Dispatcher's `add_tracker` method)

6. activation of the capture process (using the Dispatcher's `start` method)

Using the provided trackers it is possible to build simple VoIP monitoring applications with very little efforts. For example, a console applications that prints on the screen every discovered user is less than 30 lines of codes. A more complex application that shows every successfully completed calls together with the RTP statistics is just few lines more. In both cases, a single callback is sufficient and its execute method is less than five lines of code.

4.5.2 Writing custom trackers

Trackers are used to enhance the analysis capabilities of LibVoIP. Thanks to the modular approach developers can focus on their specific monitoring needs without wasting time in protocol parsing and so on. What tracker developers must provide is the implementation of methods responsible to:

- *compute an hash value over a captured packet*: an hash is used to find existing buckets which may be updated after the packet analysis. Trackers are allowed to provide one ore more hash functions to be used to find the relevant buckets.

- *create a new bucket*: trackers are supposed to keep the state of something relevant for monitoring purposes. Thus they should provide a method used to allocate their own data structures.

- *manage an existing bucket*: once a packet is captured it is dispatched together with parsing information to the trackers for being handled. Trackers provide their own method in order to perform the analysis and are allowed to emit events.

However this is only the minimum aid that the library provides to developers writing custom trackers. Beside parsing, capturing and event handling the library also provides a purging mechanism to delete expired buckets from the system. A thread called cleaner thread is responsible to visit all the Tracker records in order to find (and purge) expired buckets. Developers can override the Bucket's `expired` method in order to customise the default purging behaviour, which is timeout based. This mechanism can be used to provide a more complex and smarter purging. For example the purging mechanism has been customised during the implementation of the call tracking feature.

4.5.3 Call tracking

The CallTracker is responsible to provide information regarding active calls. In case of successfully established calls the information includes the network performance metrics for each media flow discovered by the signalling analysis. CallTracker employs the RTP analyzer in order to compute the performance metrics and it is responsible to manage the RTP stream set to be analyzed by the kernel infrastructure.

The signaling analysis involves the management of *CallTrackerBucket* instances keeping the state of each VoIP call. In particular a new CallTrackerBucket instance is created whenever a new INVITE message is detected and purged whenever the VoIP session is closed by means of a *BYE* or *CANCEL* message.

Since BYE and CANCEL messages can be lost, the CallTrackerBucket expired method has been overridden in order to perform a more sophisticated purging. In particular, in case of a successfully established call the cleaner

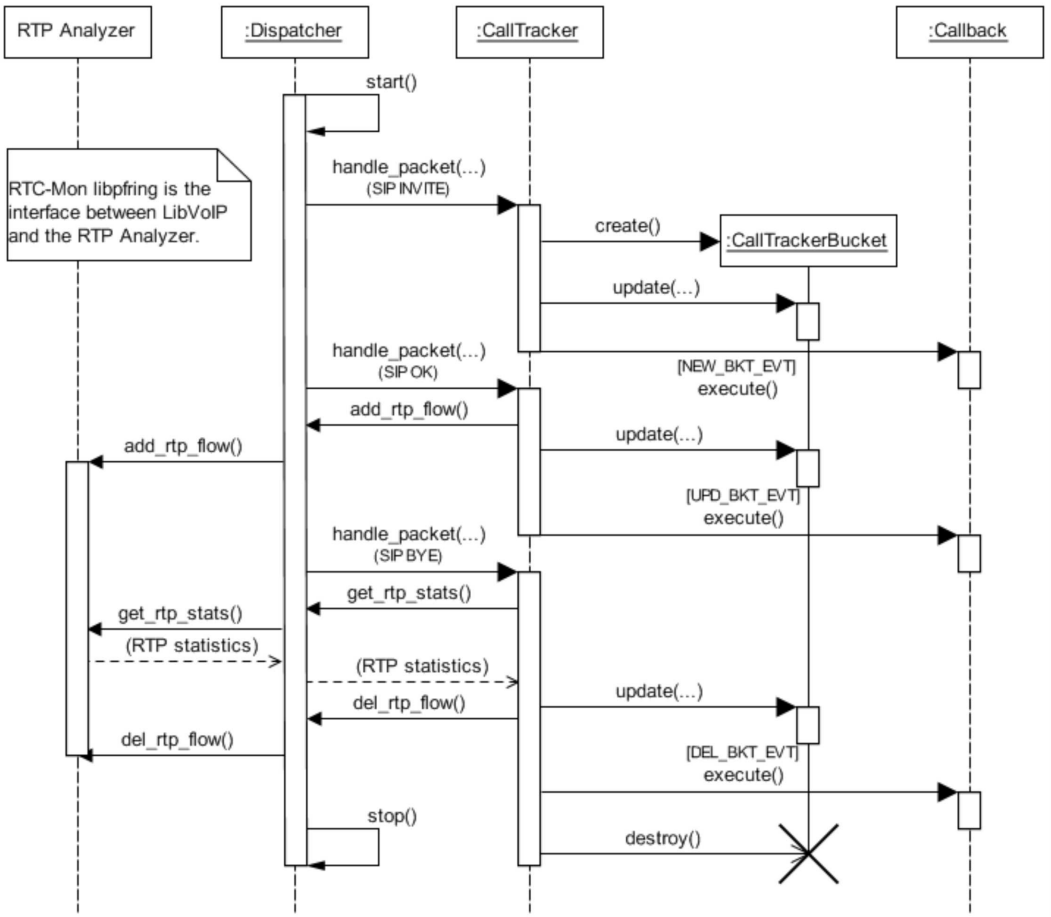


Figure 4.10: Interactions between Dispatcher, RTP analyzer and CallTracker

thread updates the statistics of each monitored RTP flows. If no packets are seen during a configurable time period, the bucket can reasonably considered expired, thus purged, even if the BYE message has not yet been received.

4.5.4 User tracking

Active calls discovering is performed by the CallTracker component whereas the UserTracker component is responsible to discover active SIP users. Keep-

ing track of users can help network managers to perform a proactive management of their networks and to discover misconfigurations. If the number of SIP users is going to increase, then network managers should expect to see more SIP calls on their networks. Moreover having a detailed view of SIP users can be useful to network managers in order to understand service usage anomalies which can be caused by denial of service attacks or simply misconfiguration.

The UserTracker component was introduced in order to discover active SIP users. The UserTracker is capable to keep the SIP registration attempts and the time of the latest successful registration for each user. In this way network manager can discover SIP phone misconfiguration (e.g. a wrong password has been inserted by an user), troubles with some SIP registrar servers and password cracking attempts.

For each discovered user the UserTracker keeps and updates in real time a set of counters representing the number of sent SIP packets for each packet type. Using those counters fraudulent users can be easily discovered.

Moreover, the UserTracker allows to keep track of users mobility since for each user it is kept the contact's IP address. Last but not least, the component detects the User Agent (soft phone or hard phone) for each discovered user.

Chapter 5

RTC-Mon validation

The previous chapter introduced the RTC-Mon framework.

This chapter will present VoIPMon, which is a sample application implemented on top of RTC-Mon and will describe some of the possible use cases for the framework. Then, the framework performance results will be presented. At the end the thesis requirements will be validated.

5.1 VoIPMon: RTC-Mon at work

The framework has been designed in order to enable faster development of comprehensive VoIP monitoring applications. Thus a complete VoIP monitoring application has been implemented on top of it. The structure of the application is depicted in Figure 5.1. The solution is composed by different components. *VoIPStorer* is a C++ application that uses RTC-Mon in order to analyse the VoIP traffic. Analysis results are then stored in a MySQL database. Time varying data, such as the maximum number of concurrent calls, is handled in a different way. In fact the *RRDUpdater* component is

responsible to store the time varying data and to produce graphs regarding a selected time frame.

Since much of the complexity is encapsulated into the LibVoIP library the software is quite simple and yet capable to provide a large set of information regarding the analysed VoIP traffic.

