POLITECNICO DI BARI

Faculty of Engineering

Department of Electrical and Electronics Engineering

UNIVERSITAT POLITÈCNICA DE CATALUNYA - ETSETB

Faculty of Engineering

Department of Computer Architecture

Second level degree in Computer Science Engineering

Master thesis
in
Multimedia transmission and coding

Evaluation of video live transmission on P2P and LISP based architecture

Supervisors

Prof. Luigi Alfredo Grieco

Prof. Jordi Domingo-Pascual

Candidate

Pantaleo Mastrapasqua

Co-supervisors

Prof. Gennaro Boggia

Prof. Albert Cabellos-Aparicio

Eng. Loránd Jakab

Academic Year 2009-2010

Table of contents

INTRODUCTION

During the recent times, the idea of Internet world has seen a revolution about content distribution. Beginning from the delivery of simple text contents like e-mail or transferring of files through FTP, Internet has moved towards new evolved applications, such as social networks like Facebook, virtual worlds, on-line games, blogs and video publishing and distribution systems.

Also the typical user has changed his behavior and his needs: he has passed from playing a passive role into the network, just downloading or browsing multimedia contents, to becoming an active producer and publisher of multimedia content.

A notable instance of this trend is the explosion of YouTube [1], the well known video streaming application that nowadays counts more than one million of video views per day.

In this dynamic environment, more and more peer-to-peer IPTV (Internet Protocol TeleVision) communities are growing. Usually, in this kind of networks there is one or more source hosts that provide a real-time video streaming to a multitude of recipients. Using an application with a so large number of involved nodes in a Client/Server architecture is unthinkable, because the server capabilities (memory, bandwidth, CPU) represents a scalability limit of the system. The development of IPTV is due to several factors: (i) the interest for the normal TV is constantly decreasing; (ii) by now the Internet network is considered like the main source of information for workers; (iii) users like the interaction with other users, so the idea of “community” is very diffused and appreciated [2].

The main topic of this thesis is analyzing the features of PPlive, a P2P live-streaming network, through the study of a quite big amount of traces captured during Michael Jackson funerals. Since this event has been very popular in network community (several tens of millions of people connected watching it), it

could be supposed that the obtained results should be significant and representative.
With more details, the analyses aim to characterize the network traffic, distinguishing between TCP and UDP traffic, generated and received traffic by peers (it means download or upload traffic), data (video) traffic or control traffic and intra-domain and inter-domain traffic. Other experiments regarding the port utilization, the population of the source channel and the balance between download and upload traffic, have been executed .

During this work, the attention has been pointed also on the presentation and explanation of CoreCast, a new network-layer protocol implemented for supporting and improving video-live streaming in p2p networks. CoreCast is build in top of the LISP, a protocol that implements the division between the location part and the identifier part of a IP address (Loc/ID split). So, in order to understand and check its performance, a comparison between intra-domain and inter-domain traffic between a CoreCast network and a “classical” p2p network has been made. It has been implemented using the information resulting from the analyzed traces, and also through the support of some analytical calculus.

The present work of thesis is structured as follows: in the first chapter the traditional delivery methods (i.e. unicasting, broadcasting and multicasting) are presented and compared; then, the peer-to-peer paradigm is introduced in the second chapter, in which some routing strategies in p2p environment can be found, too; the third chapter deals with the video delivery method in every transmission method described in chapters 1 and 2. LISP and CoreCast are presented and explained in the fourth chapter; afterwards, the work environment, the meaning and the structure of the analyses are showed in the fifth chapter; finally, the analyses and the results are showed and commented in the sixth chapter. All the conclusions are collected in a final proper section.

CHAPTER 1

Traditional Transmission Methods

Nowadays, networks world is in a constant change, because of the needs of the clients are in a constant evolution too. It is due to the technologic progress, that had allowed more resources to the computer machines (also to the domestic personal computers), that have started to ask for new services.

The great majority of these new services are based on multimedia data, such as audio, video, or audio and video combined together and many more. Beyond, the requests of these kind of services and applications are many and many, so some new methods to support the delivery of multimedia contents are necessary.

There are various approaches to implement the multi-user deliveries of multimedia data and they are different under many aspects, such as bandwidth consumption, routing algorithms, problems connected to sending a lot of data at the same time on the same communication channel.

Although in the recent time some new mechanisms have been implemented to better support the new clients' needs, the traditional transmission methods are *unicasting, broadcasting* and *multicasting*. These three kind of deliveries present a lot of differences, different advantages and different problems too.

By the way, using one of these methods in a "pure configuration" could not accomplish to satisfactory results.

The discussion starts from the simpler, the unicasting; it represents the most natural solution but also the most inefficient, because it keeps a lot of problems concerning *the emission by the source of a lot of packets*.
This limit represents a problem of scalability, because the amount of traffic on the network depends on the number of the recipients of the message. This is due to the fact that, in distributing multimedia to a large set of users, the source has to send a copy of the message for each receiver. For this reason, the source can became a bottleneck.

After the discussion about unicasting transmission, the chapter deals with multicast transmission and broadcast transmission too, two kinds of methods in which the problem of sending a lot of copies by source is avoided using a special address to reach more hosts.
Since the source has to send just one packet, on one hand the problem of scalability disappears, on other hand problems of duplication and delivery of packet come. As it will be shown later, multicasting needs some particular routing algorithms to implement the multiple deliveries.

1.1 Unicast transmission

There are various kinds of transmission in telecommunications world that depend on the principal aim of communication to be reached.
Network designers are now facing with the defiance of supporting the reliability of a lot of types of data delivery to any user, above of multimedia and real-time contents. [3].
Because of the requirement of supporting real-time data, the applications need to evolve from one-to-one communication to a one-to-many or better many-to-many communications. It’s necessary to make a distinction among the different types of

communication that can be found in a telecommunication network, in order to understand the advantages and the disadvantages of each of them.

The first distinction is between **Client/Server** paradigm to **Peer-to-Peer** paradigm:

- the former is the most classical type of communication. It’s used since the rising of internet network, and it is based on a subject who provides services (the server) to another subject that asks and uses these services (the client). This structure is very useful to control the traffic in the network as much as possible because, thanks to the servers it is possible to control everything that happens during communication. In others words, we have a centralized control system.

  But sometimes it’s not the best way to obtain good performance (about saving bandwidth, minimizing end-to-end delay, avoiding packet losses, etc…), because the QoS (Quality of Services) depends from server’s capabilities.

  Moreover control operations increase the necessary bandwidth to maintain the communications among the hosts [4].
- the latter is a new way to communicate in a telecommunication network and it based on just a type of subject: the peer. The peer can behave as a server, sharing contents or providing services, but it can behave also as a client, taking contents shared by others peers or using some services that others peers are offering to the network’s participants in that moment.

  Since it is a decentralized system, it’s very difficult to get the control of all network without centers that let know what’s happening in communication channels. Besides, the situation is more complicated because peers are free in joining or leaving the network in every moment, without giving some acknowledgments to the other hosts (these issues are discussed in detail in chapter 3).

  But this paradigm has also some advantages, first of all the independence between QoS provided by the network and hosts capabilities. It happens

because there are no servers, so the communication doesn't depend from the capabilities of any particular machine. In this way it's possible to reduce drastically the costs of powerful hardware necessary to the servers to control a huge amount of traffic with a lot of clients.
In the meantime, the absence of centralized point of control makes the workload among peers better distributed, avoiding network's collapses or possible creations of bottlenecks.

The unicast communication is referred above all to Client-Server paradigm, as a matter of fact it consists of a *one-to-one connection* between a client and a server. The client asks a server a service, and the server answers only to it.
There are two kinds of unicast:

- **single unicast**, referred to a communication between two nodes. It consists of a single path transmission or a multi-path transmission with the same source and destination nodes (that assures more reliability but increases the amount of traffic on the network, because several copies of the message are sent by different paths);
- **group unicast**, referred to simultaneous transmissions between a single source node and several destination nodes (obviously it consists in an additive bandwidth cost).

In both modalities, every delivery regards only two subjects. For this reason, it's very simple to control the only channel that has been created, to make possible the transmission of data between them.
On one side there is an absolute simplicity, a great deal of control of the data traffic and of the Qos given to clients in every moment, besides a high level of security in communication [3]. On the other side there is a consistent cost concerning bandwidth utilization: a reserved channel for every client asking for a

service is necessary. This means that the server should be able to manage a consistent amount of traffic (fig 1.1).

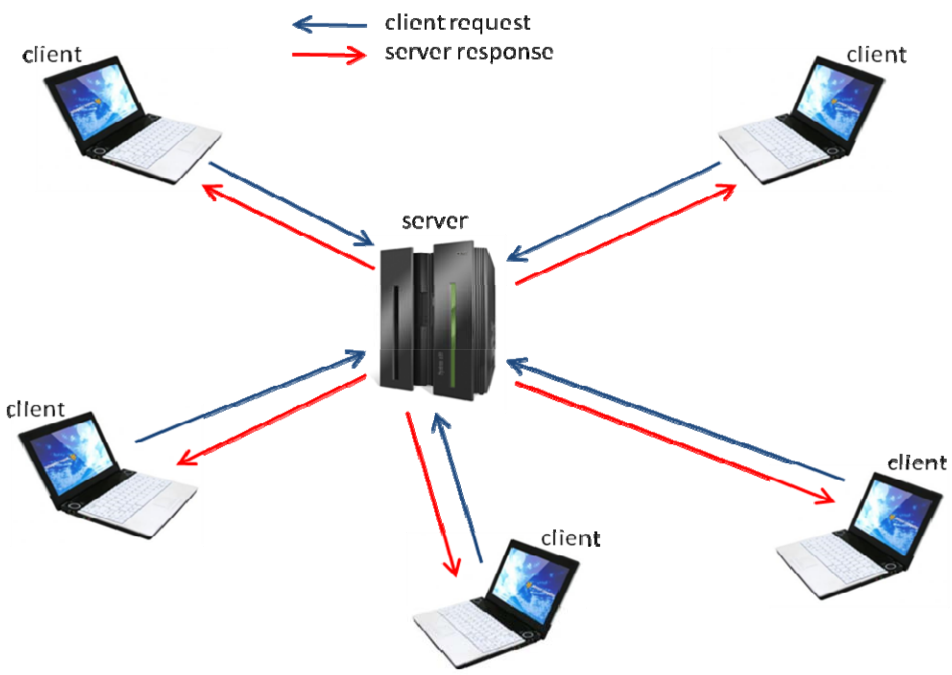


Fig. 1.1: Representation of flow of data in unicast transmissions between one server and more clients

It happens because unicast is based on TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) delivery methods, that are both end-to-end mechanisms [5]. As fig. 1.1 shows, it means that every time a client needs to ask for a service, it has to establish a session with the server exchanging some messages with it. Then the server will deliver to client a copy of the data requested.

The one-to-one sessions can offer a high level of management of the communication channel between the source and the receiver, allowing for transmission rate changes, acknowledgment of receipt, requests for data retransmission if an error occurs . But the server needs to copy the data packet and deliver it to each host who has required it [3]. It means that *the source has to emit*

a copy of the message for each request: under this process it could became a bottleneck of the network, with the consequence to risk to get network congestion. Of course this kind of transmission increases the utilization of bandwidth and in consequence the general traffic on the network, too.
So it can be drawn the conclusion that this approach is poorly scalable [4].

1.2 Broadcast transmission

As mentioned in the first section, the huge cost caused by using pure unicast transmission represents the most relevant problem to be resolved. It is due to the necessity to create a channel for each client asking for a service to the server. Under these conditions, the server becomes a bottleneck of the whole network, with negative consequences on the stability and the strength of the system.
For these reasons, new ways of transmission have been implemented, with the aim of bringing down the costs sending more messages in one time.

The more traditional way to send multiple messages using the same shared medium is the broadcasting [3]: with this protocol it’s possible to transmit one copy of a message to all network nodes, letting the recipients free to decide to take it or not. Distributing the effort of copying packets among the networks hosts consists in advantage for the source, because the effort is shared among more points of the network. So, the server doesn’t represent a bottleneck anymore.
For example, on an Ethernet LAN (Local Area Network), each computer has a hardware interface connected to the network by an Ethernet cable. Using this cable, each host can monitor the bus that connects it to all net, looking for some packets bearing a broadcast address. Then, the host will decide to receive or ignore the transmitted packets, according to its interest in carried contents. Proper

software mounted on each host has to take this decision, computing the packet or a part of it at least.
It plays the role of a communication filter mechanism (fig. 1.2).

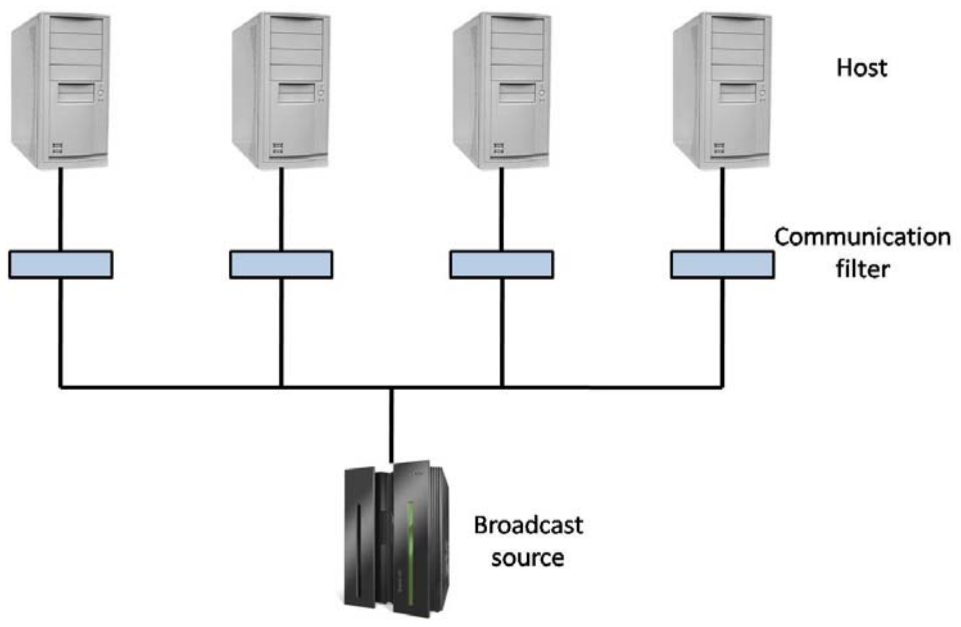


Fig. 1.2: Simple schema of a network using broadcast communication

The occupation of the bus during a broadcast call represents the most relevant problem of this communication modality, because the sending of a lot of broadcast messages might cause *network congestion*.
Using broadcast transmission only as long as generated traffic doesn't appropriate of a compromising portion of available bandwidth in shared channel is a good solution to avoid the congestion phenomena. Under this condition, communication filters should be considered a good compromise between using host resources and network ones [3].

On one side, broadcast avoids the problem of bottleneck represented by the server in unicast communications, but on the other side this traffic could be useless if only few hosts are interested in the unicast messages carried in that moment [4](wasting of network resources, as shown in fig 1.3).