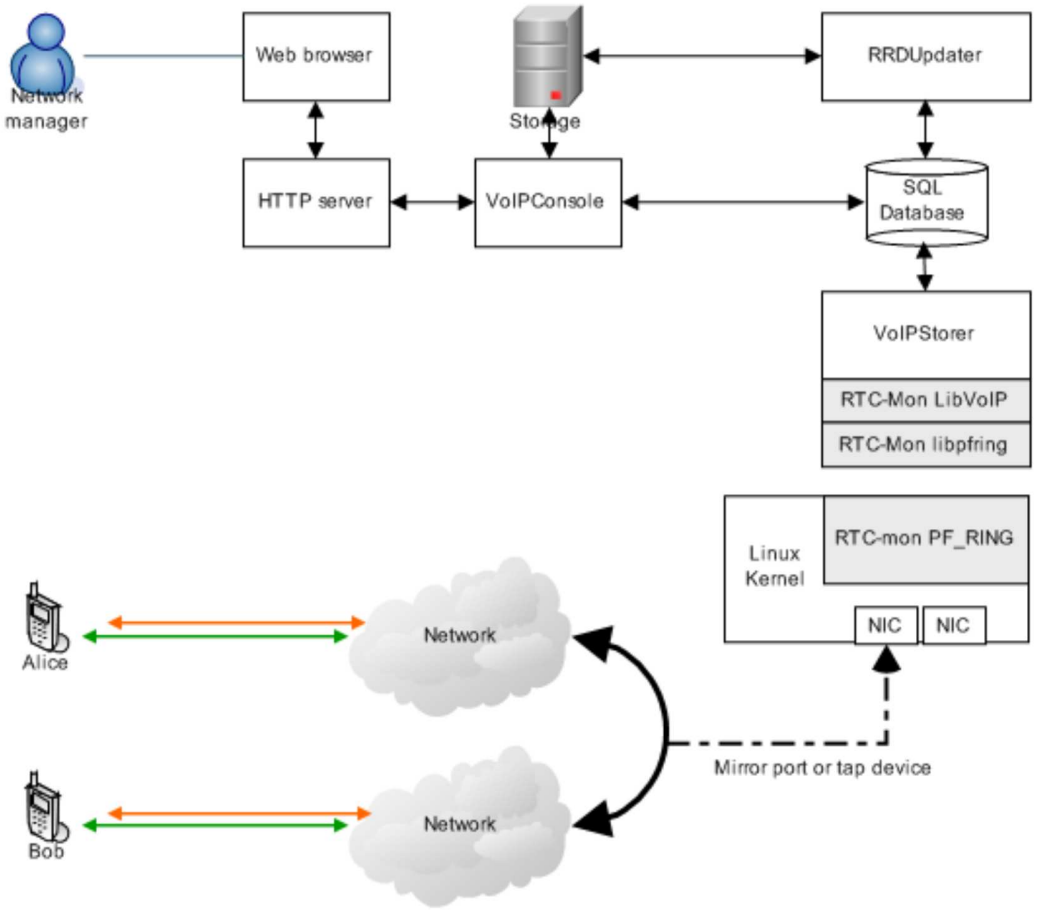


Figure 5.1: VoIP-Mon: an RTC-Mon based VoIP monitoring application

5.1.1 VoIPStorer

VoIPStorer is a C++ application written on top of the framework in order to perform the VoIP traffic analysis. The analysis results are then stored in a MySQL database. Since the packet capture and dissection and protocol analysis is performed by LibVOIP the software is simple and it is no more than 800 line of codes. The software make use of both the CallTracker and UserTracker components in order to perform a comprehensive VoIP analysis. In order to store the analysis results in the MySQL database it define two different callbacks responsible to store user and calls information respectively.

5.1.2 RRDUpdater

The RRDUpdater component is responsible to store time varying information and to generate some graphical reports (png) images that are shown by the VoIPConsole. RRD stands for Round Robin Database, which is a commonly used tool[42] to store time series data and to quickly generate graphical representations of the data values collected over a definable time period.

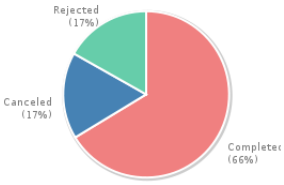
The RRDUpdater component uses the information stored in the SQL database in order to compute some metrics such as:

- *the Average Call Duration*
- *the Maximum Number of Concurrent Calls*
- *the Bandwidth utilisation*

Those metrics are computed every five minutes and then inserted into Round Robin Databases.

5.1.3 VoIPConsole

The VoIPConsole is a web application, written in php, that shows information regarding calls and VoIP users. Figure 5.2 shows a screenshot of the VoIPConsole calls page. The page shows call attempts and completed calls discovered in a selected time-frame. For each call, the caller and the called are shown. Moreover, for successfully established and completed calls the Setup Time and the Duration are reported.



Call Status	Total	Percentage
Completed	4	67%
Canceled	1	17%
Rejected	1	17%

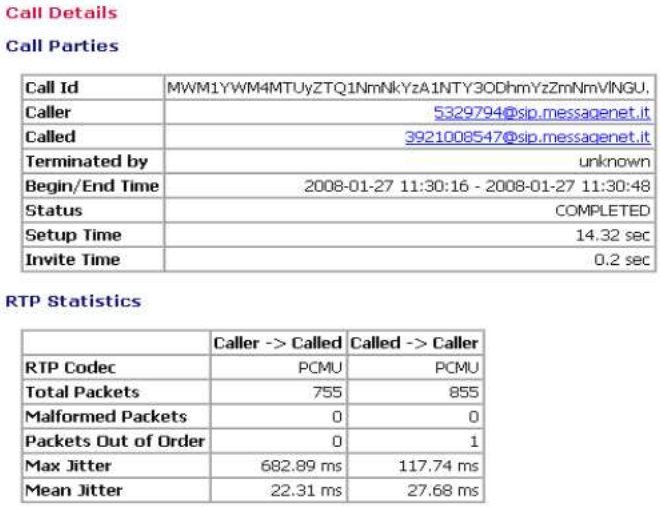
Call Details

Caller	Called	Call Start	Call Status	Setup Time	Invite Time	Duration	Details
5324056@sip.messagenet.it	5324056@sip.messagenet.it	2008-01-26 19:46:06	COMPLETED	1.3 sec	0.2 sec	2 sec	Info
5324056@sip.messagenet.it	5324056@sip.messagenet.it	2008-01-26 19:46:22	COMPLETED	1.7 sec	0.3 sec	3 sec	Info
5324056@sip.messagenet.it	5324056@sip.messagenet.it	2008-01-26 19:46:41	CANCELLED				Info
5329794@sip.messagenet.it	3921008547@sip.messagenet.it	2008-01-27 11:30:16	COMPLETED	14.3 sec	0.2 sec	18 sec	Info
5329794@sip.messagenet.it	3288169497@sip.messagenet.it	2008-01-27 11:31:45	REJECTED				Info
5329794@sip.messagenet.it	3288169497@sip.messagenet.it	2008-01-27 11:32:30	COMPLETED	14.9 sec	0.1 sec	38 sec	Info

Figure 5.2: VoIPConsole calls page

The sample application is capable to analyse both the signalling protocol and the media transfer protocol. Thus, for each successfully completed call the information reported by the sample application includes the RTP statistics for each media stream involved. RTP statistics are shown in the Call Details page, depicted in Figure 5.3. For each media stream, the number of packet, the jitter and the codec are reported.

The sample application is capable to discover both VoIP calls and VoIP users in real time. Detailed information regarding a specific user are shown in the Peer Details page (Figure 5.4). The information includes some counters



Call Details

Call Parties

Call Id	MWM1YWM4MTUyZTQ1NmNkYzA1NTY3ODhmYzZmNmVlNGU.
Caller	5329794@sip.messagenet.it
Called	3921008547@sip.messagenet.it
Terminated by	unknown
Begin/End Time	2008-01-27 11:30:16 - 2008-01-27 11:30:48
Status	COMPLETED
Setup Time	14.32 sec
Invite Time	0.2 sec

RTP Statistics

	Caller -> Called	Called -> Caller
RTP Codec	PCMU	PCMU
Total Packets	755	855
Malformed Packets	0	0
Packets Out of Order	0	1
Max Jitter	682.89 ms	117.74 ms
Mean Jitter	22.31 ms	27.68 ms

Figure 5.3: VoIPConsole call details page

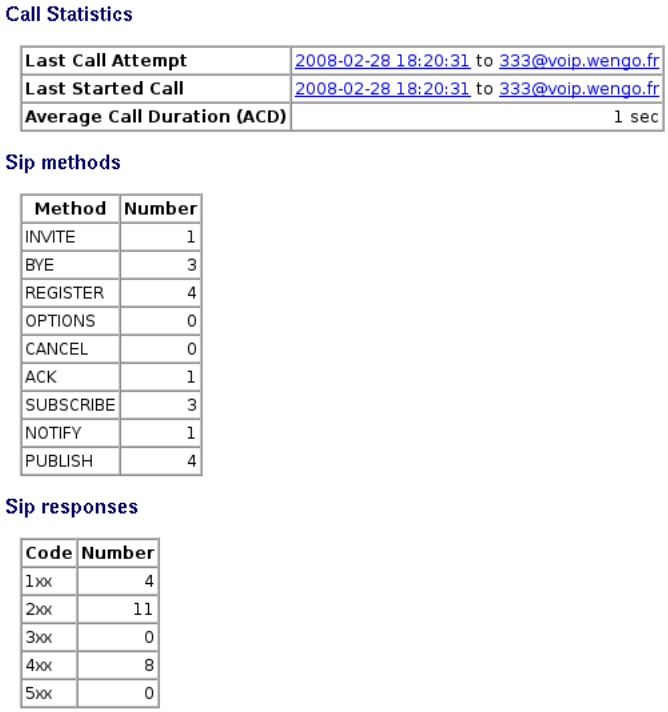
regarding the signalling traffic exchanged by the selected user and the Average Call Duration (ACD).

5.2 Further RTC-Mon use cases

In the previous Section we presented a sample VoIP monitoring application called VoIPMon based on RTC-Mon. However the framework allows the development of more complex applications (Figure 5.5).

RTC-Mon simplifies the development of distributed monitoring architectures used to segment the networks in order to provide a per link view of service parameters. IPFIX network probes, capable to export VoIP information toward a central collector can be implemented on top of RTC-Mon.

VoIP spam, often referred as *Spam over Internet Telephony(SPIT)*, is the proliferation of unwanted automatically dialed phone calls using VoIP.



Call Statistics

Last Call Attempt	2008-02-28 18:20:31 to 333@voip.wengo.fr
Last Started Call	2008-02-28 18:20:31 to 333@voip.wengo.fr
Average Call Duration (ACD)	1 sec

Sip methods

Method	Number
INVITE	1
BYE	3
REGISTER	4
OPTIONS	0
CANCEL	0
ACK	1
SUBSCRIBE	3
NOTIFY	1
PUBLISH	4

Sip responses

Code	Number
1xx	4
2xx	11
3xx	0
4xx	8
5xx	0

Figure 5.4: VoIPConsole peer details page

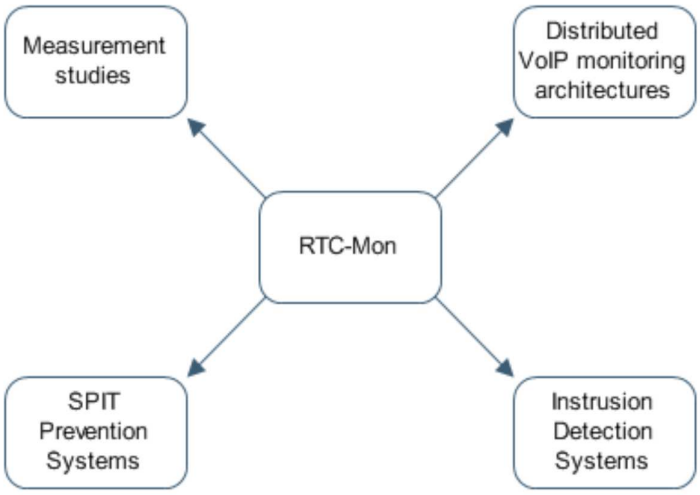


Figure 5.5: Further RTC-Mon use cases

Much like email spam, it's believed that as VoIP becomes more popular, SPIT is sure to follow. Blocking SPIT is far more complex than blocking the traditional email spam, for example because Internet telephony is a synchronous service whereas emails are by their nature asynchronous. SPIT is an active research topic and several SPIT detection methods have been proposed [48], but SPIT detection and prevention research is still at a very early stage. RTC-Mon provides a comprehensive analysis of both media and signalling traffic and allows a rapid prototyping of experimental SPIT detection methods. Since RTC-Mon allows to carry media streams to user space it can be a starting point for the implementation of complex content based methods.

RTC-Mon can also be useful to perform further measurement studies in order to better characterise the VoIP traffic and to understand its impact on current IP networks. Moreover the framework easily allows to conduct statistical surveys on usage of codecs and user equipments.

The framework can be extended in order to discover service usage anomalies. These are used by both *Intrusion Detection Systems* and *fraud management systems*. VoIP specific IDS systems, such as the one proposed in [41] can be easily implemented on top of RTC-Mon.

5.3 Performance 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. Table 5.1 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 range, since it is defined 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 [43]).

The table 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 number. As a result, in the rest of the section we will focus on number of flows from thousands to 50,000.

Codec	**Sample Size (bytes)**	**Sample Rate (ms)**	**Bit Rate (Kbps)**	**Packet Size (bytes)**	**Ethernet 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

Table 5.1: Rate information for various common VoIP codecs.
The figures assume Ethernet/IP/UDP/RTP headers.

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 including an inter-frame gap of 12 bytes and a preamble of 8 bytes; table 5.2 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).

In the rest of the section we evaluate the performance of the RTC-Mon framework as well as that of VoIP Console, the proof-of-concept application

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

Table 5.2: Maximum theoretical rates for Gigabit Ethernet.

built on top of it.

5.3.1 Testbed

As mentioned, we used a small testbed to conduct our tests. The testbed consists of an IXIA 400 traffic generator [31], two computers and an HP Procurve 1800 switch to connect them all (see Figure 5.6). We used the IXIA 400 to generate trash UDP traffic, a computer to generate VoIP traffic and another one as the monitoring system.

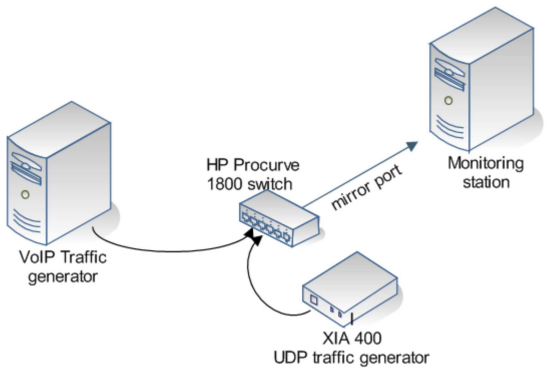


Figure 5.6: Experiment network topology
The UDP traffic generator, the VoIP generator and the network probe are shown

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* [2]. The trace contained about 1,000 calls each lasting 30 seconds: from [5] 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 Supermicro PDSMI+ mainboard, an Intel CeleronD running at 3.2 Ghz and 4GB of DDR2 Ram. The mainboard is a low cost server class mainboard providing two Intel e1000 LOM Gigabit NICs. The monitoring system also run a Linux 2.6.24 kernel; it is connected to the switch using a mirror port and receives the combined traffic from the UDP and the VoIP generators. The device driver used is the e1000 driver developed by Intel and included with the Linux Kernel, which can be expected to maximize the performance by using all the available hardware features on the NIC.

It is worthwhile to note that a low budget server has been chosen as monitoring station. In particular the Intel Celeron is a desktop class CPU whose price is less than 50 euro.

5.3.2 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 signalling 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 discovered 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 analysed 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 filtering traffic in the kernel rather than in user space.

To do so, two different libvoip flavours were implemented:

1. *LibVoIP RTC-Mon*: this is the original LibVoIP built around the kernel enhancements. It make use of both the RTP analyzer and the SIP filter and parser. Thus, the kernel is responsible to perform the first stage signalling parsing and to provide analysis result for a set of analyzed RTP streams. RTP packets never reach the user space. Analysis result are read using the PF_RING polling mechanism.

2. *LibVoIP pfring*: the library use the same packet handling approach of the architecture, however the packet processing and the protocol analysis is completely done in user space. All UDP packets reach the user space (a flat *"udp"* filter has been used). The RTP analysis is performed with the RTP analyzer code, with minor modifications. The signalling parsing is performed by the Dispatcher component in just a single stage, since the

SIP analyzer is no more adopted. This means that each packet is first matched against the RTP analyzer hash. If the packet do not belong to any monitored RTP streams and it is a SIP packet is parsed and then dispatched to the trackers, otherwise discarded. Since PF_RING is used to carry packets from user space to kernel space the parsing up to layer 4 is performed by the *ring* kernel module.

Thanks to the LibVoIP library design it has been quite simple to take advantage of the different capture and filtering technologies. In fact the Dispatcher is the only component that have been modified, so trackers have been kept unchanged regardless of the capture and filtering mechanism employed. Moreover all the flavours share the same SIP parsing code and the RTP analysis code with minor modifications.

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 VoIP traffic generator machine has been used to inject in loop the previously described VoIP traffic trace whereas the IXIA 400 has been configured to generate trash UDP traffic. We then measured the load that the VoIP analysis put on the CPU of the monitoring host.

The results in Figure 5.7 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 60% and 40% for the whole Gigabit range).

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

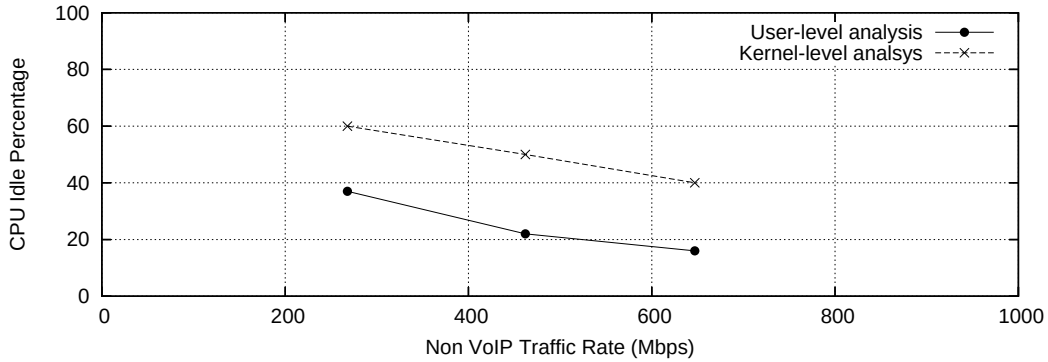


Figure 5.7: Performance when filtering trash UDP traffic from VoIP traffic.

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, *rtpstress*, 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 rtpstress 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; the monitoring system tracked every single one of these flows and kept statistics for them.

Figure 5.8 shows the results of these tests. It contains two graphs per packet size, one representing the maximum loss free packet rate when all of the

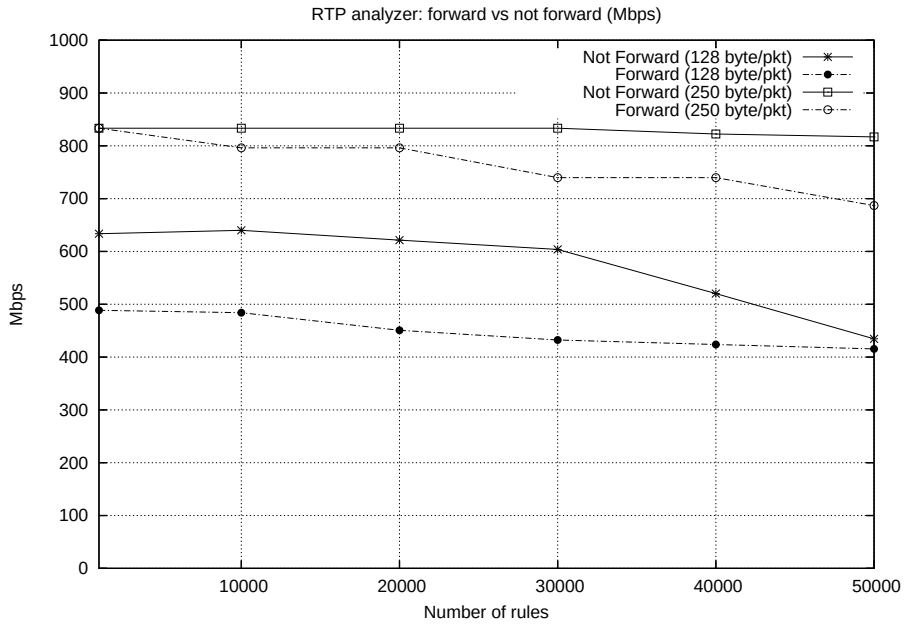


Figure 5.8: RTC-Mon performance when tracking large numbers of RTP flows.

analysis is done in the kernel without packet forwarding enabled, and another one when the packet is copied to the PF_RING circular buffer. 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 (recall Figure 5.2); for 250-byte packets, the rate jumps to about 830Mbps, 90% of the maximum. As expected, copying packets into a PF_RING socket 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.

Another test has been performed in order to evaluate the cost of performing the RTP analysis. To do so we decided to compare the RTP plugin with packet forwarding disabled with *pfcount* which is a simple capture ap-

plication written on top of PF_RING. The application captures every packets and then discards them without doing any protocol analysis.