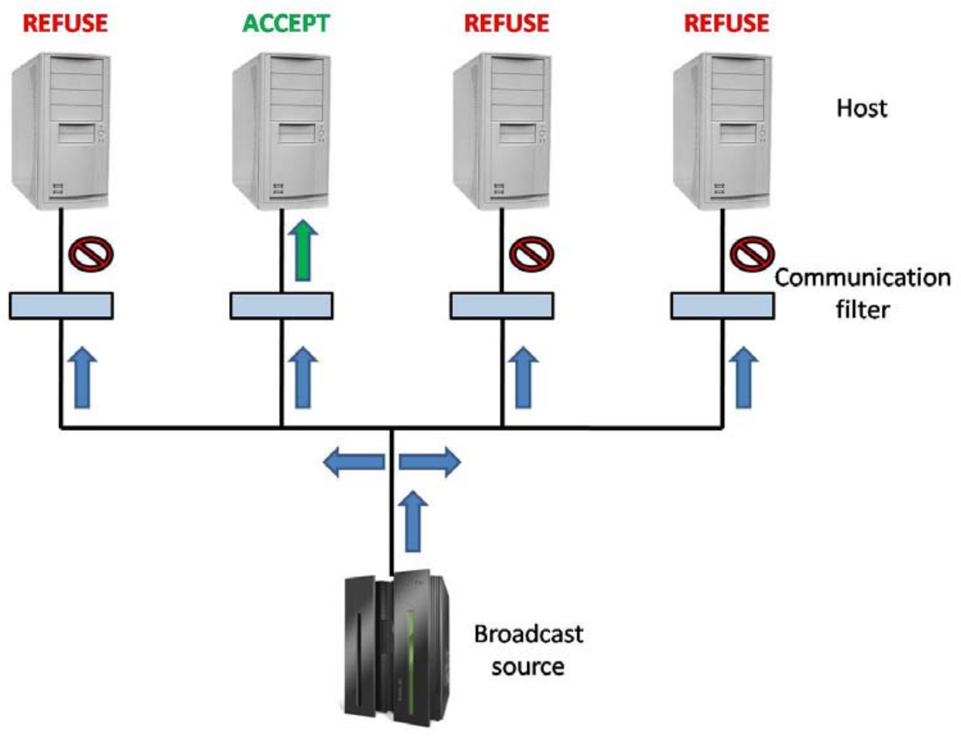


Fig. 1.3: Example of wasting of network resources (75% in this case) generated by a broadcast transmission

For this reason, the general trend is oriented to minimize broadcast traffic.
For example, in WANs (Wide Area Network), broadcast transmission is used for maintaining or diagnosing the state of the internetwork [3].

Finally, broadcast can be considered the opposite of unicast. In one hand, it resolves the problem of bottleneck represented by a server involved in a lot of unicast transmissions. As a matter of fact, using broadcast, the source is relieved from the task of duplicating any packet destined to multiple hosts: this job is transferred to the network devices (switches, routers, gateways), that duplicate the packets as needed to cover the network. But in other hand it creates some new problems.
First, broadcast traffic can quickly grow out of control in large scale networks and, in the worst case scenarios, it can shut down the network and **monopolize all of the available bandwidth** [3].

Second, the hosts are not always interested in receiving a broadcast message. In any case, their filter has to process the message, or at least a part of it, to understand if that message is useful or not for the host it belongs to. Of course this situation creates an **unwanted computation and useless traffic on the channel**.
For the reasons showed, broadcast transmission could not be considered the best way of communication in a network, despite it has some relevant uses in some situations.

1.3 Multicast transmission

Multicasting falls between unicasting and broadcasting, because rather than sending a message to a single host like unicast, or sending messages to each host like broadcast, it selects a group of hosts (**host group or multicast group**) and sends messages only to these hosts [3].
The host's grouping is the innovation key introduced by multicast transmission. Thinking of a LAN, each host's network interface accepts packets addressed to the multicast address corresponding to the host group it belongs to. Note that the group is identified by a *special single multicast address*.
On a WAN, the situation doesn't change much, it's just a little bit more complicated because of internetworking: here the host groups have to be maintained in the entire WAN, for this reason routers, too, are involved in joining and leaving host groups procedures.
Multicast system is very advantageous for several reasons:

- multicast group is composed by some hosts sharing some specific characteristics, for example simply the physical location into the network. In this way, sending a message to reach a particular kind of host becomes simpler: it just takes addressing the message to a single multicast address to reach the requested users;

- using a multicast address helps to avoid bottleneck formations, because in this way message sources don’t risk to go in overload (it could easily happen if only broadcast transmissions are used);
- avoiding pure flooding transmission, the wasting of bandwidth due to useless traffic can be reduced. Only some nodes (and it is supposed that they are the most interested in the transmitted contents, for the reason showed in the first point of the present list) are involved in the communication, so the risk of having network congestion is reduced [6].

A special class of IP addresses is reserved to reach every host belonging to the host group: class D. A generic IP address is composed of 2 parts: the first one represents the network ID, the second one represents the host ID.
Furthermore, a generic IP address is represented by 4 octets of numbers separated by dots (for example: 88.6.243.251). The first 5 bits of the first octet determine which the class the IP address is in (A, B, C, D or E). Since each class supports a different number of networks and hosts, the partitioning of net id and host id changes from class to class.

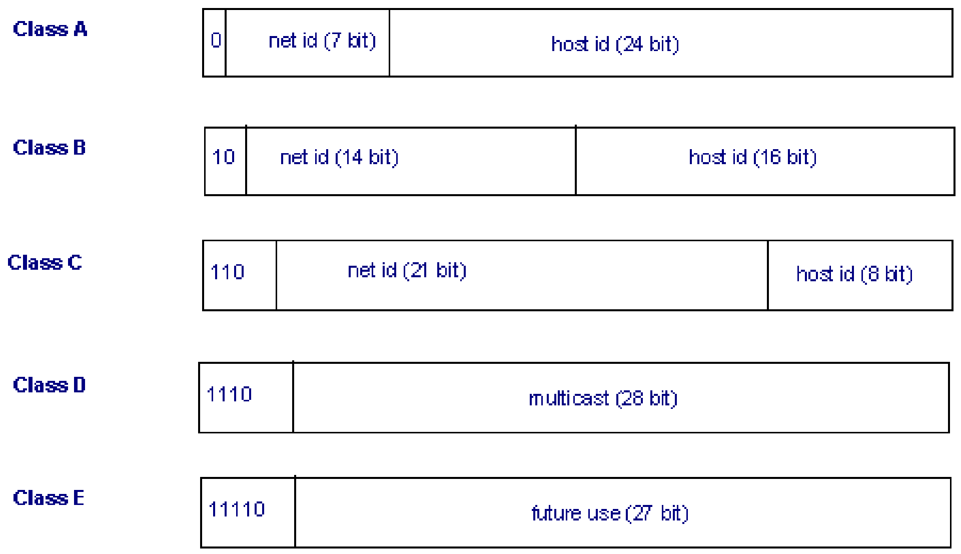


Fig. 1.4: IP address classes[7]

As figure 1.4 shows, class D addresses have no subnet mask, because all 28 bits are used to represent a host group, not a single host id. Thereby, in total there are 2^{28}= 268435456 different addresses of class D.

Another aspect of multicasting concerns the duplication of messages: *just one copy of the message crosses each link of the network*. Only when it reaches a router, some copies of it are made, and each of these ones is sent to a different subnetwork, started from a different network interface of the router (figure 1.5). This mechanism called **multicast delivery tree,** assures a consistent conservation of bandwidth, preventing loop formation, too [3].

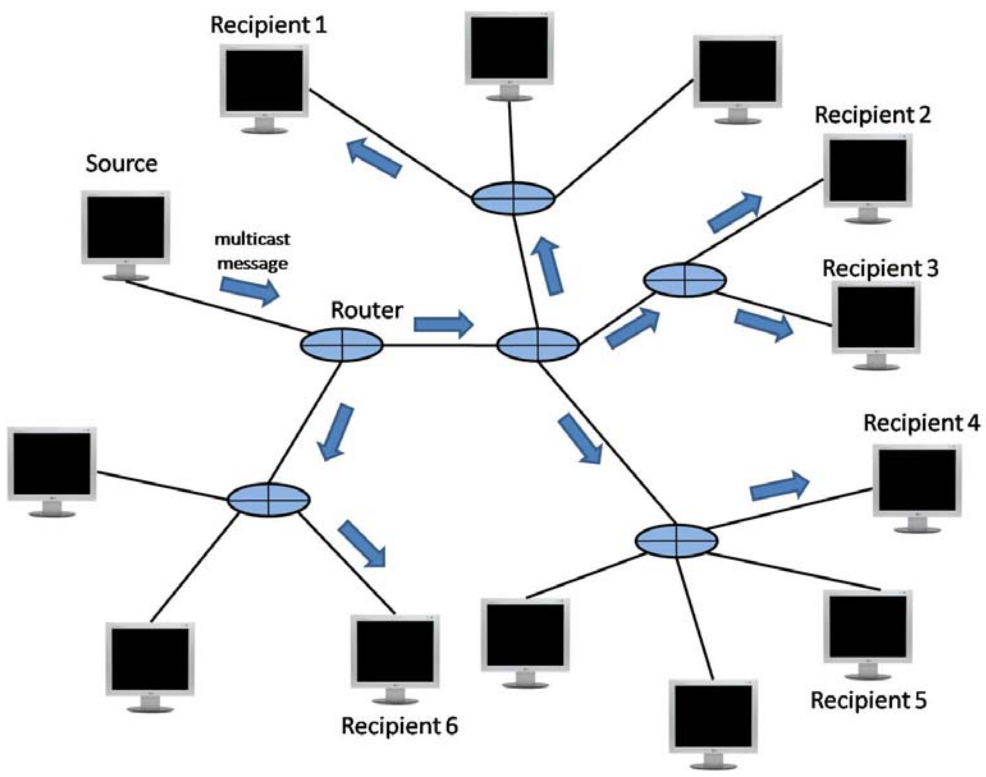


Fig. 1.5: Flow of data and message duplication in multicast delivery tree

Looking at the figure 1.5, it's easy to see that in the end there are only 6 copies of the unique original message created by the source.

In the case of unicast transmission, 6 copies of the message would be created by the source, and the routers would not duplicate anything, but just forward the packet until the recipients.
In the case of broadcast transmission in the same network of figure 1.5, it is necessary to create 12 copies of the message in total: 6 of these are useful (figure 1.6), the other~~s~~ 6 copies represent the wasting of the network bandwidth.

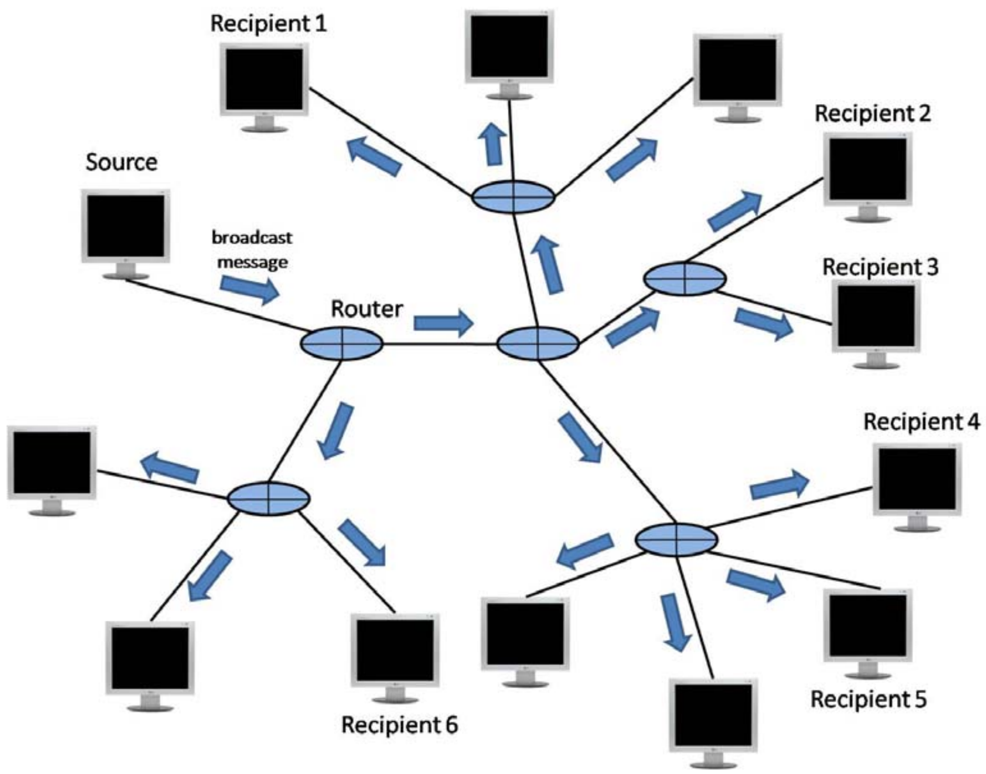


Fig. 1.6: Flow of data and message duplication in broadcast systems

Sometimes the source might want to restrict the distance of packet path: for example source could want the packet not to overcome its subnetwork.
To avoid that each packet crosses all network every time, a special value called **TTL (Time to Live)** is assigned to each IP packet. TTL is an integer number that represents the *maximum number of hops* that an IP datagram is allowed to

propagate. Each time a router forwards a packet, the TTL of the packet is decremented by 1: when TTL is 0, the packet is dropped [3][4].
Coming back at the previous example, the source has to fix TTL on 2 to maintain the packet inside the subnetwork only: in this way, when the packet reaches the first router, it is discarded without any notification error to the sender.

1.3.1 Multicast routing algorithms

Since in multicast there are some senders and a group of receivers forming a multicast group for each sender, designing effective routing protocols to determinate and maintain several paths becomes very important. Unlike unicast routers, multicast ones have to be aware of the data they exchange with each other. This information assures the forwarding of traffic in the network, and lets multicast routers have the *knowledge of the network topology*, too[3].
But there are also some additional issues derived from the dynamicity of the host groups: since every host can join or leave its host group at any time, the topology of a multicast network changes faster than a topology of a unicast or broadcast network [4][8].

For example in the context of broadcasting WAN network, there is no routing algorithm, the only method that exists is the **flooding**: when a router receives a multicast packet, it determines if it’s the first time the router receives a packet with that host group address (IP address of class D). If so, it floods the packet to the other interfaces (except the one the packet comes from, to avoid loops and wasting of network resources); if it is not the first time, it discards the packet (also this time to avoid network loops). Of course this is the simplest algorithm, but, as expected, it is also the most expensive one (it works like a broadcast routing algorithm, as figure 1.6 shows). As a matter of a fact, using a compare contents method result much more costly instead using the TTL.

On the contrary, in multicasting some routing algorithms appear. As already said, the fundamental task that a multicast routing protocol has to absolve is the creation of a **spanning tree of routers** as better as possible, because good or bad network performances of routing depend on this tree [3].
By the way, it's convenient to classify the routing algorithms according the method that they use to create these trees. So it's possible to divide these algorithms in three categories :

1. **simple-spanning tree**
2. **source based**
3. **shared tree**

1. The first one is the simplest kind of multicast routing protocol and it can be implemented using a spanning tree: the name suggests that this algorithm creates a tree of links among hosts, in which every pair of routers is connected by a single path. By definition *a tree doesn't contain loops*, and it is a good characteristic for routing, because it avoids cyclic recursions of packets in the network. Furthermore, since a tree minimizes the number of links between hosts, the number of copies of the multicast packets that each host has to make when it receives a multicast datagram is reduced. This algorithm is quite simple, but it can lead to concentrate the traffic among a small number of links, creating bottlenecks.
   It could be affirmed that the simple-spanning tree technique is very simple to apply, but it does not grant good performance in routing.