Figure 5.9 shows the results of these tests. It contains two graph per packet size, one representing the maximum loss free packet rate when the RTP analysis is done in the kernel without packet forwarding enabled, and another one when every RTP packet is captured and later on discarded by pfcount.

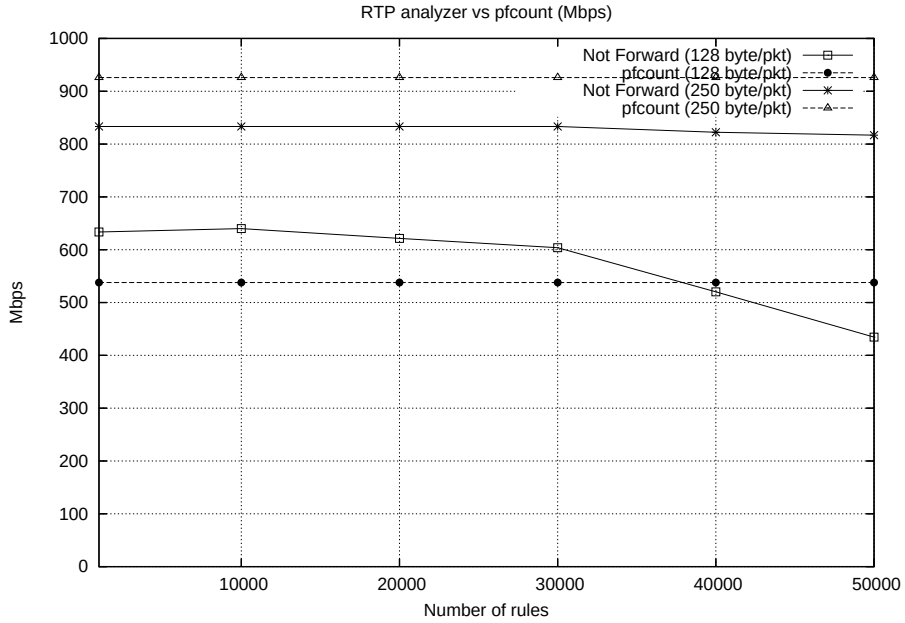


Figure 5.9: RTP analyzer versus pfcount.

The Figure 5.9 shows that performing the RTP analysis inside the kernel without forwarding RTP packets to user space is a big advantage for small packet sizes. For 128-bytes packets the RTP analyzer loaded with up to 30,000 rules can handle more bandwidth than the pfcount application which simply captures and discards packets.

Figure 5.10 shows the CPU idle percentile measured at the maximum

loss free rate. It is worthwhile to note that the RTP plugin is not only capable to handle more bandwidth with 128-bytes packets and up to 30,000 loaded rules, but it can do it leaving more spare CPU cycles on the monitoring system.

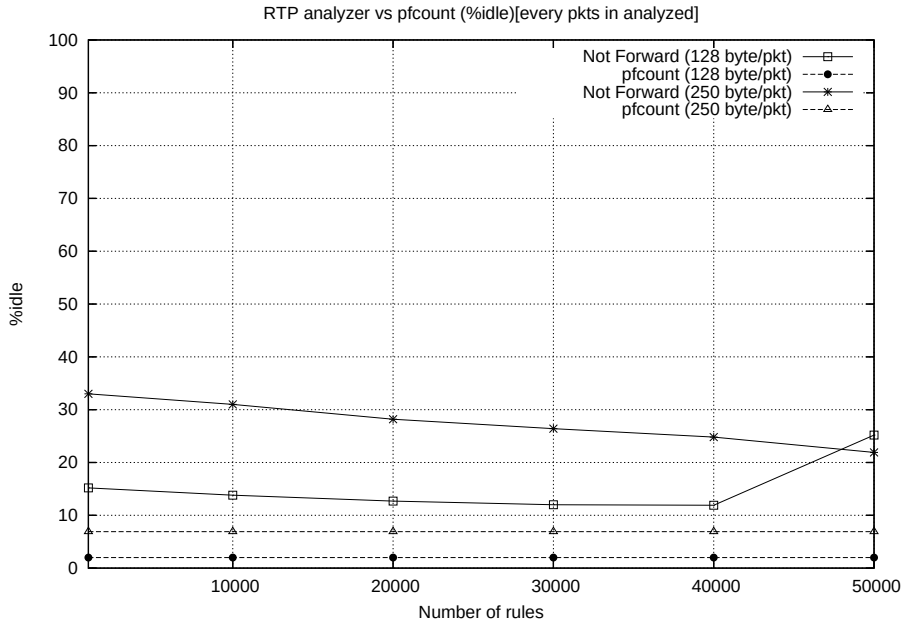


Figure 5.10: Idle CPU percentage measured at the maximum loss free rate.

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 BFP library by writing a simple C program that measures the time needed to compile a complex filter containing many expressions (monitored RTP streams). Table 5.3 shows the number of instructions and the compilation time of a single BPF filter composed by a varying number of different filtering expressions.

For a filter with 200 expressions, the compile time was 800 milliseconds (we tried filters with more expressions but the kernel refused them, returning

	Number of instructions		Compile time (msec)	
# Expressions	Optimized	non Optimized	Optimized	Not optimized
50	868	3748	50	16
100	1868	7498	200	63
200	3868	14998	800	388

Table 5.3: Compile time and number of instructions for a complex BPF filter.

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.

# 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	527

Table 5.4: Time needed to change a rule (a monitored stream) in RTC-Mon. Ins stands for insertion, del for deletion and usec for microseconds.

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 Table 5.4). The results clearly show that it is possible to insert rules much faster than with BFP 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 would go untracked.

5.4 Thesis validation

In order to validate this work is necessary to return to Section 1.5 to see which requirements are satisfied.

1. *Extensibility*

   The framework supports extensibility at different layers. The kernel infrastructure allows the introduction of plugins, implemented as kernel modules. On the other hand *LibVoIP* is extensible by means of pluggable components, called *trackers*.

2. *Ease of use and development*

   RTC-Mon simplifies the development of VoIP monitoring applications because packet filtering, packet parsing and protocol analysis are completely performed by the framework. A VoIP monitoring application that shows the successfully completed calls and performs analysis on the involved media traffic can be implemented with RTC-Mon in less than 30 lines of code.

   In Section 5.1 a more complex RTC-Mon based VoIP monitoring application has been described. The application makes use of the most important building blocks of the framework. It uses the SIP filter and the RTP analyzer and delegates the signalling traffic analysis to the LibVoIP library. Thanks to RTC-Mon the VoIPStorer component, which is responsible to perform the VoIP traffic analysis is quite simple. Its implementation did not require any networking knowledge and most of its code (800 lines of code) consists of SQL instructions.

3. *Scalable and high-performance applications*

   In Section 5.3 the performance of the framework has been evaluated. The RTP analyzer component have been independently tested. The experiments results have shown that the RTP analyzer is scalable in number of different RTP streams and offer great performance. With small sized packet the RTP analyzer outperforms the simplest capture application written on top of PF_RING, that simply discards packets.

4. *Flexibility*

   The framework is structured in layers, so that framework users are allowed to choose the features they need from the proper layer. LibVoIP, the upper layer, is a service oriented layer that can be used to build VoIP monitoring application with very little efforts. Users can extend LibVoIP to introduce more complex service oriented metrics.

   More advanced users can benefit from the lower layers, which are packet oriented. The kernel infrastructure represents a ready to use environment for the implementation of more complex analysis plugins.

5. *Promotion of reuse*

   The enhanced PF_RING core natively supports IP defragmentation and parsing up to transport layer; in addition it provides a polling mechanism, an hash based flow tracker and allows to carry parsing information from kernel space to user space. Those features substantially simplifies the development of custom PF_RING plugins.

6. *Efficient resource utilization and ability to run on environments of limited resources*

Real-time communication services, such as VoIP, produce many media streams carried over UDP packets having dynamically assigned ports. Without an effective filtering mechanism a lot of resources are wasted, since many packets have to be discarded in user space. The framework is able to pick out VoIP packets from a loaded link and to discard non VoIP packets early. Moreover the framework can perform the media stream analysis inside the kernel. Those solutions reduce the number of packet copies and bring to a better resources handling.

7. *Commodity hardware*

   The framework does not require any special hardware to run and can be used with every network interface card supported by the GNU/Linux operating system. Unlike other solutions it does not rely on specialised monitoring hardware to achieve performance.

Chapter 6

Final remarks

This work has shown that VoIP monitoring is necessary in order to meet VoIP user expectations, to allow a proactive management and to detect in real time service degradations caused by misconfiguration, network congestion or security attacks.

Monitoring current high speed VoIP networks is a challenge for many reasons. First, the traffic produced by VoIP services is the worst traffic to analyze in a passive way because it is mostly composed by RTP streams using dynamically assigned ports and because each stream is composed by small sized packets. Second, several indicators have to be considered in order to define and measure the overall quality of experience.

A novel monitoring framework has been designed and implemented to solve the previously mentioned issues. The main results of this work are a modular kernel infrastructure providing a SIP packet filter and a RTP protocol analyzer, an user level library exploiting the features offered by the kernel infrastructure and a sample VoIP monitoring application implemented as a test bed for the proposed framework.

The validation phase has shown that the adoption of a mixed kernel space user space approach is the key to achieve high performance. A large VoIP network has been simulated using synthetic traffic and some experiments have been conducted in order to compare the framework against the current state of the art solutions for packet filtering and packet capture. First of all, the filtering solution adopted allows to pick out VoIP traffic from a loaded link and to discard non VoIP traffic at the kernel level. This reduce the CPU load on the monitoring system by a factor of two in case of highly loaded links. Moreover the RTP analyzer is capable to analyze several thousands of different audio/video streams using a low budget server. The time needed to dynamically add a new discovered stream to the monitored set is below 3 ms even with a monitoring set composed by 50,000 different streams. This value is two order of magnitude lower than time needed to dynamically change a filter corresponding to only 200 media streams with the current state of art filtering technology.

6.1 Open issues and future work

The framework already provides a large set of information including signalling performance indicators and streaming performance indicators such as jitter. There are, however, several aspects to the framework that requires further work.

The CallTracker component have to be extended in order to compute some more signalling metrics.

Moreover a very valuable addition would be the introduction of the RTCP protocol dissection. In particular the end-to-end delay, which can be

computed using the information present in RTCP packets, would have been very useful to compute the MOS (Mean Option Score). Another interesting addition would be the ability to generate RTCP reports regarding the observed RTP streams. Furthermore, the framework assumes that the signalling traffic and the media traffic can be captured from the same observation point. This, however, is not always true in practice. So the framework has to be extended in order to allow the remote instrumentation of RTP analysis probes.

The modular kernel infrastructure, on top of which the SIP filter and the RTP analyzer have been implemented, is not bound to VoIP monitoring. An IPFIX[9] plugin is currently under development; an HTTP plugin has already been implemented. However the framework is packet oriented and does not have a TCP reassembler. The lack of a TCP reassembler is not a big issue for VoIP protocols, but it can represent a limitation for some protocols, such as HTTP, where some fields can be split across different IP packets (e.g. the url for the HTTP protocol).

The framework has been designed and implemented in order to build complex VoIP monitoring architectures. The VoIPMon sample application is no more than a prof of concept intended as a test bed to measure the framework against the monitoring requirements. In the future, we plan to use the framework for the implementation of distributed VoIP monitoring architectures capable to provide to network managers a per link view of the VoIP service quality.

Appendix A

Session Initiation Protocol

According to the definition in *RFC3261* [53], Session initiation protocol(SIP) is an application layer control (signalling) protocol for creating, modifying and terminating sessions with one or more participants. These sessions include multimedia conference, multimedia distribution and least but not last Internet telephony calls.

A.1 Purpose of SIP

SIP supports five facets of establishing and terminating multimedia communications:

1. *user location*: determination of the end system to be used for communication

2. *user availability*: determination of the willingness of the called party to engage in communications;

3. *user capabilities*: determination of the media and media parameters to be used;

4. *session setup*: "ringing", establishment of session parameters at both called and calling party;

5. *session management*: including transfer and termination of sessions, modifying session parameters, and invoking services.

SIP dictates no protocol to be used inside a session. Anyway SIP follows the standard IETF approach which have always been protocol reuse. The most common use of SIP is to describe audio and video session and SIP adopts the *Session Description Protocol (SDP)* as payload to describe media streams.

A.2 Transport protocols

SIP does not make any assumption regarding the transport protocol to be used: it can run on top of TCP, UDP or SCTP[52]. Since SIP implements its own retransmission mechanism to recover from loss packet and since it is a connection less protocol usually SIP uses UDP as transport protocol. Moreover the adoption of UDP, rather than TCP, offers some advantages in term of response time and resource usage.

IANA[1] assigned port number 5060 for UDP, TCP and SCTP, and 5061 for TCP over TLS.

[1]Internet Assigned Number Authority

A.3 SIP entities

A.3.1 User agents

User agents (UAs) are endpoints that use SIP protocol to find each other and to negotiate session characteristics. UA can be physical devices (such as desk phones or PDAs) or software applications which interact with human users, but also services like PSTN gateways and so on.

Each UA acts as two different logical entities:

- *User Agent Client (UAC)*: it creates request messages and uses the client transaction state machinery to send it and wait for the response
- *User Agent Server (UAS)*: it accepts requests from UAC and generate a response. The response accept, reject or redirect the request.

A.3.2 Registrar

The Registrar server process the registration requests and places the information it receives in those requests into the location service for the domain it handles. It is the front-end to the user location service and it is often colocated within the Proxy server of its domain.

A.3.3 Proxy server

The proxy server is an intermediate device that receives SIP requests from a client and then forwards the requests on the client's behalf. It is used primarily for routing purposes: its aim is to forward the messages to another entity

"closer" to the targeted user. This entity can be a proxy server, a redirect server or an User Agent.

A.3.4 Redirect server

It receives request messages and sends back a list of alternative URIs in a `3xx` class response. Those URI can be used by the UAC to get closer to the target UAS. Redirect servers are mainly used to reduce the load of routing requests, pushing back routing informations to the requester.

A.4 SIP messages

Like in HTTP, SIP messages can be requests from a client to a server or responses to a request. For all the messages the general format is the following:

1. a mandatory start line (request or response)
2. some header fields
3. an empty line
4. an optional message body Each line ends with a carriage return-line feed (CRLF).

SIP requests

The first line of a SIP request has the following structure:

```
METHOD Request-URI SIP/2.0
```

RFC3261 define six type(method) of requests(listed in Table A.1); anyway it allows extensions to specify new methods.

Method	Description
INVITE	Request to establish a new session
ACK	Confirms a final response reception
BYE	Terminates a successfully established call
CANCEL	Used to CANCEL a previous request sent by a client
OPTIONS	Queries the capabilities of the server
REGISTER	Registers the address listed in the To header field with a SIP server

Table A.1: SIP requests

SIP responses

The first line of a SIP response has the following structure:

```
SIP/2.0 Status-Code Reason-Phrase
```

RFC3261 defines six classes of status codes (listed in Table A.2); the `Reason-Phrase` gives additional information about the corresponding status code.

Code	Description
1xx	Provisional response: the request has been received
2xx	The action was successfully received, understood and accepted
3xx	Further actions needs to be taken in order to complete the request
4xx	Server error: the server failed to fulfil the request
5xx	Global failure: the request cannot be fulfilled at any server

Table A.2: SIP status codes

A.4.1 Header fields

Among the other fields that can follow the request line `From`, `To`, `Call-ID`, `Via`, `CSeq` and `Max-Forwards` are mandatory. These six header fields are the fundamental building blocks of a SIP message, as they jointly provide for most of the critical message routing services including the addressing of messages,

the routing of responses, limiting message propagation, ordering of messages and the unique identification of transactions. Table A.3 shows the header set that has some relevance in order to better understand this work. In the following sections a description of each field is given.

Field	Description
`Call-ID`	a globally unique identifier for a call
`To`	logical recipient of the request
`From`	logical identity of the initiator of the request
`CSeq`	used to identify retransmissions
`Max-Forwards`	maximum number of times a message can be forwarded
`Contact`	a SIP URI that represents a direct route to contact an user
`Via`	indicates the transport used for transmitting the request and identifies the address to which the response has to be sent

Table A.3: SIP header fields

From

The `From` field indicates the logical identity of of the initiator of the request. It contains a URI (a SIP URI or another URI schema) and optionally a display name. The following is an example of a valid `From` field:

```
From: "Luca Deri" <sip:luca@ntop.org:5060>
```

To

The `To` field indicates the logical recipient of the request. This may be or not the ultimate recipient of the request. The following is an example of a valid `To` field:

```
To: <sip:luca@ntop.org:5060>
```

Call-ID

The `Call-ID` is an unique identifier to group together a series of messages and the same `Call-ID` value is present in subsequent messages belonging to the same dialogue. The following is a valid Call-ID header field:

```
Call-ID: 12456124561245612456
```

CSeq

The `CSeq` contains an integer and a method name. The integer is a traditional sequence number, incremented for each new request. The `CSeq` field has the following structure:

```
CSeq: number METHOD_TYPE
```

A.5 SIP requests

A.5.1 REGISTER

SIP users register their current location to a registrar server using `REGISTER` request. Registrar servers act as the front end to the location service for a domain, reading and writing mappings based on the contents of `REGISTER` requests. The location service is consulted by the proxy server that is responsible for routing requests for that domain.

A REGISTER request specifies the SIP user with the `To` header field and the current IP address using the `Contact` field. Moreover the UAs may indicate

how long the mappings should be considered valid, specifying a duration in seconds in the `Expire` header.

A.5.2 INVITE, ACK, BYE

The INVITE method is used to establish a session between two UAs. This request initiates a transaction. An INVITE request can receive the following common provisional answers:

- *100 Trying*: this response is sent by intermediate SIP nodes in order to stop retransmission of the INVITE request
- *180 Ringing*: this response is sent by the remote UA to indicate that the user is being notified of the incoming call, but has not yet answered

The transaction ends with a final response. Typical responses are:

- *200 OK*: the call has been accepted by the remote user
- *486 Busy Here*: the remote user is busy

A.5.3 SUBSCRIBE, NOTIFY

RFC3265[50] is a SIP extension defined in order to allow request notification from remote nodes indicating that certain events have occurred. For this purpose two new methods has been defined: `SUBSCRIBE` to request notifications and `NOTIFY` to report. A simple usage of this mechanism is the presence subscription, defined in *RFC3856*[51].

A.6 Authentication

SIP defines a stateless, challenge-based authentication mechanism similar to HTTP authentication described in RFC 2617[21]. When a registrar server receives a request that have to be authenticate, it responds with a `401 Unauthorized` message, conveying the *challenge* in the `WWW-Authenticate` header field. Then the UAC, re-issues the request again adding an Authorization header containing the response to the challenge. A sample message exchange is depicted in Figure A.1

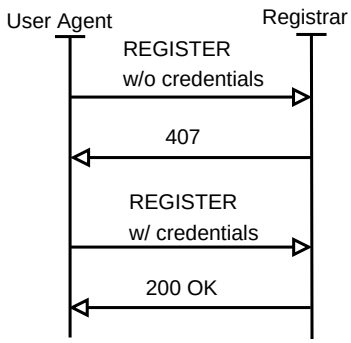


Figure A.1: UAC registration

The same procedure applies to proxies, except that the authentication response message is `407 Proxy Authentication Required`.