2. The second category mentioned implies *the construction of a delivery spanning tree for each potential source of the network.* Since in the same host group many potential sources could exist, the system constructs a different delivery tree rooted in each of these sources. Because of these algorithms are based on the router's knowledge of the shortest path back to the source, they are called **RPF algorithms (Reverse-Path Forwarding).**

They can be divided in three types:

- **reverse-path broadcasting (RPB)**: as long as a packet from a source arrives on a link that the local router believes to be on the shortest path back toward the packet's source, the router floods the packet on all of its interfaces (except the incoming one). If the packet does not come from the shortest path back toward its source, it is discarded.
  The information about the various paths is maintained in a *routing table*. Because of multicast group memberships are not took into account during the construction of the delivery tree building, packets could be forwarded onto subnetworks that have no members in the destination host group. This behavior represents the major limitation of RPB.
- **truncated reverse-path broadcasting (TRPB)**: this algorithm uses the information about memberships to trunk the subnetworks from the tree, if there are no group members in them, with the aim to overcame the limit of RPB during the delivering phase. TRPB partially resolves the problem of RPB, because during the building phase it not even takes into account group membership, but it works later, during the delivery phase.
- **reverse-path multicasting (RPM)**: this algorithm use TRPB and a *pruning technique* together. When TRPB finds out a leaf of spanning tree that represents a subnetwork without any recipient of multicast packet, a message of pruning is sent to the router. Then, the router prunes those leaves from its memorized spanning tree.

These methods are relative simple to implement but they generate more complicated and big routing tables. Moreover, they are not scalable and they waste bandwidth, especially if there are a lot of recipients at the edges of the delivery spanning tree.

3. Rather than building a source-based spanning tree for each potential source of the network, *a single delivery tree is shared among all member of a multicast*

group. Under these conditions, a different shared tree is defined for each multicast group: so multicast traffic is sent and received over the same spanning tree, regardless of the source. To manage the routing, a **Rendezvous Router** is elected. The others routers encapsulate the multicast message in an unicast message and send it to the Rendezvous Router, that provides the multicast routing.

Since just one router is enough to maintain the information about one group, shared-tree algorithms use resources in a more efficient way then the source-based ones.

Besides, the number of the sources doesn't represent a problem anymore, consequently scalability problems are resolved. But on the other hand, the Rendezvous Routers represent critical points of the network, in others words bottlenecks.

In details, a rendezvous router or **Rendezvous Point (RP)** acts as a common point among different autonomous system. Its task consists in allowing multicasting across several networks. The mean aim is supporting routing of inter-domain traffic, overstepping the intra-domain traffic limit of the normal implementation of multicast. Referring to a multicast shared tree, the RP represents the shared root [9].

There are several definition of an Rendezvous Point:

- **static RP**, that is the simplest method. It consists in a static configuration of the RP for a multicast group range. Every router that point to the RP into the spanning tree is configured with the static RP address. Static RP may be an attractive option if the network is quite small and doesn't change very often. On the contrary, in bigger networks it becomes too expensive, because if RP changes its address, each router of the network has to be reconfigured.
- **BSR (Bootstrap Router)**, an elected router that chooses the RP collecting information from some candidate RPs. It sends some BSR messages, then

the candidate RPs answer through a candidate-RP advertisement: these messages let the BSR elect the RP. BSR method is selfconfiguring and robust, so it works better than static RP in wide networks.

- **Auto-RP**, consisting in an automatic method to elect the RP, based on an exchange of messages between the candidate-RP and the RP mapping agents. A reserved multicast address (224.0.1.39) is used in order to distribute automatically the information about the current RP address: all network routers just have to join this multicast group, and copy the RP address into their cache. Auto-RP and BSR can operate together in the same network.
- **Anycast-RP**, that gives the same RP address to two or more candidate-RP, creating redundancy. The aim is improving the fault tolerance of the RP, because in this scenario if an RP fails, the other RP-candidate with the same RP address is activated in substitution.
- **Phantom-RP**, in which no physical allocation is assigned to the RP-address. As a consequence, the RP address could be an address in a subnetwork, without a location in a router interface.
- **Embedded-RP**, that works very similarly to the Auto-RP, but it uses IPv6 addresses. Even using this method, routers don’t need to be reconfigured, they has just to join a reserved IPv6 multicast group to catch automatically all RP information.

Among these solution, of course static RP is the worst, but it is convenient in small scenarios. Anyway, the most attractive method is the Embedded-RP, because it is the only one that makes use of the advantages carried by IPv6 address technology.

CHAPTER 2

Peer-to-Peer Networks

The previous chapter deals with the traditional approaches concerning the delivery of data from the source to the recipient.

The first method analyzed is unicasting, in which there are *one-to-one communications* between the source and the various clients. According to this protocol, the source has to produce a copy of the packet for each received request, then it instantiates a reserved communication channel to permit the delivery of the message. As showed in chapter 1, this situations ends in several problems, such as wasting of bandwidth, formation of bottlenecks and poor scalability.

Instead, in broadcasting and in multicasting *one-to-many communications session* let to reach several receivers using the same session: in this case, the source sends just one packet, then the routers will copy the packet according to the routing policy and the physical addresses locations of the addressees. Although these approaches avoid the problem of bottleneck, they result too much expansive (above all broadcasting), because a large amount of traffic could be created to control the data flow.

Because of the several problems of the traditional transmission modalities, based on the well-known Client/Server paradigm, new approaches, based on P2P (Peer-to-Peer) paradigm, have been implemented. The strongest point of p2p networks is their distributed nature: it can help to reduce bottlenecks formation (like in unicasting) and wasting of resources (like in broadcasting and multicasting).

2.1 Introduction to Peer-to-Peer world

As said in the paragraph 1.1, there are two paradigms that characterize communication between two or more subjects. The more traditional communication modality is the well-known **Client/Server**, in which there are two distinct entities:

- the Client, that asks for services from the network;
- the Server, that uses its resources to provide services to the network.

Because of the sharp-cut distinction of roles, the servers take charge of central points of the network, as a matter of fact network performances depend from them. Because of this aspect, Client/Server are defined as *centralized* systems: it’s a good way to maintain the total control of the network traffic, but, as it can be easily expected, this situation brings *scalability* problems [3].
This problem appears when the number of the clients becomes much bigger than the number of servers, that fall in overload and stop providing their services. For this reason, the growing of the clients has to take into account the number of servers, otherwise problem of network congestion could happen.

Thinking that the number of clients is usually much bigger than the number of servers, can give an idea about the *fault tolerance* of the system: a server breakdown constitutes a resource loss for the network, so fewer services for clients. Besides, *servers hardware has to be very powerful* to manage a huge quantity of requests, and it represents a relevant cost, of course.
The servers actually represent bottlenecks for the whole network, i.e. points of vulnerability that became more critical as the number of clients grows.

Established these issues, researches have moved from centralized systems towards *decentralized* ones, to avoid the dependence of the network from few critical hosts.
Peer-to-peer paradigm is based on this idea, in fact *it has no roles division*: there is just one subject, the **peer**, who can play like a server, sharing resources and

giving services to others peers, or like a client, asking for services from other peers, that act as servers in that given moment [3].

This new way of thinking to a network gives some advantages, first of all the independence from the resources of the servers machines. The network does not present point of vulnerability, because the resources are more distributed along the network.

For instance, if a certain peer is not yet able to provide a service, the same service could be easily disbursed by another peer [4]. *So, communications without a central point of control are allowed, creating direct message interchanges directly between the end-systems.*

The source is not fixed like in multicasting [3], but it becomes dynamic. This change improves the fault tolerance: changing of source-peer is enough to avoid losing of network resources and services. Besides, mounting of very expensive hardware on the servers machines to support huge amount of requests is not necessary yet, because there isn't any point of centralization of the requests.

As a consequence, the risk of overloading is drastically reduced (moreover nowadays, more and more peers are equipped with very powerful resources, in terms of CPU, RAM, memory devices, network interfaces, etc…, that allow them connecting to the network through wide-large bandwidth).

The probability to find the same service provided by several peers is so high because this sort of network might be constituted of thousands of peers.

It is possible because p2p (peer-to-peer) networks don't suffer of scalability problems: the entering of a new peer into the network doesn't represent a threat to the stability of the system, on the contrary it contributes to add new resources to share among peers [4].

But this characteristic has its limits, because if the number of peers increase too much, the situation could get worse. Essentially, it is due to the limited upload capacity of some of the peers, that usually join the network using an ADSL (Asymmetric Digital Subscriber Line) connection [10]: it does not result efficient in supporting some particular data transmission, such as real-time multimedia.

Another problem of p2p networks is connected with the idea that they are a sort of *shared environment* in which several kind of independent applications, with different needs and resource's requests, are executed simultaneously. In this situation several applications could reciprocally interfere, or a "resource race" could have place [4].
A typical bad situation of concurrency between application is the *WAR (Write After Read) error*. For example, if there is a file accessible in write and read modality, it could happen that some users accede (in read access modality) to an old version of the file, because in the same breath other peers are modifying it (write access modality) and file updating is not been done yet.

Another issue of p2p network is its *incontrollable dynamicity*. It refers to the fact that a peer can join or leave the network in every moment, without signalizing or giving any acknowledgments to the other peers it was communicating with. Because of these problems, constant and good quality performances are not guaranteed [4][10].

2.2 Routing strategies in P2P networks

Peer-to-peer networks have a lot of advantages that can be taken into account in order to transmit very requested and popular contents. The most powerful advantage that p2p can give in improvement of the network performances is the possibility of *using direct communications between end-systems* [4].
This new scenario causes a reduction of vulnerability of the network and better possibilities of scalability, but in the same breath the general traffic on the network increases.

Generally, the raise of traffic in a peer-to-peer network is due to the using of **flooding** delivery method. This is because in a p2p environment, since there are no central control nodes, sending packets in all directions to find the wanted resource, rather than using a routing algorithm aimed at a certain peer, could be simpler and faster.

Because of flooding, peers should ask a lot of others peers to have more possibilities of finding the contents that they are looking for. In others words, *the probability to find the requested information is proportional to the number of peers involved into the search.* As a consequence, a large amount of traffic is created.

In order to solve this problem, sometimes it's preferable to renounce to a pure peer-to-peer configuration by the introduction of some centralized nodes in order to better control the traffic. In these "hybrid systems" there are some elected peers that assume the role of group managers. This solution could surely help the improvement of the routing, avoiding the brutal method of flooding, but in the same breath it implies the coming back to the Client/Server model, facing against its problems again.

Hence, it's better finding other solutions in order to organize the routing in p2p environment in a more efficient way.

The first issue of p2p system is the random way used for the creation of groups of users. For instance, in multicasting a group of nodes is created according to receiver's subscription [3]. Using this received oriented approach, controlling the transmission among the hosts results simple, so there are no big problems in managing the group.

But in peer-to-peer contest it is impossible to decide how the groups have to be created, because there are no central entities to control them. So, groups of peers are often created randomly [11]. This situation is not preferable, because groups created in this way might have some problems of:

- latency, because messages should go through lots of middle peers before they reach the recipient. Since these peers could not be members of the sender's group, so could not be the actual recipients, this mechanism generates a big wasting of time;

- performance is degraded by slow peers that the packet might eventually meet along its path;
- partial employment of peers bandwidth is used to shunt the traffic generated by others pees, in order to allow the message to end in the true addressee.

Therefore, the most congenial solution is trying to use peer-to-peer advantages (above all potential scalability) in combination with a policy aimed to manage groups in a more efficient way. The idea is avoiding flooding and random group building.

Publish/Subscribe[11] is a well adaptable model to peer-to-peer framework, because it is characterized by two subjects:

- subscriber, that catches a message
- publisher, that publishes a message

At first view, Publish/Subscribe concept looks similar to Client/Server one, but there is a substantial difference between them: the former allows every peer joining one or more communication groups as publisher or subscriber, so the role of the peer can be changed whenever it wants; on the contrary, the latter does not allow peers to change their roles, so if one node is a client, it cannot become a server and vice versa.

Because of its flexibility, Publish/Subscribe model is well adapted to dynamicity of p2p world. Furthermore, using JXTA[12] protocol, it is possible to realize data sessions among different communication peer groups.

JXTA realizes an overlay network to maximize peers potentiality, also allowing communication groups interactions.

These overlay structures are called **virtual self-organizing overlay subnetworks**[12]:

- *virtual*, because they imply virtual relationships between nodes. So there are not any change in the physical layer;

- *self-organizing*, because each environment can organize itself according to the resources that the participants have and share;
- *overlay*, because a peer could participate to more than one group in the same time, so these subnetworks can be intersected;
- *subnetwork*, because they are constituted of peer participants in a communication group that, actually, could be interpreted like a subnetwork in application layer contest.

An example of virtual self-organizing overlay subnetwork is shown in following figure 2.1, where it is composed by highlighted nodes and links.

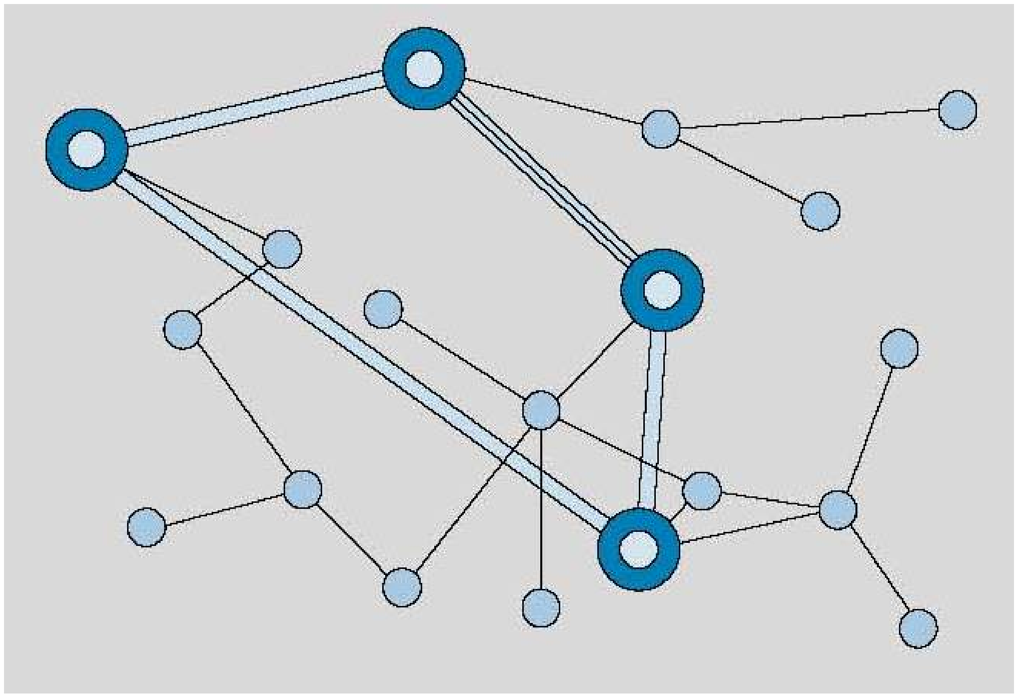


Fig. 2.1: A virtual overlay subnetwork in a Peer-to-peer network

Adopting this structure it is possible to obtain also interesting results about QoS parameters, because its transmission among group's peers can be controlled in a better way. Under these conditions, guarantying a certain delay threshold or limit the bandwidth consumption become achievable.

The realization of a structure with these characters could be implemented through a **multiring topology** [11], as figure 2.2 shows.

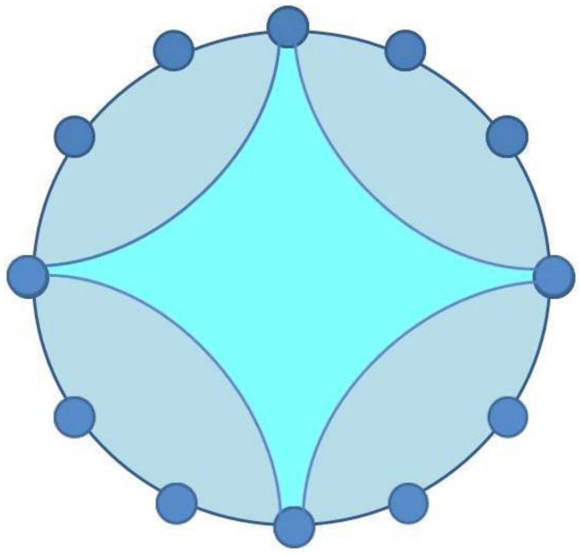


Fig. 2.2: Scheme of a multiring topology

The choice of this topology is justified by several factors, first of all the scalability: since *each node is connected just with two neighbors* (in the general hypothesis of only one ring), its workload is independent by the total number of peers. Furthermore, this characteristic makes easier to manage the traffic in a decentralized environment too, without the supporting of any central entity.

If a greater reliability is requested, additional internal rings might be built as **optional or backup paths**. This expedient helps also in reducing the latency, because following these alternative links, some peers can be reached in less hops. As a consequent, backup path results faster than the external ring.

Building additional rings means also *increasing the workload of interconnected peers*, because more than 2 links will be connected to them. That is the reason why the peers equipped with the most powerful hardware are located in these strategic points, in order to avoid collapses of the network and situations of imbalance.

Another advantage of multiring topology concerns the delivery. In this situation, sending a message to all group's members means just sending two messages to the two neighbors. Afterwards each node receives the message and puts it in its internal FIFO (First In First Out) queue, until a sender process forwards the messages to the next neighbor. If a node receives the same message twice, it simply ignores the message (of course a control on message ID is requested).

Even if on one side the introduction of backup rings increases network stability, on the other side it increases also the general amount of traffic, because one message might be received a lot of times by the same node.

To avoid this situation, **dual mode link** [11] has been implemented. The idea consists in dividing the nodes in two modalities:

- *primary link modality*, which allows immediately the sending of messages on the link;
- *secondary link modality*, which does not allow the sending of messages on the link immediately , but a previous announcement is necessary. Then, only if the recipient side is interested in receiving the message, the delivery will be finalized, otherwise no sending will have place.

Dual mode link creates an implicit process, in which the best paths to connect two or more peers are dynamically built. The system behaves as a network that adapts its topology to the ongoing message flow. For this reason, the dual mode link mechanism is also called **implicit dynamic routing**.

CHAPTER 3

Video Delivery

In the previous two chapters, traditional transmission delivery methods and delivery methods in peer-to-peer networks have been studied. Now the reader might have a general idea about the mechanisms of all these kind of transmission, about their strong points and their weak spots too.

Usually, the video delivery deals with scenarios in which there is one source that spreads a video content towards a large group of receivers. Very often, a lot of simultaneous sessions, which have the same source or different ones, are established. This is due to the possibility of having overlapped groups, because a host may belong to many groups in the same breath.

This chapter deals with some video delivery methods, comparing the situation in the various delivery transmission methods' scenarios. Video delivery is so important nowadays because the users' interest is growing more and more in this direction.

The fast growth of broadband connections to the Internet network contributes to the spreading of this phenomena, because the diffusion of large-band connections allow more and more users to use multimedia contents and services, like VoD (Video on Demand), streaming live video of some events, videoconferences, Internet radio, etc…

3.1 Video on demand service

As pointed out in the first paragraph, the unicast transmission mode is not very efficient when the number of services requests increases: one copy of the contents for each session that must be established with each recipient that asked for it is necessary. Furthermore, the server hardware has to be very powerful, and of course costs grow up [3].
There is another aspect that must be taken into account: the content type.
It's easy to understand that users' needs change in concomitance with the kind of data that have to be managed in the established channel. Of course, it's necessary to get *more bandwidth and less delay* (and delay jitter, too) to transmit a video (in streaming or a preregistered one) than to send an e-mail or a simple text file.
For this reason, using an unicast transmission to transfer video in a network could be not very efficient, as some researches about the using of unicast transmission in VoD services have shown.

First of all, it is rightful to briefly explain how a VoD system works. In this kind of system there are some video contents that can be viewed from clients under their specific request. The source establishes an unicast channel for each client who asks a video, and then starts to send it video packets. This mechanism, known as **TVoD** [13] (True VoD), represents high cost for its users, because the server that holds video contents should have a very powerful hardware to support a large amount of communications (for this reason it represents also a bottleneck for the whole network).
On the contrary, there is another method called **NVoD** [13] (Near VoD) using multicast channels: the video is periodically sent without taking care about the clients requests. This system is cheaper than the first one, because it spends less bandwidth, but users should wait for same time before the system allows them the video download.

A right compromise between the two mechanisms is represented by **UVoD** (Unified VoD), a system that uses both unicast and multicast channels.

Multicast ones periodically (for example at pre-established time instants t_m) transmit the video content, regardless of the number of user requests. Moreover, each channel stays alive until video end.

Assumed that:

- M represents the number of videos,
- N_M is the number of multicast channels,
- N_U is the number of unicast channels,
- L is the duration of video in time units,

the quantity $\frac{N_M}{M}$ represents the number of multicast channels allocated per video object, and the total number of channel is simply the sum $N = N_M + N_U$.

Finally, the *offset between two adjacent multicast channels* can be calculated as follows:

$$T = \frac{L}{\left\lceil {N_M}/{M} \right\rceil} \tag{1}$$

The requests collected during the period T will be served by unicast channels, that, as said before, represent an high cost in terms of bandwidth.

A predetermined *waiting time δ* has been established, To avoid that this cost grows up too much. It's calculated as shown here:

$$(t_m - t) \leq \delta \tag{2}$$

where the parameter t represents the time instant when a request is received.

As UVoD uses unicast and multicast channels, it is demonstrated [13] that an optimal number of multicast channels is obtained with the following formula:

$$N_M^{opt} = \left(\frac{LN}{2LN - \delta N} \right) \frac{N}{M} \tag{3}$$

Utilizing this formula, it's possible to obtain the number of multicast channels to avoid waiting for a video, considering that the parameters N and M are very huge to be approximated to the infinite (i.e. $N \rightarrow \infty$ and $M \rightarrow \infty$) and imposing $\delta = 0$ in (3):

$$N_M^{opt} = 0,5 \qquad \Rightarrow \qquad N_M = N_U \tag{4}$$

It means that the number of unicast channels has to be the same of the multicast ones, to avoid delay in serving the requests: it's easy to understand that this would imply an exorbitant cost to admit.

Besides, this formula (3) doesn't take into account the request arrival time rate (from now on this parameter will be referred to as λ), and it is wrong because this is a crucial element in a VoD system.

Because of this mistake, **enhanced UVoD** [13] has been implemented, it consists of a new UVoD architecture taking in account of request arrival time. In enhanced UVoD there are also multicast and unicast channels, but the second ones are active only when there are no multicast channels available: this policy aims to minimize the unicast utilization to save bandwidth.

In particular, considering that:

- requests of video data come at arrival time instants $t_1, t_2, t_3, t_4 \dots t_k$;
- $t_1, t_2 < t_3 < t_4 \dots < t_k$;
- $t_i \ \forall\, i$ are independent events;

it's possible to approximate the requests arrival process with a stochastic Poisson process.

According to these hypothesis, if the parameter λ becomes too high, the maximum number of unicast channels can be approximated, as showed in formula (5):

$$N_U^{max} = \frac{T - \delta}{\delta} \tag{5}$$

This is the best worst scenario, because it has been supposed that the request arrival time rate is enough to consider true this approximation.

Now, using the formula (1), T can be substituted in (5) to obtain the following result:

$$N_U^{max} = \frac{L\,M - \delta\,N_M}{\delta\,N_M} \tag{6}$$

As before said, when some requests arrive during the time interval T, an unicast channel will satisfy it in the waiting time δ imposed, before the following multicast channel is activated. Supposed that in the system there are K not overlapped δ-windows it can be deduced that:

- because of the δ-windows are disjoining, and for every one of them just one unicast channel works, K also represents the total number of unicast channels used in the whole network;
- the number of requests stored up during a single interval time T is represented by the quantity $\lambda(T-\delta)$;
- the amount of requests in a single δ-windows is $\delta\lambda$;
- the total number of requests in the network is $\lambda(T-\delta) - K$.

Therefore there are two different ways to express the same quantity, than following equation can be imposed:

$$K\,\delta\,\lambda = \lambda(T-\delta) - K$$

now using the (5) and the (6) the value of K (see formula (7)) that also represents the number of unicast channels can be extracted:

$$N_U = K = \frac{\lambda(L\,M - \delta\,N_M)}{N_M(\lambda\delta + 1)} \tag{7}$$

As shown by comparing this formula with the (3), *the number of unicast channel also depends on the request arrival rate*, and it is more correct because this parameter plays a fundamental role in a VoD system [13].

Anyhow, N_U is directly proportional to L and M. This means that if long videos have to be sent in a network with a consistent number of users, lots of unicast channels are necessary. Accordingly, it represents a too high cost.

For these reason, using a unicast transmission in a VoD network is not efficient, and it can be easily deduced that the situation gets worse passing to a video-live transmission network. This kind of network needs more restrictive parameters, such as a large bandwidth, a low delay and delay jitter, and these needs render unadvisable the submitting to a simple unicast transmission mode [3][13].

3.2 Unicast's limitations

As underlined more and more times during this chapter, unicast transmission mode has some limits when it is used in video data system, like VoD or UVoD [13].

Scalability is the most visible problem of this type of transmission channel, because it's necessary one channel for each client asking for a data (at least one channel for every δ-window, as shown in enhanced UVoD systems [13]) under these conditions. When these clients became too many, the amount of traffic in network speedy increases.

The problems grow even more when video is the data type to transmit, because the volume of data (the situation worsens when video length increases) influences negatively the number of channels required to support all the incoming requests (referred to parameter L in formula (7)).

Since in real-time video transfer networks there are also more strict constraints to respect, using an unicast modality of transmission could be very inefficient to get an acceptable quality and fluency of video streaming [3]. For example, video streams in IPTV (Internet Protocol TeleVision) [14] are first encapsulated in IP packets, and after distributed by IP unicast and multicast. This delivery method clashes with the classical broadcast system in cable and satellite TV.

Savings gotten in substitution of broadcast modality with an unicast-multicast one justifies the choice taken by these researchers.

By the way, this system uses unicast channels just when a user switch from a multicast channel to another one, because unicast transmission is faster but more expensive too.

It's evident that the use of multicast communication is minimized in this situation, too, just because of its too high cost.

Nowadays, pure unicast system is not a good solution to support the constant growth of real-time services offered by the network. Nevertheless, it could be a consistent help to optimize performances of multicast transmission, because using multicast and unicast in a conjunct way, could turn out results more efficient and cheaper than using simple unicast (also than the traditional broadcast).

3.3 Multicast in video live transmission

A lot of applications have already been developed to take advantage of IP multicasting technology. For example, a streaming multimedia contents, like audio, video or combined audio-video, can be sent on a multicast network using the routing algorithms examined in the previous paragraph.
The following table shows which are the most diffused categories of applications that can use multicast transmission:

	Real Time	Non-Real Time
Multimedia	Video server Videoconferencing Internet audio Graphics and audio	Video and Web services Content delivery Intranet Internet
Data-only	Stock quotes News feeds Whiteboarding Interactive gaming	Information Delivery Database replication Software distribution

Fig. 3.1: Classification of multicast applications [3]

Another example of the multicasting using is given by **pushing technologies**, i.e. systems in which the request of a transaction is started by the publisher (the server), rather than the subscribers (clients). The transported data could be news, stock quotes or, in general, some information of common interest for a large number of people. Even if in the past these systems used unicast transmission, using also proxy and caching servers to improve workload distribution, nowadays they use multicast, above all in real-time push systems [3].

Since the use of interactive and multimedia applications has become large spread, today's networks have rapidly evolved to support the transmission of these contents.

A **multimedia** can be defined as a combination of more kinds of digital contents, such as video, audio, text, pictures and graphics. But, to define a data like multimedia, at least two of these forms are enough.

Because of ever-increasing number of users interested in multimedia, a wide variety of multimedia applications has been diffused: crossing from unicasting to multicasting had place just to support this trend in a better way.

The figure 3.2 represents the most diffused categories of multimedia applications:

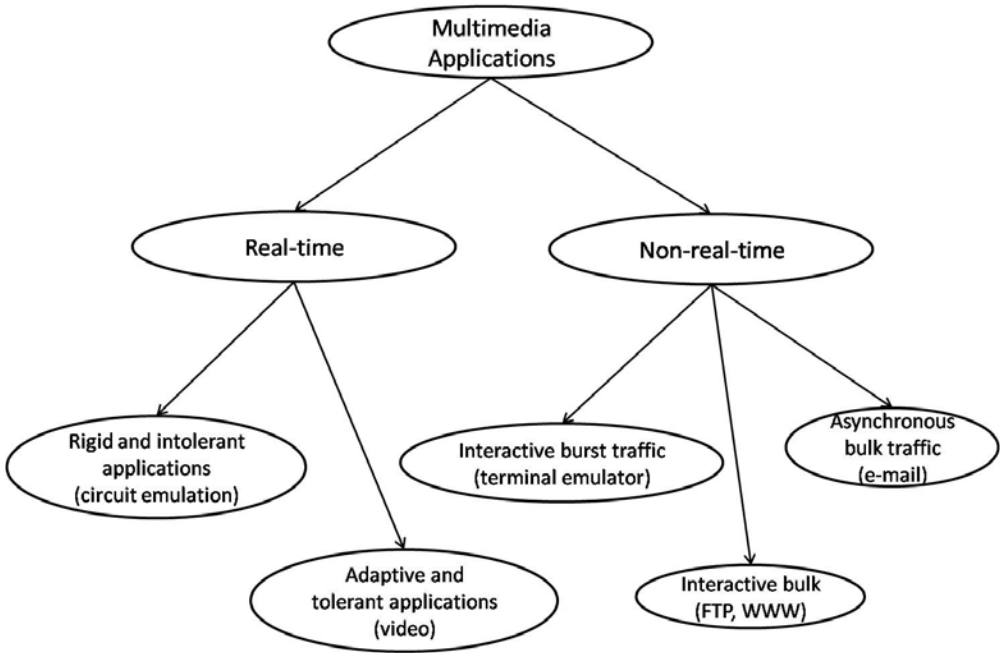

Multimedia Applications
Real-time
Non-real-time
Rigid and intolerant applications (circuit emulation)
Adaptive and tolerant applications (video)
Interactive burst traffic (terminal emulator)
Interactive bulk (FTP, WWW)
Asynchronous bulk traffic (e-mail)

Fig. 3.2: Categories of multimedia applications[3]

Multimedia data need more strict constraints to be sent, especially if they are real-time, too (on top-left side of figure 3.1): apart from bandwidth, delay, jitter delay and packet loss rate assume a key importance in this scenario.

For instance, the well known YouTube is a pull video streaming system in which there are more than one million of video views per day. Founded in 2005 by Chad Hurley, Steve Chad and Jared Karim, nowadays it has became the leader in the online video field, as a matter of a fact it represents the biggest video sharing platform of all Internet.