A.7 Message routing

A.7.1 SIP transactions

A transaction is a series of independent SIP messages exchanges. A transaction starts with a request, may include some provisional responses (`1xx`) and ends with a final response. SIP requires that responses follow the reverse path of

the request, i.e. they traverse the same SIP entities but, in reverse order. This can be accomplished using the `Via` header. At every hop of the request a `Via` header is added to the message before the previous one so that the UAS receiving the request has a list of all the SIP network elements that has been traversed.

A.7.2 SIP trapezoid

Figure A.2 shows the typical message exchange for a VoIP call, that is commonly referred as *sip trapezoid*. The user *alice*, identified by the URI *alice@ntop.org* uses its phone to call the user *bob@unipi.it*. The intermediate proxy servers help to setup the session on behalf of the users. The outbound proxy for user *alice* is responsible to locate the proxy server of the target user and to forward any request to the target UA.

After the message exchange of the transaction, both the UAs know a valid transport (from the `Contact` header) to reach their peer, so that they should send subsequent message directly, without traversing SIP proxies.

A.8 SIP and VoIP

SIP is a general purpose session establishment protocol. In relation to VoIP, SIP has the role of delivering the offer and the answer containing the description of the multimedia session to be established. Information about the sessions are exchanged using the *Session Description Protocol(SDP)*??.

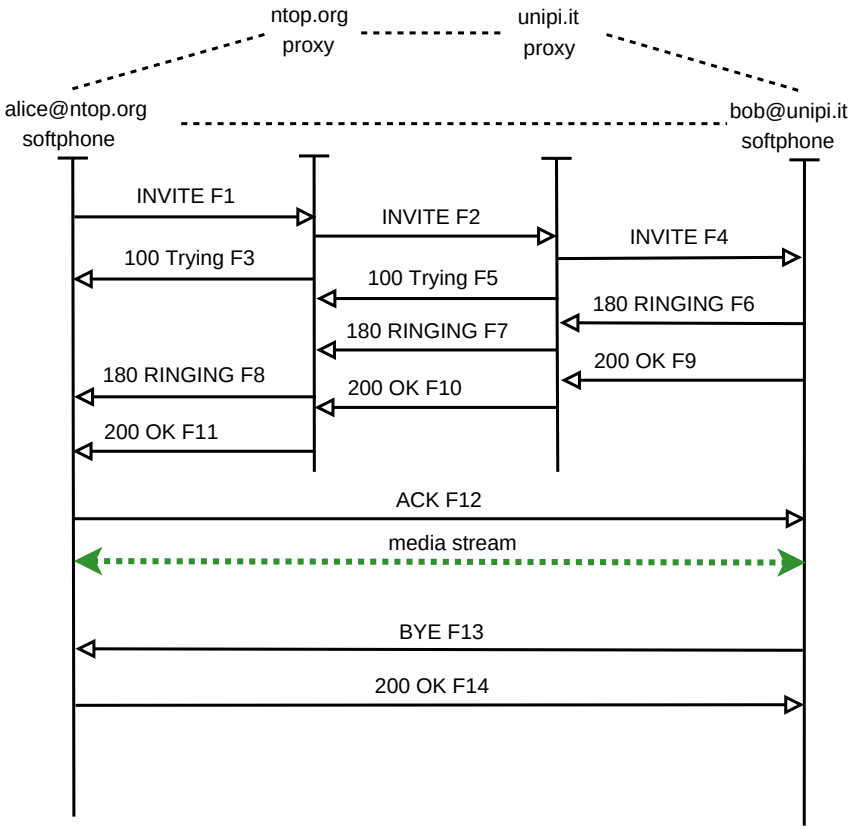


Figure A.2: SIP trapezoid

A.8.1 SDP

SDP provides a textual description of the session to be established, independently of the actual transport protocol. It is usually employed to describe RTP and RTCP flows, described in Appendix B.

The description of such a session includes the following information:

- session name and purpose
- information needed to receive the media (addresses, ports, formats, etc
- contact information for the person responsible for the session

SDP is a textual protocol composed by a number of text lines of the form `type=value`, where type is always one character long and case significant, while value is a structured text.

An SDP session description consists of a *session level description* and optionally several *media-level descriptions*. The session level part starts with a `v=` line and continues to the first media-level section. Each media description starts with a `m=` line.

The session level description lines we are interested in are the following:

- `v=<protocol version>`

  current version is 0

- `o=<username><session id><version><network type><address type><address>`

  this line specifies the name of the user who is going to create the session. The session id is a randomly generated number; the version is the sequence number of the session announcement; network type is a string representing the type of network(`IN` stands for Internet); address is the IP address of the UA.

- `c=<network type><address type><address>`

  This line can be omitted if every media level description provides its own connection information. The fields of this line follow the same convention of the line `o=`.

The media level description lines we are interested in are the following:

- `m=<media><port><protocol><format>`

```
v=0
o=francesco 2890844526 2890844526 IN IP4 192.168.0.210
s=-
c=IN IP4 192.0.0.210
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Figure A.3: A sample SDP offer

`media` can be audio, video, text and other values. `port` and `protocol` specify the reception port and the protocol used, which is usually `RTP/AVP` *(RTP with Audio-Video Profile)*.`format` is the list of supported codecs[2].

- `a=rtpmap:<payloadtype> <encoding name>/<clock rate>[/ <encoding parameters>]`

  `rtpmap` attributes are used to convey additional information regarding payload types specified. Up to one `rtpmap` attribute can be specified for each media format.

- `c=<network type><address type><address>`

  it is the same `c=` line of the session level description.

  Figure A.3 shows an SDP offer example.

A.8.2 Session negotiation

SDP messages are exchanged as SIP payloads between the two parties. In this way during the session creation the characteristics of the session are negotiated. The caller appends the SDP as payload of the SIP `INVITE` request, while the called party inserts it in the `200 OK` response.

[2]IANA

Appendix B

Real-time Transfer Protocol

Real-Time Transfer Protocol (RTP), defined in RFC3550 [55], provides transport functions to carry real-time data, such as audio and video over multicast or unicast networks.

RTP is used to transport real-time data, but it does not ensure timely delivery or provide other quality of service guarantees. RTP relies on lower level services to do so, but it does not assume that the underlying services are capable to grant delivery or prevent out of order delivery. RTP usually uses UDP as transport protocol.

RTP provides the following functions suitable for real-time content delivery:

- *payload type identification*: the information is used by the receiver to know what kind of content is being transferred.
- *sequence numbering*: PDU[1] sequence numbers are mainly used to detect losses. The sequence numbers increase by one for each RTP packet

[1]Protocol Data Unit

transmitted.

- *time stamping*: it is used to synchronize the coder and the encoder; timestamps are used to place the incoming audio or video packets in the correct timing order. Subsequent packets may have the same timestamps.

B.1 RTP sessions

A RTP session is an associations of participants communicating with RTP. A participant may be involved in multiple sessions at the same time. This is what usually happens for multimedia sessions where each media is carried in a separate RTP session. For example a video call involves the creation of two distinct RTP sessions: the first one for the audio and the second one for the video.

RTP uses the transport address to multiplex different RTP sessions. Each RTP session is identified by a pair of transport addresses. A transport address is composed by an IP address and by a pair of UDP ports. The first port is used to identify the UDP flow on top of with the RTP PDUs are transported. The second port identify the UDP flows carrying RTCP PDUs. RTCP will be described in Section B.3.

An alternative choice would have been to transfer over the same RTP session different media streams corresponding to the same application. However this choice would have introduced several problems:

- would have been much more difficult to dynamically change the payload type for one of the media flows

- mixers would have to recognize each media carried over the same RTP session
- the reception of a single media (e.g. audio) would have been more complex

The adoption of transport addresses to multiplex different RTP sessions solves the previously listed issues, but introduces some other issues for monitoring applications performing the passive analysis of RTP sessions. Using a different RTP session for each media increases the number of different RTP flows to be monitored. For example, an audio call involves the creation of two different RTP streams whereas a video call produces four RTP streams.

B.2 RTP header

Figure B.3 shows the RTP header structure.

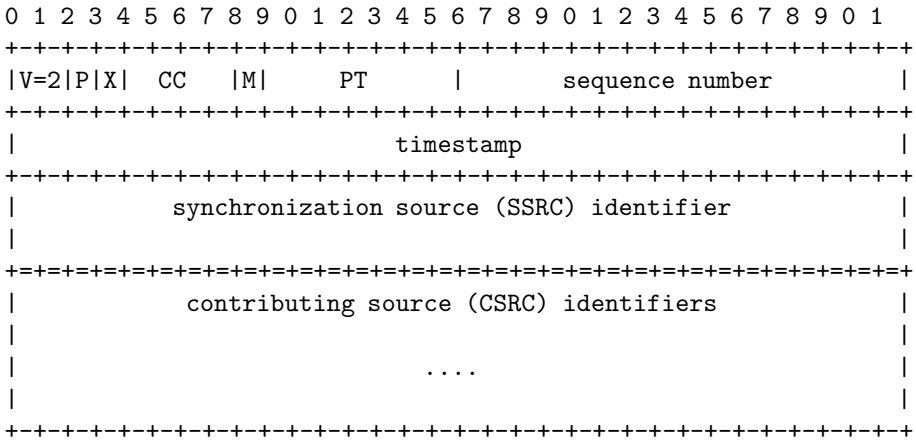


Figure B.1: RTP header

- *version(V)*: 2 bits

  This field identifies the version of RTP.

- *padding(P)*: 1 bit

  If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the payload.

- *extension(X)*: 1 bit

  RTP header is extensible by means of header extensions. If the extensions is set, the fixed header is followed by exactly one header extension.

- *CRCC count(CC)*: 4 bits

  Identifies the number of CSRC identifiers that follow the fixed header.

- *marker(M)*: 1 bit

  Used by specific applications.

- *payload type(PT)*: 7 bits

  The payload type identifies the format of the RTP payload. This field is not intended for multiplexing separate media, thus it is usually constant for a given RTP stream.

- *sequence number*: 16 bits

  The sequence number is incremented by the transmission source by one for each RTP packet sent. It is used by the receiver to detect packet loss and for packet ordering. The initial sequence number for a given RTP stream is chosen randomly.

- *timestamp*: 32 bits

  The timestamp reflects the sampling instant of the first octet in the RTP packet. The sampling instant must be derived from a clock that increments monotonically and linearly in time to allow synchronization and jitter computation.

- *SSRC*: 32 bits

  The source of a stream of RTP packets is identified by a randomly chosen numeric identifier, called SSRC. The source of a stream is called synchronization source.

- *CSRC list*: 0 to 15 items, 32 bits each

  This field is updated by intermediate systems called mixers. Mixers combine multiple RTP streams into a single one, which is identified by a different SSRC. The CSRC list contains the SSRC of the contributing sources for the newly created stream.

B.3 RTCP

Besides RTP, another protocol is optionally used to convey streaming information: the *RTP Control Protocol (RTCP)*, defined in RFC3550 [55]. The main purpose of RTCP protocol is to provide feedback on the quality of data distribution. The protocol is based on the periodic transmission of control packets to participant of RTP session, using the same distribution mechanism as the data packets.

Like RTP packets, RTCP packets begins with a fixed part, followed by

structured elements with variable lengths. RTCP defines five types of RTCP packets:

1. `SS`: Sender report, conveys transmission and reception statistics from participants that are senders.
2. `RR`: Receiver report, for reception statistics form participants that are receivers and not senders
3. `BYE`: Used to indicate end of participation
4. `SDES`: Conveys additional information regarding the source
5. `APP`: Application specific function

SR and RR packets conveys information regarding end-to-end RTP stream quality. Figure B.2 shows the structure of a sender report RTCP packet. The packet is composed by an header, a sender information which is followed by zero or more report blocks.

The header contains the version and padding fields, such as in RTP packets. The payload type identify the packet as SR RTCP packet. The length represents the packet length expressed in bytes.

The second section, the sender information, is 20 bytes long and contains information regarding the RTP stream sent by the sender issuing the RTP sender report. The fields have the following meaning:

- *NTP timestamp*: 64 bits

  Represents the time when the Sender Report has been sent.

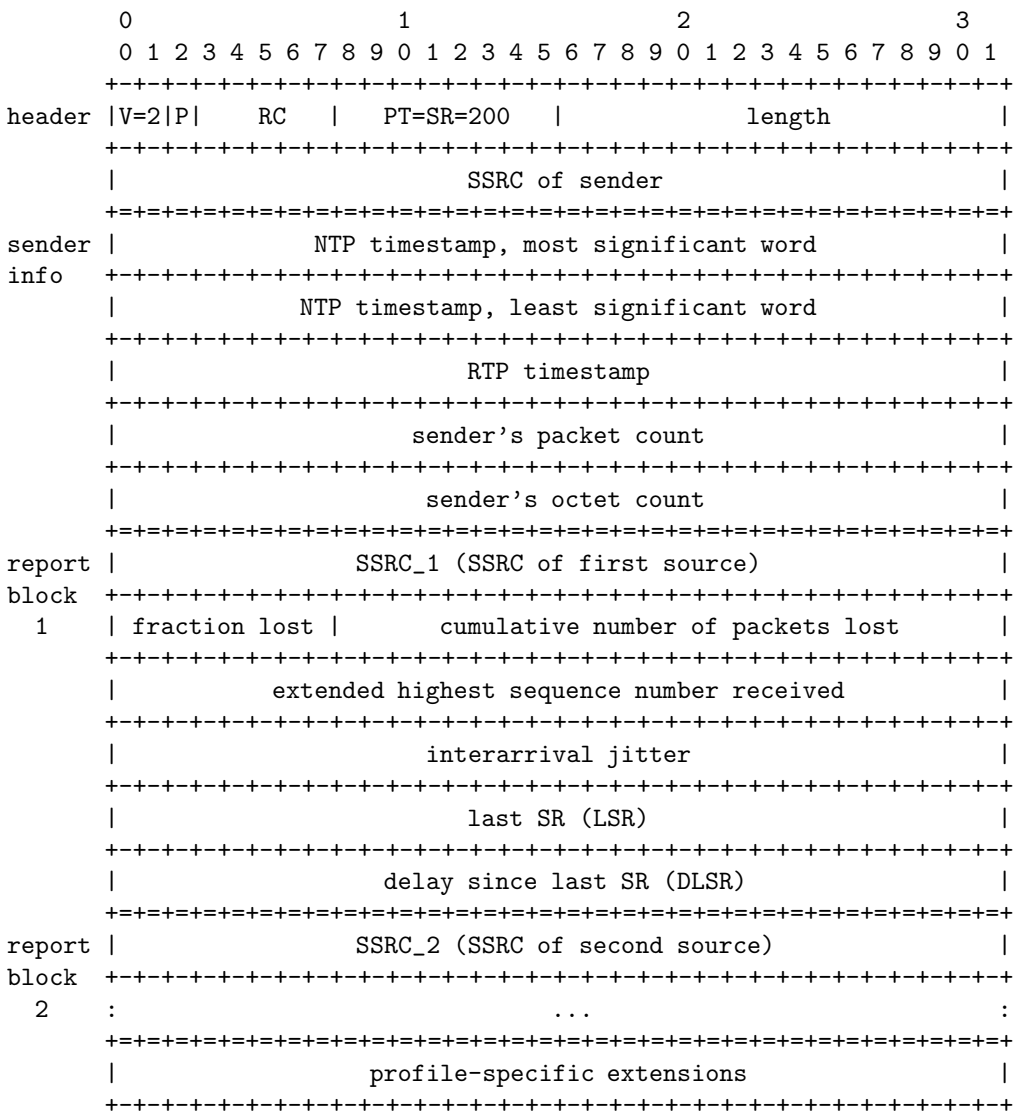


Figure B.2: RTCP sender report header

- *RTP timestamp*: 32 bits

  It is the same time of NTP timestamp, but expressed in the same units and with the same random offset as the RTP timestamps in RTP packets.

- *sender's packet count*: 32 bits

  The total number of RTP packets sent by the sender since starting transmission.

- *sender's octet count*: 32 bits

  The total number of bytes transmitted in RTP packets by the sender. Header and padding are excluded by octet count.

The third sections contains reception report blocks. Each reception report block conveys statistics on the reception of RTP packets from a specified synchronization source (sender).

- *SSRC_i*:

  The synchronization source identifier.

- *fraction lost*: 8 bits

  The fraction of RTP packets lost since the previous SS or RR was sent.

- *cumulative number of packet lost*: 24 bits

  The total number of RTP packets received from SSRC_i that have been lost since the beginning of reception.

- *highest sequence number*: 32 bits

  The low 16 bits represent the highest sequence number received from SSRC_i. The other couple of bytes represent the sequence number cycles.

- *jitter*: 32 bits

  The jitter is an estimate of the statistical variance of the RTP data packet interarrival time. The jitter is measured in timestamp units and expressed as an unsigned integer. The jitter is the difference of the "relative transit time" for two subsequent packets; the relative transit time is the difference in timestamp units between a packet's RTP timestamp and the receiver clock at the time of arrival.