YouTube allows people to upload and share video clips in several ways, such as mobile devices, blogs, e-mail or simply websites. It uses the Adobe Flash technology in order to render its video contents: video are converted and visualized in *flv* format. The key of its success and incredible grow is its simplicity and intuitively [1].

Because of the dynamicity of multicast groups, several messages to exchange information about network conditions are necessary to assure a certain level of performances in transmission of multimedia contents.

Multicast session usually relies on the RPT/RTCP protocol:

- **RTP** (Real Time Protocol) flows carry multimedia data, so they represent data traffic;
- **RTCP** (Real Time Control Protocol) flows carry signaling and synchronization data creating control traffic. Inside these messages, information about network conditions (ratios such as network packet loss rate, available bandwidth, maximum latency, variance of maximum latency) are encapsulated.

  There are two kinds of messages: **SR** (Sender Report), carrying feedbacks from source to receiver, and vice versa **RR** (Receiver Report), carrying feedbacks from receivers to the source. These reports are necessary to let the source adapt its output rate to the network conditions (for example based on these feedbacks, source can decide if it is opportune sending additional enhancement layers of the video or not)[8].

Among the mentioned ratios, the most critical one is the available bandwidth. Although new technologies, like Gigabit Ethernet, might resolve the bandwidth problems, providing bigger channels to the clients, anyhow the situation about network latency and jitter delay is still complicated to solve. Real-time data requires also some type of bandwidth reservations based on **QoS parameters**, as well as priorities.

For all these matters, the traditional best effort service provided by IP protocol is incompatible with multimedia data, above all if a live video streaming service is wanted. As a matter of facts, this particular kind of service requires specific QoS parameters, such as to limit the bandwidth between 128 Kbps and 1 Mbps [3].

But even in a **VoD transmission**, some particular QoS parameters are required: here the quality of the video also depends on its popularity, as well as from the number of users that chooses viewing the same video in the same time. Some evaluations and measurements made on the occasion of the 2008 Beijing

Olympics games, show that user behavior plays a crucial role in establishing the quality of delivered videos [15].
For example, the analysis indicates that 80% of the users only watch the first 10 minutes of a video, regardless of the video duration. It means that caching the initial segments of the video, instead of the whole one, can optimize the utilization of memory resources.
As often as not, VoD services allow users to utilize full streaming functionalities, such as pause, seek, rewind etc… The same researchers indicate that only few users actually use these functions. It implies that simpler delivery modalities on VoD systems can be implemented, decreasing the computational cost of the algorithms and maintaining still high the user satisfaction in the same time.
Since the evaluations regards the Olympics games, that represents one of the worst-case scenario for VoD system (because it’s live, event-driven, large-scale and long in duration), better performances are expected in normal conditions.
By the way, in VoD or in live streaming video, there are usually more than one recipient. So, efficient delivery methods are indispensable to avoid overloading network bandwidth: for this reason, IP multicasting has been largely utilized to optimize multiple deliveries [3].

Combining multicasting with QoS based resource reservation strategies could help in getting much better video live streaming results. But it’s not enough yet, because there are some problems about using multicasting, bound to the necessity to construct a spanning tree referring to all recipients of an host group (shared tree algorithms [3]).
It is a problem, because to allow the routers to get the knowledge to build a spanning tree like this, a lot of messages between routers have to be sent, risking an overload of the network.
Because of new multicasting IP-based multimedia services often don’t require many-to-many communication offered by the well-known ASM (Any Source Multicast), a one-to-many approach is preferred. So **SSM** [8] (Single Source Multicast) has been implemented to support this new trend. Usually, one source

has many sessions with many receivers, therefore it supposed that SSM is the best method to provide all kinds of IP-based multimedia sessions one-to-many, such as streaming IPTV.
One of the most relevant advantages offered by SSM regards the implementation of the spanning tree: *having only one source makes simpler to build a tree, because it will have a unique root*. Another benefit is the achievement of a better control of the established sessions. For example, localizing distribution problem of particular hosts is simpler, because crossing a one-root spanning tree is faster than crossing a multi-root spanning tree.

3.4 Multicast's limitations

Drawing a conclusion about multicast, it's certain that this protocol faces the frequent necessity to address the communications to several recipients in a good way.
As a matter of facts, it avoids the wasting of bandwidth, that, on the contrary, unicasting generates allocating a single one-to-one session for each subscriber, creating also bottlenecks of the networks. Besides, it also solves the problem of flooding used in broadcasting, avoiding useless deliveries to not interested hosts. In this way the overloading problems decrease.
There are also some common issues among unicasting, multicasting and broadcasting, for example routers updates data represent a scalability problem the more network grows, the more packets are necessary to keep routing tables updated.

But multicasting has some issues that unicasting and broadcasting have not. For instance, **TCP feedback or error packets** are very useful in unicasting, in order to implement the error and flow control. But in multicasting these reports cannot be used, because it can be expected more sources of feedback from a

multicast session than from a unicast session. So, using an unicast-style method, the source would be easily overloaded.
For this reason, different techniques are necessary to manage the big amount of control data created in a multicast session [3].

Another problem of multicasting is the necessity of the routers of **maintaining the knowledge of network topology**. In a large part of multicasting schemes, the routers have the responsibility to inquire each other about the location of the participant hosts.
It means that every router has to know every variation in the delivery spanning tree, to avoid sending messages to leaves (subnetworks) without any host group's member inside. This refreshing costs enough, because a lot of control messages are necessary to maintain the consistency of the knowledge of the all routers. But the question is also about **scalability**, because as the number of members of the host groups increases, the amount of information that routers have to store grows.
Besides, this additional data to be managed might pinch off resources destined to others router's tasks.
All these points make obvious that multicast routing protocols are not designed as well as to scale multicasting until covering the entire Internet.

As mentioned in this chapter, the necessity to use *QoS-based resource reservations combined with multicast delivery methods*, is the consequence of recent multimedia diffusion.
In order to guarantee QoS parameters through a resource reservation, several new protocols have been developed. For instance RSVP (Resource Reservation Protocol) is a protocol that aim to divide the network traffic according some QoS requirements, communicated through a *flow specification* data structure [16].
A *filter specification* is associated to each data structure, in order to connect the QoS requirements to a particular data flow (flow specification along with filter specification form the *flow descriptor*) [17].

RSVP discriminates the traffic in three different categories:

- **best effort** traffic, i.e. the traditional IP traffic. Flow of this kind are processed by the usual *best effort service* of the IP networks [16]. The application associated to this category are file transfer, mail transmission, transaction traffic, disk mounts, etc…
- **rate-sensitive** traffic, that offers *a guaranteed bit-rate service* in which the bandwidth has to be chartered, even at the expense of the delay. For example, H.323 videoconferencing use this kind of service, because it needs a (nearly) constant rate.
- **delay-sensitive** traffic, in which the focus is the reduction of the delay, also giving up the bandwidth consume. In this case, the rate change accordingly to the required time limit of delivery. For instance in MPEG-2 there are two kinds of frame: key frame, that represents a real data, and delta frame (smaller), describing the changes compared to the key frame. So the delta-frame rate varies, in order to stay into the designed time limit. The delay sensitive traffic can support the *controlled-delay service* and the *predictive service*. The latter is a non-real time service, the former is a real-time service.

Another important aspect of RSVP is the **Reservation style**, that defines the parameters with which the reservation is defined. The reservation can be *distinct*, if for every flow of the session corresponds a sender, or *shared*, if several senders use the same flow without interferences.
The fixed-filter (FF) style (distinct reservation) implies a different reservation for each sender. So the total reservation on a link in a certain session is the total of the FF reservations for all requested senders.
In the *wildcard-filter (WE) style* (shared reservation) all senders share the same resource reservation, propagated upstream towards all senders. When a new sender appears, the resource reservation is automatically enlarged to it.

Finally, in the *shared-explicit (SE) style* (shared reservation), the receiver making the reservation specifies an explicit list of senders that can share that reservation [17][16].

But supporting services such as bandwidth reservation or latency guarantees, to offer the quality asked by each recipient, could end in a overloading of the network. It happens because the network should transmit a consistent amount of traffic (because multimedia contents are usually larger than others) from one source (that, as previously said, can converts in a bottleneck, causing network congestions) to multiple addressees at the same time.

3.5 Video transmission in P2P networks

Since peer-to-peer networks have more potentialities about scalability than traditional Client/Server ones, p2p approach can be implemented in transmission of multimedia data. In recent time, multimedia contents are the most requested and the most popular data on Internet, as a matter of fact a multitude of multimedia applications are spreading toward this field.

Among the multitude of multimedia data, video is the most popular, but it also has more requirements in terms of **bandwidth**. Indeed it needs between 10kbps-5Mbps, but many *compression methods* can reduce its consumption. Using also some *adaptive coding methods*, it is possible to modulate video's quality according the variability of the network conditions (MPEG-4 scenes for example [17]). Another stringent constraint in video transmission is the **end-to-end latency**.

Dealing with video contents in peer-to-peer networks, three kinds of transmission can be distinguished:

- **delay tolerant file download of archival material** [18], in which there is a certain elasticity in completion of video download before starting with its visualization. There is a receiver buffer that keeps the data until video transfer finishes, afterwards it gives an acknowledgment to the user to notify the download completion. In the end, user starts to play video without stops or quality degradations.

- **delay sensitive progressive download (or streaming) of archival material** [18], in which there is an application that decides the instant in which video playback can start. This estimation aims to start video rendering as soon as buffer is sufficiently full in order to let user see video without interruptions. Actually, the application makes this evaluation considering that the difference between download rate and playback rate never should deplete the buffer before the end of file.
- **real-time live streaming** [18], in which just an initial buffering of not more than a few seconds is tolerated. This is because the main goal in this case is allowing the recipient start to see something almost immediately. Of course, among the three modalities here presented, real-time live streaming has the most stringent delay requirement.

Because of the soaring interest in videos, lots of peer-to-peer commercial applications working in video live streaming, such as Joost, LiveStation, SOPCast, TVants, Zattoo[18], PPLive[19], etc. have been introduced in recent years.

These systems take advantage from *peer-to-peer distributed environment*: as a matter of a fact, in p2p networks finding video contents is easier than in Client/Server ones, just because usually p2p networks are constituted of a multitude of nodes, each of which shares its resources, its files and its multimedia contents too. Reaching a so huge amount of nodes is very difficult in Client/Server environments, due to more stringent scalability limits [8] (the dependence of servers' capabilities is the most heavy limitation).

A widespread communication technology that use peer-to-peer networks is **IPTV**. Because of its potentially hundreds of millions of users, it is impossible thinking to realize an IPTV architecture in a Client/Server network, because managing a so large amount of clients needs a lot of power servers, and it should cost too much. For this reason, IPTV is used on p2p networks, through which it is able to provide watching video streams until about 500 kbps per user [19].

Another field of application for peer-to-peer network is the **satellite TV**. **Zattoo**[18] works in this contest, it is one of the most popular live streaming providers in Europe and it consists of a peer-to-peer live streaming system that rebroadcasts satellite TV on internet. Zattoo is constituted of more than 3 millions of registered users, with a maximum of over 60,000 concurrent users on a single channel. For each TV channel a p2p delivery network is dedicated, so peers can switch among channels just leaving and joining these p2p networks.

Joining operations are quite fast, as a matter of a fact they comport a medium delay of 2-5 seconds. The only limitation in this sense is that peers can join only one network at any one time.

Zattoo is defined as a *received-based peer-division multiplexing (PDM) protocol*, because the receiving peer can decide how to multiplex a data stream, electing by itself a set of neighboring peers with which to share its contents. A virtual circuit of neighboring peers is built to avoid the forwarding of several pre-packets handshaking among peers. This circuit is maintained until the joining peer switches to another TV channel.

The architecture of a typical TV channel on the Zattoo network (showed in figure 3.3) is constituted by two main clusters of servers: the **broadcast servers** perform the operations concerning the download from satellite (H.264/AAC - Advanced Audio Coding - stream), than the encryption of data and finally the delivery to the viewers; the **administrative servers** instead work on all operations concerning control deliveries and users' authentication.

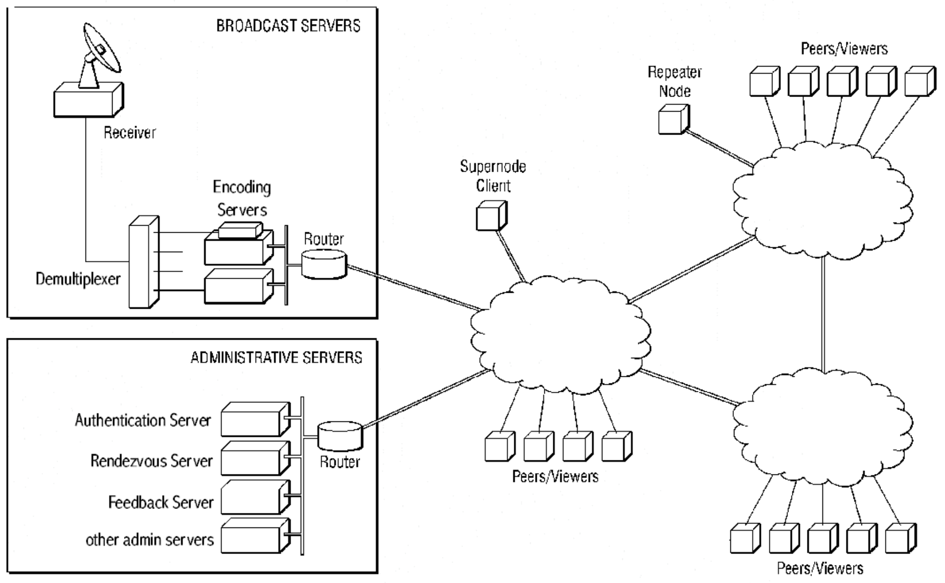


Fig. 3.3: Zattoo single channel architecture

In detail, in broadcast servers block there are:

- the satellite, i.e. the video source;
- the demultiplexer, that encodes the signal from satellite into a variable-bit rate stream. If source is busy, generated data is packetized into a packet stream, each of these ones has a limited size (fixed maximum); instead if source has not a big workload to attend, no data is generated;
- the Encoding Server, that multiplexes the packets streams as segments constituted of n logical sub-streams. Fixed the parameter i as packet index ($1 \leq i \leq n$) and m as segment index ($m \geq 0$), this server works as a cyclic mechanism. Now fixing the parameters:
  - $x = n \cdot i$
  - $y = m \cdot n + i$

it can be deduced that the x-th packet belongs to the x-th sub-stream. Since each sub-stream carries the y-th packets, the number y actually represents the sequence number of the packet;

- the router, to deliver the packets to the recipient peers.

Instead, in the administrative servers block there are:

- the Authentication Server, necessary to let a peer able in receiving the signal of the channel. Upon authentication procedure, the user is granted a ticket with limited lifetime;
- the Rendezvous Server, that receives the tickets from the users who want to view a TV channel. If the ticket authorizes a particular user to watch a particular TV channel during a certain time, the Rendezvous Server returns to this user a list of peers currently joined to the correspondent p2p network (carrying the selected channel) together with a signed channel ticket. If there are no peers connected at that time, the user receives only the Encoding Server. Afterwards the user contacts the peers presented into the received list, receives from them a live stream of the channel, finally joins the channel and starts to view the video content;
- the Feedback Server, that collects users' error logs submitted asynchronously by users.