  The difference D of the "relative transit time" between two packets i and j can be computed using the following formula:

$$D(i,j) = (R_j - R_i) - (S_j - S_i) = (R_j - S_j) - (R_i - S_i)$$

  where R_k is the time of arrival of packet k and S_k is the RTP timestamp of packet k.

  The jitter should be computed continuously and the new computed value must be averaged using the following formula:

$$J(i) = J(i-1) + (|D(i-1,i)| - J(i-1))/16$$

- *last SR timestamp(LSR)*: 32 bits

  The middle 32 bits out of 64 in the NTP timestamp received as part of a previous SR packet from SSRC_i.

- *delay since last SR(DSLR)*:

  The delay between receiving the last SR packet from SSRC_i and sending this reception report block.

Receiver report (RR) packets have the same structure of Sender report (SR) packets excepts for the sender info section which is omitted. The payload type for RTCP packets contains the constant 201.

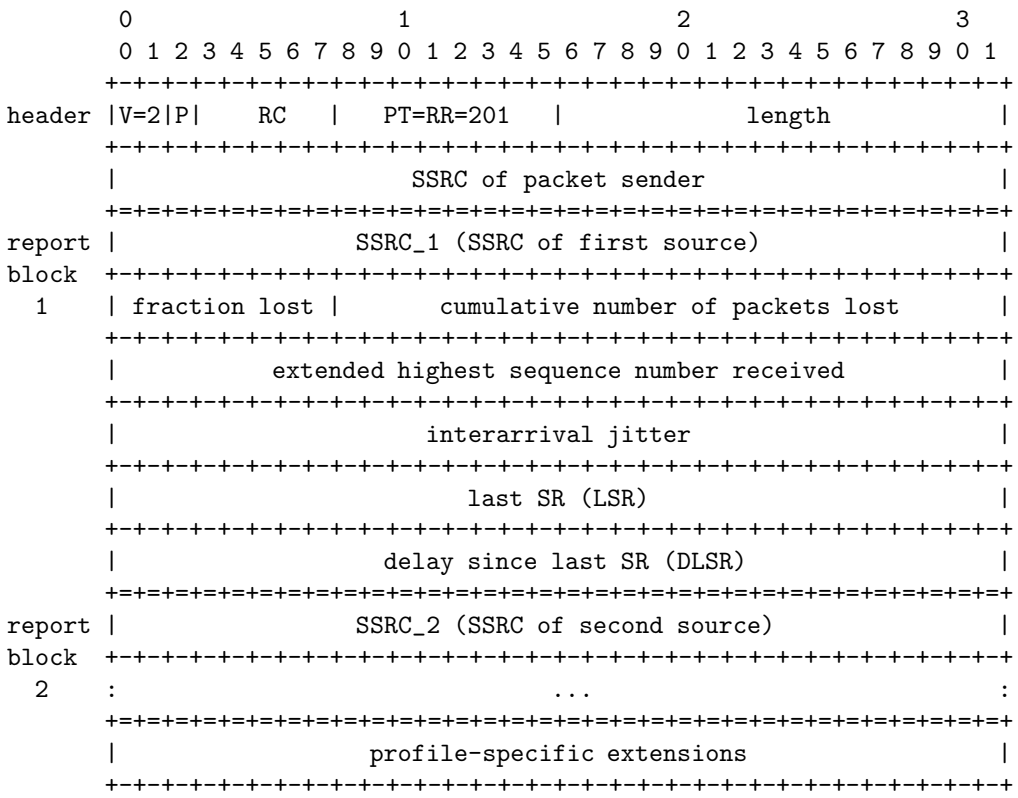


Figure B.3: RTCP receiver report

If both the sender and the receiver exchange SR and RR packets each pair of the communication is able to compute an estimation of the round trip time.

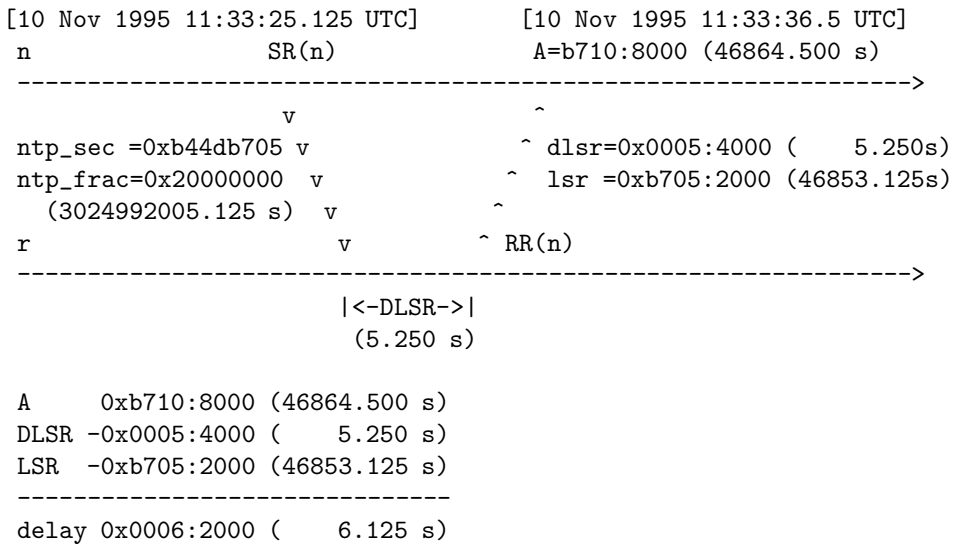


Figure B.4: Round trip time computation

Acknowledgments

First of all, I wish to present my sincerest gratitude to Dr. Luca Deri for introducing me into the field of network monitoring and for his encouragements during these years.

I would like to thank NEC Europe Ltd. for welcoming me to work on this subject at the NEC Network Laboratories in Heidelberg. I especially thank Dr. Saverio Niccolini for his valuable discussions on the problems examined in this study.

Endless gratitude goes to my family, who allowed me to fulfil my wishes and supported me in everything I did. Last but not least, I would like to thank all my study mates in Pisa and friends.

Fusco Francesco

The University of Pisa
July 2008

Bibliography

[1] libpcap. www.tcpdump.org/libpcap.

[2] RRDTool - About RDDTool. http://tcpreplay.synfin.net.

[3] 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.

[4] 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.

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

[6] 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 Opearting Systems Design & Implementation*, pages 24–24, Berkeley, CA, USA, 2004. USENIX Association.

[7] Cavium. octeon. http://www.cavium.com/.

[8] 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.

[9] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. Number 5101 in Request for Comments. IETF, January 2008.

[10] Jan Coppens, Steven Van den Berghe, Herbert Bos, Evangelos Markatos, Filip De Turck, Arne sleb, and Sven Ubik. SCAMPI: A scalable and programmable architecture for monitoring gigabit networks. In *Proceedings of E2EMON Workshop*, Belfast, UK, September 2003.

[11] Tilera Corporation. TILExpress-64 Card. http://www.tilera.com/.

[12] Ana Flàvia M. de Lima, Leandro S. G. de Carvalho, José Neuman de Souza, and Edjair de Souza Mota. A framework for network quality monitoring in the voip environment. *Int. J. Netw. Manag.*, 17(4):263–274, 2007.

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

[14] Luca Deri. High-speed dynamic packet filtering. *Journal of Network and Systems Management*, 15(3):401–415, 2007.

[15] Nick Duffield and Carsten Lund. Predicting resource usage and estimation accuracy in an ip flow measurement collection infrastructure. In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 179–191, New York, NY, USA, 2003. ACM.

[16] Nick G. Duffield, Carsten Lund, and Mikkel Thorup. Learn more, sample less: control of volume and variance in network measurement. *IEEE Transactions on Information Theory*, 51(5):1756–1775, 2005.

[17] ITU-T E.411. Series e: Overall network operation, telephone service, service operation and human factors. 1998.

[18] Endace. DAG network monitoring cards. http://www.endace.com/.

[19] David Endler and Mark Collier. *Hacking Exposed VoIP: Voice Over IP Security Secrets & Solutions (Hacking Exposed)*. McGraw-Hill Osborne Media, 2006.

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

[21] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999.

[22] F. Fusco, F. Huici, L. Deri, S. Niccolini, and T. Ewald. Enabling high-speed and extensible real-time communications monitoring. Ready for submission to the IFIP/IEEE International Symposium on Integrated Network Management, jun 2009.

[23] ITU-T Recommendation G.107. The e-model, a computational model for use in transmission planning. *ITU International Telecommunication Union*, 2002.

[24] GNU. The GNU oSIP library. http://www.gnu.org/software/osip.

[25] S. Gokhale and J. Lu. Signaling performance of sip based voip: A measurement-based approach. 2005.

[26] The Tcpdump Group. tcpdump. http://www.tcpdump.org.

[27] The Wireshark Group. Wireshark. http://www.wireshark.org.

[28] P. Gupta and N. McKeown. Algorithms for packet classification, 2001.

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

[30] IST-SCAMPI. A scaleable monitoring platform for the internet. http://www.ist-scampi.org.

[31] IXIA. Ixia - leader in ip performance testing. http://www.ixiacom.com/.

[32] Panos Lekkas. *Network Processors: Architectures, Protocols and Platforms*. McGraw-Hill, Inc., New York, NY, USA, 2003.

[33] J. Levandoski, E. Sommer, and M. Strait. Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net/.

[34] D. Malas. SIP End-to-End Performance Metrics, February 2008.

[35] 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.

[36] Steven McCanne and Van Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *USENIX'93: Proceedings of the USENIX Winter 1993 Conference*, pages 259–270, Berkeley, CA, USA, 1993. USENIX Association.

[37] J. Mogul. Efficient use of workstations for passive monitoring of local area networks. *SIGCOMM Comput. Commun. Rev.*, 20(4):253–263, 1990.

[38] J. Mogul, R. Rashid, and M. Accetta. The packer filter: an efficient mechanism for user-level network code. *SIGOPS Oper. Syst. Rev.*, 21(5):39–51, 1987.

[39] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proceedings of Winter Usenix Conference*, January 1996.

[40] Napatech. X Adapter Family. http://www.napatech.com/.

[41] Mohamed Nassar, Saverio Niccolini, Radu State, and Thilo Ewald. Holistic voip intrusion detection and prevention system. In *IPTComm '07: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 1–9, New York, NY, USA, 2007. ACM.

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

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

[44] ITU-T Recommendation P.563. Single-ended method for objective speech quality assessment in narrow-band telephony applications. *ITU International Telecommunication Union*, 2004.

[45] ITU-T Recommendation P.861. Objective quality measurement of telephone-band (300-3400 hz) speech codecs. *ITU International Telecommunication Union*, 1998.

[46] ITU-T Recommendation P.862. Perceptual evaluation of speech quality (pesq): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs. *ITU International Telecommunication Union*, 2001.

[47] P.Wood. http://public.lanl.gov/cpw.

[48] J. Quittek, S. Niccolini, S. Tartarelli, M. Stiemerling, M. Brunner, and T. Ewald. Detecting spit calls by checking human communication patterns. In *ICC '07: Proceedings of the 2007 International Conference on Computer Networks and Mobile Computing*, pages 1979–1984, Washington, DC, USA, 2007. IEEE Computer Society.

[49] L. Rizzo. Device polling support for freebsd, 2001.

[50] A. B. Roach. Session initiation protocol (sip)-specific event notification. RFC 3265 (Proposed Standard), June 2002.

[51] J. Rosenberg. A presence event package for the session initiation protocol (sip). RFC 3265 (Proposed Standard), August 2004.

[52] J. Rosenberg, H. Schulzrinne, and G. Camarillo. The Stream Control Transmission Protocol (SCTP) as a Transport for the Session Initiation Protocol (SIP). RFC 4168 (Proposed Standard), October 2005.

[53] 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.

[54] Jamal Hadi Salim. When napi comes to tawn, 2005.

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

[56] H. Schulzrinne, D. Oran, and G. Camarillo. The Reason Header Field for the Session Initiation Protocol (SIP). RFC 3326 (Proposed Standard), December 2002.

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

[58] Sourceforge. Sofia-SIP Library. http://sofia-sip.sourceforge.net/.

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

[60] 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.

[61] Takanen A. Wieser C., Rning J. Security analysis and experiments for voice over ip rtp media streams. In *8th International Symposium on Systems and Information Security (SSI'2006)*, November 08-10 2006.

[62] winpcap. The packet capture and network monitoring library for windows. www.winpcap.org.

[63] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 13–13, Berkeley, CA, USA, 1994. USENIX Association.