Because of extremely dynamicity of a peer-to-peer network like Zattoo, an algorithm to check and avoid errors on the channel is required.
For this reason, Zattoo employs the **Reed Solomon error correcting code** (RS-ECC) to forward error corrections. It is a systematic code, based on redundant information: in a segment formed by n packets, k < n of these ones contain the actual data information, others n-k ones just carry redundant data necessary to control if data received are corrupted or not.
Since the high variability of source bit rate, some segments could be of size less than the maximum allowed into the system: it doesn't represent an efficient

condition to compute quickly the RS algorithm, so these smaller packets are "zero-padded" to the maximum packet size.

As mentioned above, IPTV is an emerging internet application that uses peer-to-peer networks.

For instance, **PPlive**[19] is a free peer-to-peer architecture for IPTV application, and it is defined as a *mesh-pull p2p streaming system*. In this network, carried video contents are not property of PPlive, but they are forwarded from TV channels. According to some measurements, published on PPlive website in May 2006, this p2p framework provides more than 200 different channels, with an average of 400,000 daily users. The majority of the channels has a bit rate of video programs into the 250 kbps - 400 kbps range, but few channels can reach also 800 kbps.

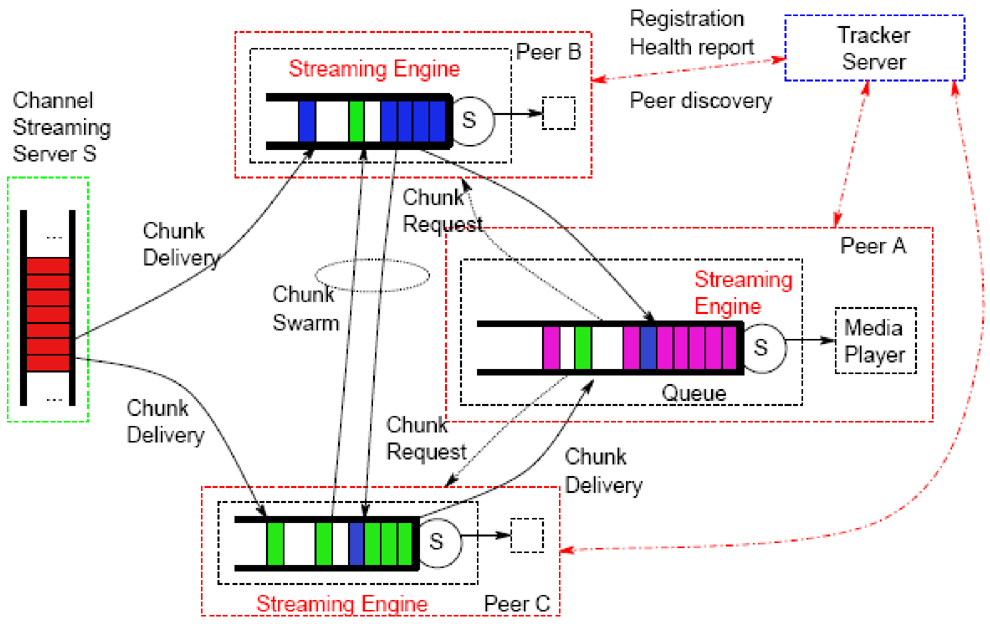


Fig 3.4: Mesh-pull p2p live streaming architecture[19]

As figure 3.4 shows, there are 3 entities into the system that perform different tasks. The **Channel Streaming Server** converts the multimedia content into small video chunks, to provide a simpler delivery to the peers. Instead the **Tracker Server**, takes charge of all operations aimed to allow peers join the network and download video chunks from multiple peers. It provides to each peer all information about joined peers and relative available chunks of the media content of interest. In the end, the **Streaming Peer Node** is the usual peer that can download or upload video chunks on the network. It is constituted by:

- a *streaming engine*, that executes all operations about download of video chunks from others peers or directly from the channel streaming server;
- a *queue*, inside to the streaming server, in which chunks are stored before rendering the video;
- a *multimedia player*, that takes care of reassembling chunks and play-backing the video to the user (for instance, Windows Media Player or Real Player). WMV (Windows Media Video) and RMVB (Real Media Variable Bitrate) are the two video formats used to encode the channels' streams.

The video display mechanism is based on some interactions between the streaming engine and the multimedia player, as figure 3.5 shows.

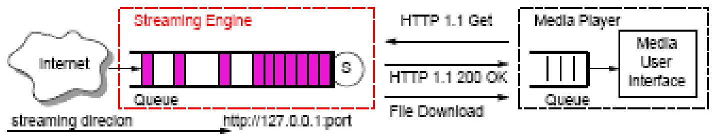


Fig 3.5:Interactions between Steaming Engine and Media Player inside a Streaming Peer node of a mesh-pull p2p live streaming system[19]

When the waiting queue has collected enough contiguous chunks, the streaming engine launches the media player and then sends an HTTP request to the streaming engine. In the end, the engine answers sending the video to the media player. The media player also has an internal queue to buffer the received chunks from streaming engine. When this queue becomes enough full, video rendering starts.

If during playback, the streaming engine becomes disabled to continue in providing the video player with sufficient chunks (for example if some source peers has leaved the network, streaming engine needs time to look for others sources), then the media player will starve. Starvation could be more severe or less severe: in the first case the video will be frozen for a certain time, in the second hypothesis the media player will skip some video frames, but the rendering of video will not been interrupted.

3.6 Limitations of live video transmission in peer-to-peer network

As already said before, large-scale peer-to-peer streaming applications are developing very quickly in recent times in all Internet network. One kind of these applications is represented by the *mesh-pull p2p streaming systems*, in which different entities that work jointly to deliver chunks of video among peers (like PPlive[19] network).
Some studies about the latter have revealed that on one side they are able to accommodate millions of users at the same time, managing hundreds of channels too; but in the other side they cannot grant a given streaming quality to the all users and in each channel simultaneously [20].

The term **streaming quality** of a channel refers to the percentage of high-quality peers in the channel, where high-quality peer parameter indicates a peer that has more than 80% of its playback buffer full of video chunks. The 80% buffering level (referring to a mesh-pull system, actually it represents the number of consecutive blocks stored into the queue inside to the streaming server of a streaming peer node [19]) is an empirical threshold based on some analysis that show it reflects the playback video continuity of a peer in the following 5 minutes. According this, these researchers have taken it as a benchmark for them work.
In this work, more than two terabytes of data have been collected during one year from **UUSee**[20], one of the most famous large-scale commercial p2p live streaming framework in China, along with PPlive[19] and PPStream[20].

The aim was studying users' behavior and network's inefficiencies.

As already pointed out, streaming quality is a very important factor to determinate the user satisfaction degree in p2p networks that work in video streaming delivery. For this reason, finding the triggering causes of these inefficiencies assumes a critical importance in determining the achievement or the failure of the system.

After these studies, it has been found out that the most influential factors on these inefficiencies are:

- **the number of peers in a channel**, as a matter of a fact peers in large channels enjoy a better streaming quality, on the contrary peers in smaller channels usually get less server capacity, so low streaming quality.
  This is because the volume of peers in a channel follows a sort of "*inverse demand and supply law"*: if several peers are asking for the same video content, the demand increases along with the supply (in this case supply means portion of resources, in terms of streaming quality); vice versa if peers ask for the same video content are less, the demand and the supply too decrease.
  The scenario is more complicated, because (fixed the previous law) in both kinds of channels the situation gets worse during *daily peak hours*, when the maximum number of users is reached. During these hours, the streaming quality suffers of a generalized decrease.
- **server capacity**, because it still plays a fundamental role also in p2p streaming networks, above all during peak interval times. Besides, server upload bandwidth becomes an important factor when upload peer bandwidth doesn't aid enough to the total available bandwidth on the network: if it becomes too less, some peers could get bad video quality for a variable-long time interval.
- **inter-peer bandwidth availability**, that represents the most evident advantage that p2p networks have against client/server ones. This bandwidth is the main source of upload in a p2p streaming channel, so if

number of peers increases too much, especially during daily peak hours, inter-peer bandwidth could became not enough for supporting all the traffic. In this case, it becomes a significant bottleneck of the network.

- **intra-ISP (Internet Service Provider) and inter-ISP peer link bandwidth availability**, that reveals some possible inter-peer bandwidth bottlenecks in the vicinity of ISP boundaries.

CHAPTER 4

LISP and Corecast transmission

A lot of problems there are still in the video delivery, above all when unicast, broadcast or multicast are used. It is due to the Client/Server nature of these traditional transmission methods: in this kind of networks there are less possibilities of scalability. This is because the resources and the services that can be provided strongly depend from servers' capabilities and quantity.
Even if the adoption of multicasting in the network layer represents the best method to distribute the video live streaming transmission, it involves high deployment costs.

The situation gets better in the peer-to-peer networks, because they have no central entities that provide services. Since the p2p network is a decentralized environment, there are equal nodes that can exchange data and share resources directly, without passing through any server.
Nevertheless, some problems still remain in the p2p networks: first, because of the absence of central units, it is more difficult adopting the control flows; second, the extreme dynamicity due to the continuous leaving and joining of peers from the network, makes complicated getting the total control of QoS regarding to each peer.

Since the QoS parameters play a decisive role in the video delivery, researches are oriented towards finding new solutions to improve video services on the p2p networks, above all in the mesh-pull live streaming peer-to-peer networks.

About this theme, in this chapter a new efficient inter-domain live streaming architecture that operates on top of LISP (Locator/ID Separation Protocol) is presented: Corecast.

Before to entering into the heart of the Corecast's functionalities, it is appropriate to deal with LISP, understanding its mechanism and its functions.

4.1 The Loc/ID split

The researches' efforts are constantly oriented to finding better routing systems in order to support the scalability and the dynamicity of the modern network environments. The situation in the Internet, with the growth of the DFZ (Default Free Zone) routing tables, is becoming an actual trouble [10].

One of these studies is oriented towards giving some modifications to the addressing allocation system. The idea starts from the studying of the well-known IP address. It consists of the numeric address composed by the four decimal numbers, each of them represents a group of 8 bits.

The function of the IP address is essentially gives an unambiguous identification for each network interface that connects hosts or associated devices to the Internet [21]. Even if the general idea is that IP address accomplishes only one function, actually it plays a double role (figure 4.1):

- **locator** role, because the IP address indicates the paths used to reach the end-host;
- **identifier** role, because the IP address identifies (using the ports) the endpoint of transport flows.

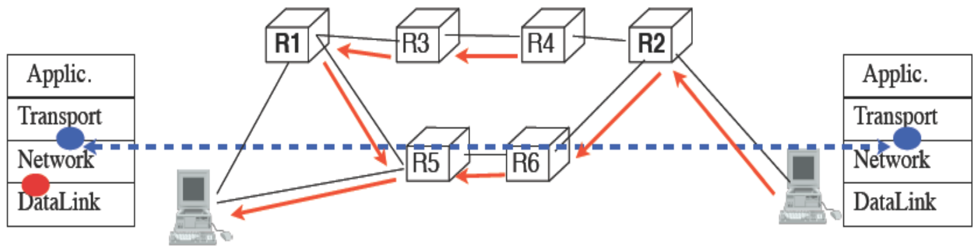


Fig. 4.1: Representation of the double role of IP address in a general network: identifier role (blue line) and the locator role (red line)[21]

Since this statement of fact, the idea to separate these two function was born. This separation is known as Loc/ID split [22], and it aims to divide the two pre-mentioned functions in this way:

- **Routing Locators (RLOCs)**, which describe how the device is linked to the network. The RLOCs are globally routable and attached to the routers.
- **Endpoint Identifiers (EIDs)**, which define the identity of the device, using a single numbering space (the IP address space). They are not globally routable, so hosts in a given area are expected to use EIDs in the same prefix.

Supporters of Loc/ID split are convinced that this double function of the IP address makes impossible to create an efficient routing protocol; instead through the separation of the function in two, they assure that a better scalability of the system will be achieved.

This firm belief is supported by the *Rekhter's Law* [22], according to it, there is an actual existence of two contrasting purposes that RLOCs and EIDs try to get. The law affirms that the RLOC and also EID addresses must be assigned in a way congruent with the network's topology, in order to render a routing system efficiently scalable. In this way, a more accurate aggregation of RLOCs can be obtained.

An example of identifier that already exists in Internet is the *loopback address*: it is implemented in routers and it is not tied to a particular physical interface. For this reason, loopback addresses are always directly published by the routing protocols and they remain reachable while one (at least) router interface is up.

Usually the EID doesn’t follow topology’s structure, but it is bounded in some organizational criterions of assignation: for this reasons, a strong incongruence arises. Because of network’s topology and organizational constraints are almost always in contrast, the realization of a single efficient address space, in order to follow both purposes (without imposing too strict restrictions), is very hard, if not impossible.

In the end, this reasoning affirms that an effective routing system has to be based on topological bases, otherwise wide scalability’s potentialities are not reachable. In this scenario, the Loc/ID split represents the best solution to resolve this problem.

4.2 LISP description

According to the Rekhter's Law [22] mentioned in the previous paragraph, the separation of locator and identifier functions is necessary to implement an efficient and scalable routing system. This operation of division is known as Loc/ID split.

Many researches are working to find suitable methods to enforcing the Loc/ID split, but nowadays nothing has been standardized yet. Basically, the usual approaches to implementing the Loc/ID split is the **map-and-encap**. This method is composed by two steps:

1. In the *map phase*, if a node wants to send a packet to a destination node’s EID outside its domain, it is necessary a mapping of the destination EID to a RLOC. So the mapped RLOC refers to a border router belonging to the destination domain. This conversion is executed on the border router of the source domain [21].

2. Subsequently, the *encap phase* starts: the source border router sets the destination address according to the RLOC obtained from the mapping process, then encapsulates the whole packet.

INNER - HEADER		OUTER - HEADER	
Source EID	Destination EID	Source RLOC	Destination RLOC

Fig. 4.1: Structure of packet's header processed with the map-and-encap method

According to this scheme, the packet has an *inner-header*, constituted by source and destination EIDs, and an *outer-header*, composed by source and destination RLOCs (as showed by figure 4.1). This double operation makes able the destination border router to decapsulating the packet and to forwarding it to the recipient [22].

The map-and-encap method does not require any changes in the core routing infrastructure, furthermore it can work with both IPv4 and IPv6 (of course address rewriting method cannot work with IPv4) and return the original source address, without any modification. This last feature is useful in some cases.

As a matter of a fact, the *Locator/Identifier Separation Protocol (LISP)*, one of the most reliable implementation of the Loc/ID split, is a network-based map-and-encap protocol.

It is also an instance of the *jack-up architecture* [22], because it is interposed between physical layer and network layer of TCP-IP stack. So, it jacks up the whole stack:

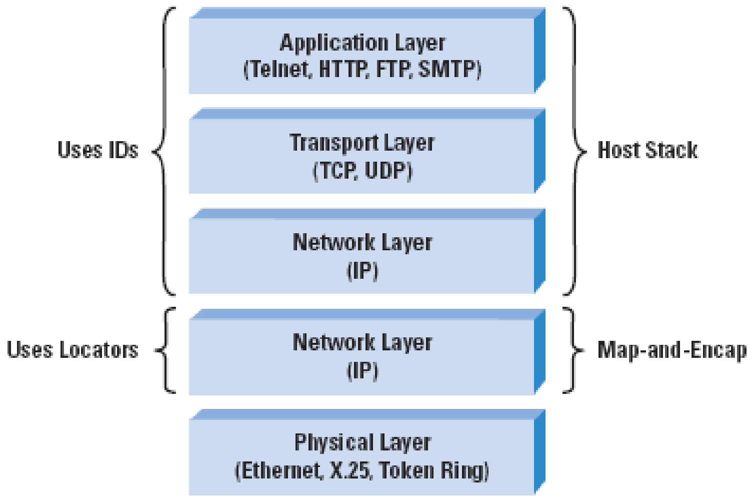


Fig. 4.3: TCP-IP stack with the added LISP layer [22]

LISP specifies two new network entities, that can physically coexist in the same router [10]:

- **the Ingress Tunnel Router (ITR)**, a router which receives in input a packet without LISP header (i.e. with a single IP header) and produces in output a LISP-encapsulated IP packet. An EID-to-RLOC mapping process has place into the router, but only if it has not been done yet for that particular EID. After this conversion, a LISP header is appended to the packet before the forwarding toward the Internet. This LISP header contains the result of the mapping (an *inner IP address* treated as an EID) in the destination address field and the RLOC in the source address field (since the RLOC is globally routable, it is considered as an *outer IP address*).
- **the Egress Tunnel Router (ETR)**, a router which receives a LISP-encapsulated IP packet in input and sends a decapsulated IP packet in output. Actually, ETR takes the outer address from the LISP header, then strips it down and forwards the packet according to the next IP address headed.

It can be elicited that an efficient mechanism to map an Endpoint Identifier onto the Routing Locator(s) of the site router(s) is necessary to execute the encapsulation of the LISP packet. This process is known as **EID-to-RLOC mapping**.

There are three types of mapping mechanism: the *push model*, in which a mapping table is received by the **LISP map server** through a certain protocol; the *pull model*, where the routers receive a packet unmapped every time, then they ask for the information to the mapping mechanism and refresh their mapping table; in the end there is a *hybrid model*, in which push model is used only for popular mappings, otherwise pull model is used.

In each of these models, the LISP packet assumes the same structure. In figure 4.4 a LISP header for IPv4 is shown.

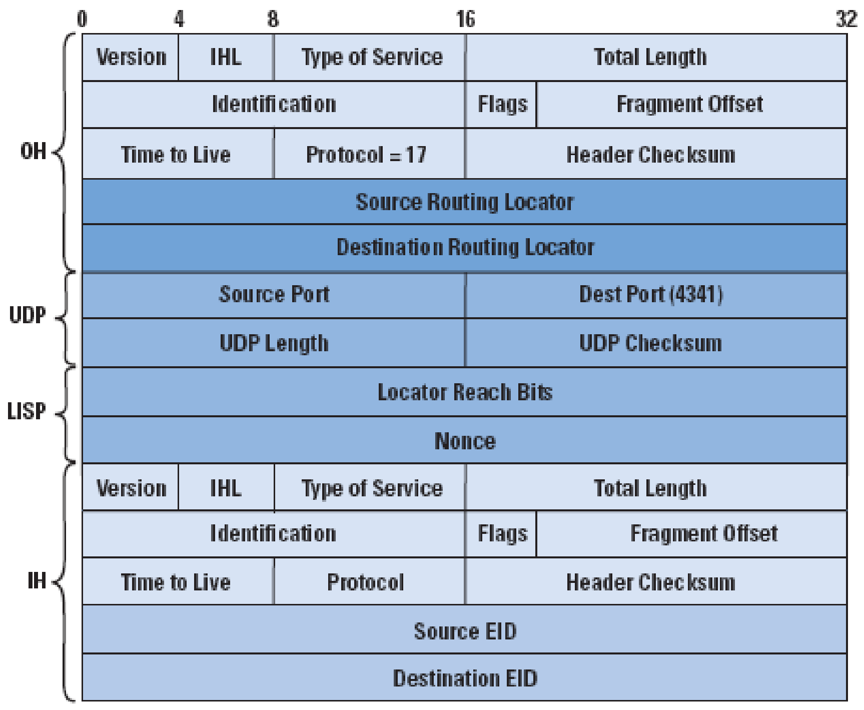


Fig. 4.4:LISP header format for IPv4 [22]

As previously mentioned, LISP is a map-and-encap protocol, (also backed by Cisco) so in its header there are two well split parts: on one hand the source Routing Locator (source RLOC) and the destination Routing Locator (destination RLOC) belong to the outer header (OH), on the other hand the source Endpoint Identifier (source EID) and the destination Endpoint Identifier (destination EID) situated in the inner header (IH). This separation is a pure representation of Loc/ID split.

LISP has great potentialities to improve network's performance, because using it makes possible to split the locator function and the identifier function.
As already pointed, this split generates improvements in scalability and routing system efficiency, because a greater aggregation of RLOCs, based on network's topology (according to the Rekhter's Law, it is an inescapable issue in order to obtain a better routing system) should be accomplished.

Also some additional advantages, concerning security and network mobility, could be obtained using LISP [22].
Besides, no big changes to the Internet framework are required: for instance LISP implementation does not imply hardware or software changes to hosts and to the router's hardware too. Furthermore it can minimize or avoid packet loss through EID-to-RLOC mapping [21].

4.3 CoreCast's proposals and aims

As already explained many times, the diffusion of users with broadband connections to the Internet has caused the growing of people's interest in multimedia applications. Among the multitude of multimedia data, video is the most popular among the users, but unluckily, it has also the *most stringent constraints in term of minimum bandwidth and minimum delay and delay jitter.*

Because of the growth of video popularity, a lot of video on demand and video streaming live applications has been developed. In this scenario, the

network layer represents the crucial point to allow the development of these services: some particular protocols able to *support the broadcast of a video content from one source to many recipient*s are necessary.
As already shown in the third chapter, the optimal solution in this sense is represented by SSM [8] systems, above all with regards to inter-domain routing. Because of their expensive costs (for example, the routing requires a big amount of traffic and a large space in router memories, too), the research community is oriented towards the Peer-to-peer world.

P2p networks, as the well-known UUsee[20] and PPlive[19], offer a decentralized environment with a lot of advantages, such as the possibility to instantiate *direct communications between the end-systems*, without passing through any central server. It causes a substantial reduction of the control traffic, as a consequence a noteworthy reduction of costs.
At the same time, the using of flooding method to deliver packets to the peers, could generate a generalized increasing of the data traffic in the system [6].
Also the high dynamicity, caused by the uncontrollable joining and leaving of peers to/from the network, represents a tricky issue, because it makes difficult controlling the QoS parameters at any time. Some studies have revealed that, during peak hours, the video streaming quality degrades as the peers quantity increases: this curious behavior brings additional problems in QoS parameters supporting [10].

Moreover, the majority of the peers have access to the network through an asymmetrical connection, so their *upload bandwidth is quite narrow*: actually, their contribution to the total available bandwidth on the system is not so sizeable.
For all these reasons, several times p2p networks are unable to guarantee a reasonable viewing experience to the all peers in the same breath [11].

In order to move around this hurdle, alternative network layer solutions are developing. To this aim, CoreCast [10] has been implemented.
It is a new efficient architecture working **on top of the LISP**, that is considered the most reliable instance of the *Loc/ID split* [22]. According to several studies

done in IETF (Internet Engineering Task Force) in the matter of improving network's scalability and routing protocol efficiency, the Loc/ID split is the best way to handle the unsatisfactory situation in which multimedia services and applications are. Because of this, LISP is considered the finest solution to hold down the worrying growth of DFZ routing tables.

With the awareness of LISP's potentialities, CoreCast aims to **reduce the inter-domain traffic**, in order to get less costs than multicast and existing p2p solutions. So, CoreCast is able to offer a reliable live streaming service, guaranteeing the respect of some QoS parameters, too. Avoiding inter domain bandwidth consumption, it is also **ISP-friendly**, because there is no traffic between peers belonging to different ISPs. This feature is well appreciated by the ISPs, because the large amount of traffic generated by p2p application (especially file-sharing ones) can be drastically reduced in this manner. Furthermore, Content Distribution Networks and ISPs could easy establish SLAs (Service Layer Agreements), because in this scenario they have not to take into account inter-domain connections.

4.4 CoreCast's functioning

CoreCast is a push-based architecture mounted on top of the LISP. Just few changes to the LISP are required in order to implement it (but it is not a problem, because LISP is not standardized yet, it is still in development phase).
To allow the Loc/Id split brought by LISP, CoreCast manages two different types of packet [10]:

- **Payload Packet** (figure 4.5), containing the payload, its identifier (hash) and its length;

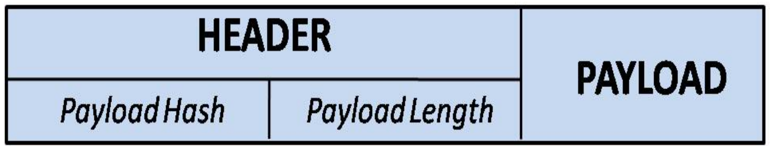


Fig. 4.5: Structure of Payload Packet

- **Header Packet** (figure 4.6), containing the IP header and the Corecast PDU. The latter is constituted by the destination EID and the hash of the payload that should be sent to the client.

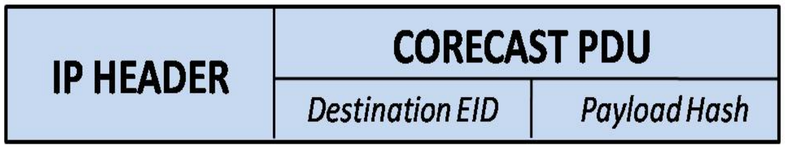


Fig. 4.6: Structure of the Header Packet

Anyhow, all LISP devices are maintained in CoreCast implementation, because they are fundamental to obtain the wanted improvements.

The figure 4.7 shows an example of a small CoreCast architecture with 3 distinct ASes (Autonomous Systems), and 8 clients distributed in them.

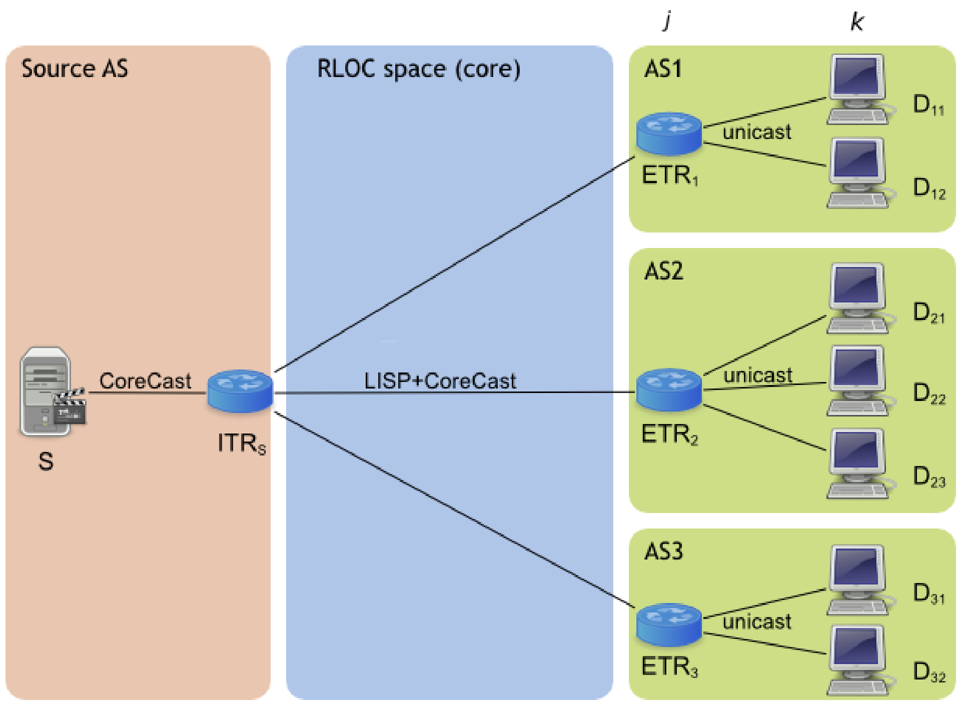


Fig. 4.7:Example of CoreCast architecture [10]

The scenario is constituted by a source *S* that wants to send some packets to *k* users, located in *j* different ASes (it could be an IPTV service, or a video live streaming communication, etc…).

The source *S* just sends the content once to its ITR, instead sending it *k* times, as it would happen in a unicast network. This is a great advantage, because the source overload and the eventuality of network congestion are avoided.

Along with the video content, the source sends a list of ETRs, too, defining the destination ASes. Later, each ETR will send the packet by a simple unicast transmission (or by IP multicasting ,in alternative: the choice depends from each AS) to the recipients inside its AS.

In order to allow the CoreCast's functioning, it's necessary keeping some memory structures into the source, the ITR and the ETRs.

Into the source, an array called **ChanDstList** (Channel Destination List) is stored, containing the list of the EIDs that are currently receiving that stream.

When a client connects to the stream, its EID is appending to the ChanDstList; on the contrary when a client switches channel or simply leaves the network, its EID is deleted from the ChanDstList.
The ITR and the ETRs have two data structures stored inside them (figure 4.8):

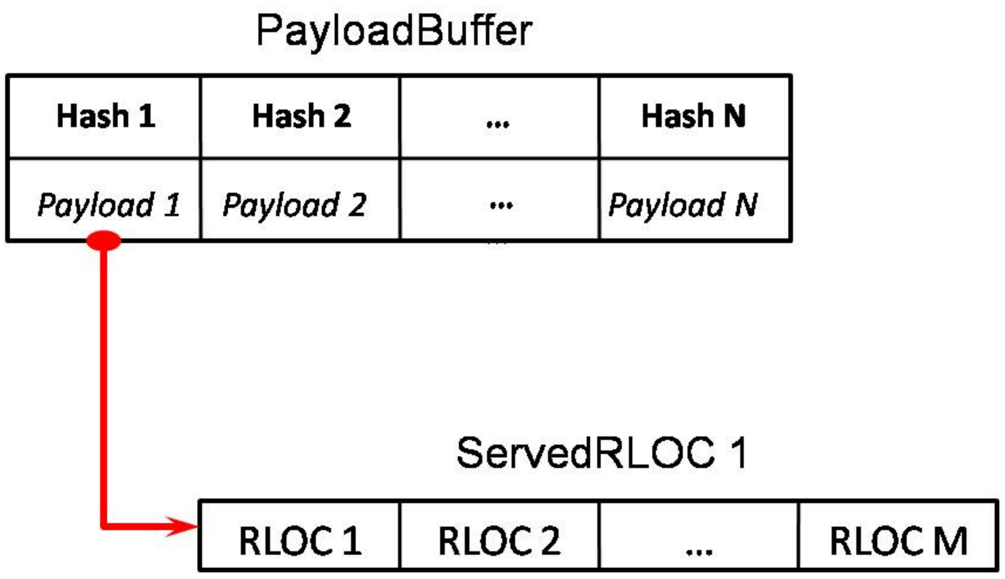


Fig. 4.8: Interaction between PayloadBuffer and ServedRLOC1

- the **PayloadBuffer**, containing the list of the current payload packets of each stream. This buffer is indexed by the hash of the payload packets, so a certain payload is univocally indentified by the couple *hash(payload)*.
  To save memory, actually only one payload per stream at a time is held;
- the **ServedRLOC**, a vector holding a list of RLOCs. Every element of the payloadBuffer points to a ServedRLOC. So, each ServedRLOC represents the RLOCs that have already received the payload, referring to the one that is pointing the current ServedLOC.

These memory structure does not represent a high cost in terms of memory spaces, because few Mb are enough to contain all of these arrays (for more details see in [10]).

As already pointed, for each channel corresponds an EID, besides an EID space is reserved to the CoreCast streams into the local domain. For each of these channel, the source divides the multimedia data into chunks, then, for each chunk, S executes the following operations:

1. sending the first packet with the payload (payload packet of LISP);
2. setting the destination address of the EID, relative to the reserved channel;
3. after the first sending, the source repeats this procedure following its ChanDstList. So a header packet is sent to every EID of the ChanDstList.

This process is periodically iterated, according to the bandwidth requested for the stream (further on, this aspect will be take into account to make calculus about some characteristics of the CoreCast protocol)

Afterwards, the packet reaches the source ITR, that represents the first demultiplexer point of the network. The ITR makes some checks on the packet, in order to understand toward which ASes is it addressed:

1. the identifying of the channel, using the reserved destination EID contained into the header IP of the payload packet;
2. the extracting of the EID and the correspondent hash from the CoreCast header;
3. the EID-to-RLOC mapping, a function of the LISP that allows the ITR looking up to the RLOC associated to the client EID: using the extracted hash from the CoreCast header, the ITR checks the payloadBuffer:
    a. if the correspondent payload is found, the ITR checks into the pointed ServedRLOC, in order to find the relative RLOC. If there is the RLOC, it means that the payload has been already received by the ETR. So the ITR forwards the header to the ETR through a simple LISP encapsulation;
    b. otherwise, the payload is created before sending the header to the ETR.

Finally, the packet reaches the ETRs, representing the second demultiplexer point of the network. They work like an ITR, but with a difference: in output, rather than sending the headers using a LISP encapsulation (like ITR does), the ETRs expand the header, obtaining normal payload packets. Then, each ETR sends the extracted payloads to the end-systems belonging to its AS (receiving hosts) through unicast connections (also IP multicasting can be used, it is an efficient solution in intra-domain scenarios). Data payloads are retrieved from the payloadBuffer, that is kept up to date by the ETRs, through operations of adding and removing payloads to/from it.

As mentioned before, there exist an upper limit to the number of clients that CoreCast can support. This threshold is bound with the requested bandwidth by the stream, and it is a function of the following parameters:

- C = line rate [bps]
- T = time interval between two payload packets [s]
- P = payload size [4]
- H = header packet size [4]
- BW = requested bandwidth per steam [bps]

This limit could be represented by the formula (1):

$$MaxClient_{CC} \cong \frac{C \cdot T}{8 \cdot H} = \frac{C \cdot P}{H \cdot BW} \tag{1}$$

The comparison between the formula (1) and the formula inherent to the maximum number of the clients in a unicast transmission (formula (2))

$$MaxClient_{UC} = \frac{C}{BW} \tag{2}$$

reveals that the accuracy of CoreCast protocol depends from the ratio $\frac{P}{H}$, i.e. the ratio between the payload size and the header size. As this ratio increases, the CoreCast gain in term of maximum number of supportable clients grows, too.

Furthermore, the efficiency of CoreCast depends also from the ratio $\frac{k}{j}$ (referring to the figure 4.7), that represents the distribution of the clients among ASes. If this ratio is high, it means that the clients are spread among a small number of ASes: this is a good scenario for CoreCast, because when the ASes are not a lot, an impressive bandwidth saving could be obtained.

Strongly connected with this aspect, the differentiation between intra-domain (connections between ETRs and end-systems) and inter-domain traffic (connections between the ITR and the ETRS) assumes a crucial importance in CoreCast's efficiency. The aim, of course, is the minimization of the latter, because it represent a very high cost.

The evaluation of the impact of both on the total amount of traffic, can be made using the following two formulas:

$$BW_{\mathrm{int}\,er} = \frac{1}{T}\left(\left(LISP + CCP\right)\cdot j + \left(LISP + CCH\right)\cdot k\right) \tag{3}$$

$$BW_{\mathrm{int}\,ra} = \frac{1}{T}\left(\left(IP + DATA\right)\cdot k\right) \tag{4}$$

where LISP and IP denote respectively the LISP header size and IP header size, CCP and CCH represent respectively the CoreCast PDU size for a payload packet and a header packet, finally DATA is the size of a multimedia chunk.

As formula (3) shows, in this contest the parameters *j* and *k* play also an important role.

Analysis and evaluations [10] about an implementation of CoreCast on a Linux kernel, reveal that it comports an *increase of CPU usage of about 52% on average*, compared with a normal unicast transmission. Anyway, this increase should not be a limit for the router hardware and software, because several optimizations, in order to support the new operations introduced by CoreCast, can be easily implemented.

It is also necessary taking into account that this CoreCast implementation is not optimized to work on a Linux kernel as the unicast one is (it has already been fit since several years). This consideration ends to the belief that the processing overhead of the CoreCast is not a risk for router's performance (more details about these researches can be found in [10]).

In conclusion, CoreCast can be considered a simple protocol requiring small memory costs and no reconfiguration of routers; besides it is *ISP-friendly*, because its main aim is the minimization of inter-domain traffic.

Thanks to these aspects, CoreCast allows ISPs and Content Distribution Network to relate independent SLAs, regardless of the inter-domain connections.

Note that CoreCast does not take into account the operations of AAA (Authentication, Authorization, and Accounting), as a matter of fact the source can manage this operations at application layer, freely choosing a framework. This is due to the fact that CoreCast does not operate at application layer, like the majority of the p2p live streaming services, creating a p2p overlay. On the contrary, it works at network layer on top of the LISP, taking advantage of the Loc/ID split [22]. This characteristic is the most strong point of CoreCast, because gives it the potentiality to be adopted in the largest live streaming frameworks of the Internet.

CHAPTER 5

Work environment and analysis' bases

The fifth chapter deals with the instruments and the selected methods used to realize this thesis. It consists of a series of analysis based on a big amount of data traces captured using Wireshark. It is the well-known sniffer program, used in a lot of industries and universities in the world.
The aim of this work is studying and understanding the potentialities of the CoreCast protocol considering various points of view. The process of evaluation consists in the extraction of some relevant data and ratios in order to compare CoreCast performance with a normal p2p network for live video streaming.
The traces have been captured on the occasion of the *Michael Jackson funerals* (Staples Center, July 7th - 2009), a very popular event in all over the world.

Several tens of millions of people went on line that day to watch the event via TV and via Internet: CNN, Facebook, and the main p2p platforms, such as PPlive[19] and UUSee[20], transmitted the event for about four hours.
Because of the huge volume of users, as a consequence of data too, it is reasonable thinking that the estimations obtained in this case should be actually representative, and should be considered as one a realistic scenario for testing CoreCast.

This event has involved a large volume of data, for this reason it is supposed that these researches acquire a relevant meaning.

The experiments done during this work regards the traffic sniffing from two different networks at the same time, one in Spain and the other in Romania. Even if the true duration of the event has taken 2 h, the traces captured regard a total interval of 3 h and 41 minutes: this choice aims to cover also the time lack of the preparations to the funerals and the final interviews (also these parts could be interesting for the majority of the users). The number of the users is different in the two networks, there are 3292 ones in Spanish scenario and 24463 ones in Romanian scenario. The studied traces bring several information, such as the source and the destination IP address, the transport-layer protocol name, the payload size, the timestamp, the source and destination port number, etc…

5.1 The general meaning of the analysis

The present work is based on several analysis realized in many steps. The point of beginning of everything are the **traces**.

These traces regard the Michael Jackson funerals, precisely they concern the video live streaming of the event on the **PPlive network** [19], that, as mentioned in the second chapter, is one of the biggest p2p live streaming network of the world.

The event has been caught from two different points of the network: the first point of the survey was located at the Universitat Politècnica de Catalunya (UPC) of Barcelona - Spain, the second one was located at the Universitatea Tehnica Cluj Napoca (UTCN) – Romania.

Through the collaboration of both universities it has been possible collecting a huge amount of data, in the form of data traces of PPlive traffic sniffing by the

Wireshark. The collecting of the traces has been done in the same breath from Spain and Romania during an interval time of about four hours.

The following block datagram (figure 5.1) represents the logical and chronological operations done in order to get our results:

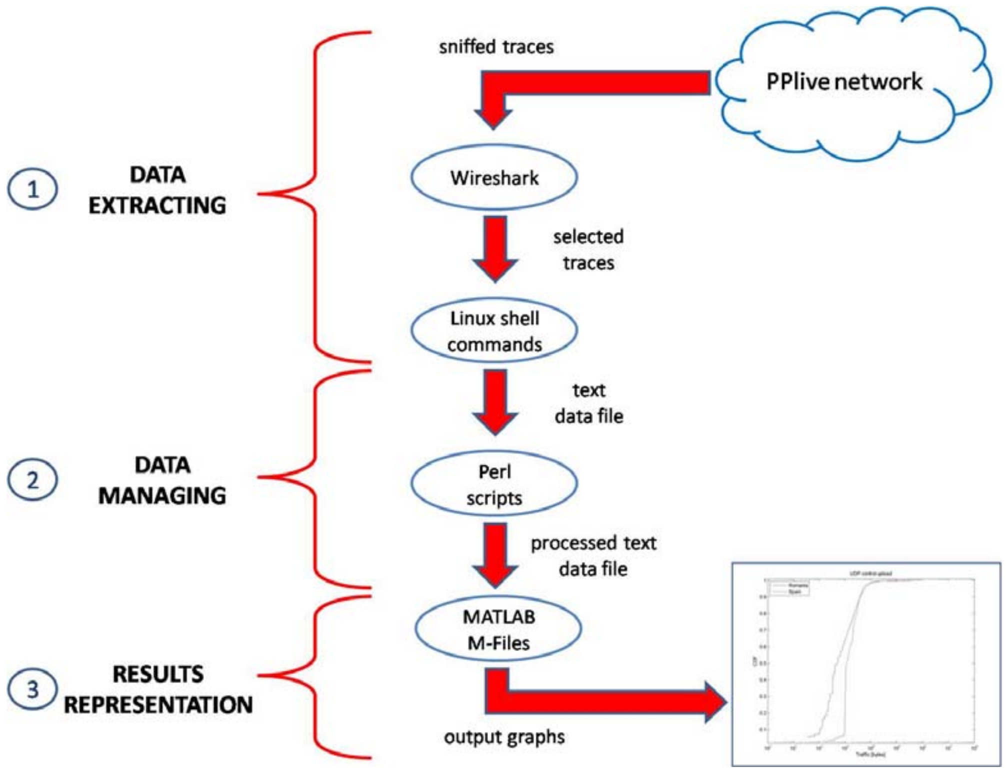


Fig. 5.1: Block diagram representing the logical and chronological data processing

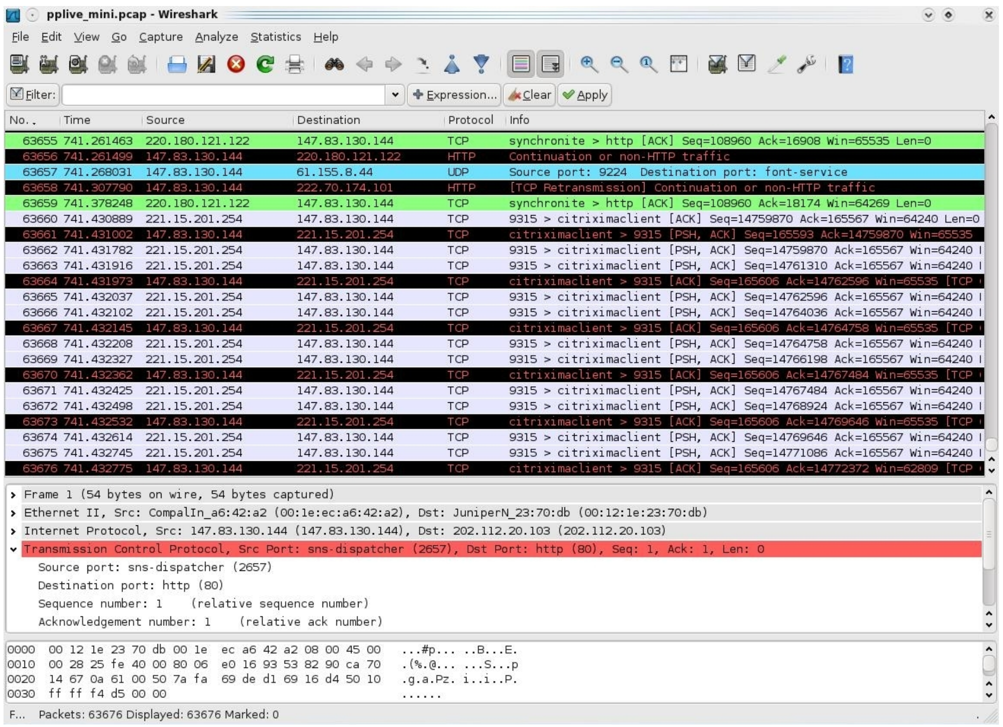
As the figure 5.1 shows, the work is fundamentally composed of three steps:

1. *data extracting from* the traces, i.e. the choosing of the fields of interest for the particular analysis that has been executed in that moment. In this phase the used instruments are Wireshark and some Linux shell commands.
2. *data managing* and elaboration of relevant information, using Perl scripts.
3. *result representation and interpretation* using M-file of MATLAB, in order to produce some representative diagrams.

During the following paragraphs each of these phases will be explained with more details.

5.2 Data extracting

This first step of the work aims to catch the data traces from the PPlive network, then to examine them using **Wireshark**. Through this software, the fields and the kinds of data of interest can be selected from time to time and forwarded in output.
Wireshark project started in 1998 and today represents one of the world main sniffing and analyzing network protocol. It is frequently used in many industries, universities and educational institutes [23].
Wireshark runs on almost all UNIX platforms and on the majority of the Windows platforms, too. In order to be executed, it requires GTK+, GLib, libpcap and some others libraries [24]. It is an open source software, released under GNU GPL (General Public License). During the realization of the present researches, Wireshark in version 1.2.0 on Debian GNU/Linux has been used.


pplive_mini.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression...
Clear
Apply

```
No. .  Time        Source            Destination       Protocol  Info
63655 741.261463  220.180.121.122   147.83.130.144    TCP       synchronite > http [ACK] Seq=108960 Ack=16908 Win=65535 Len=0
63656 741.261499  147.83.130.144    220.180.121.122   HTTP      Continuation or non-HTTP traffic
63657 741.268031  147.83.130.144    61.155.8.44       UDP       Source port: 9224  Destination port: font-service
63658 741.307790  147.83.130.144    222.70.174.101    HTTP      [TCP Retransmission] Continuation or non-HTTP traffic
63659 741.378248  220.180.121.122   147.83.130.144    TCP       synchronite > http [ACK] Seq=108960 Ack=18174 Win=64269 Len=0
63660 741.430889  221.15.201.254    147.83.130.144    TCP       9315 > citriximaclient [ACK] Seq=14759870 Ack=165567 Win=64240 Len=0
63661 741.431002  147.83.130.144    221.15.201.254    TCP       citriximaclient > 9315 [PSH, ACK] Seq=165593 Ack=14759870 Win=65535
63662 741.431782  221.15.201.254    147.83.130.144    TCP       9315 > citriximaclient [PSH, ACK] Seq=14759870 Ack=165567 Win=64240
63663 741.431916  221.15.201.254    147.83.130.144    TCP       9315 > citriximaclient [PSH, ACK] Seq=14761310 Ack=165567 Win=64240
63664 741.431973  147.83.130.144    221.15.201.254    TCP       citriximaclient > 9315 [ACK] Seq=165606 Ack=14762596 Win=65535 [TCP
63665 741.432037  221.15.201.254    147.83.130.144    TCP       9315 > citriximaclient [PSH, ACK] Seq=14762596 Ack=165567 Win=64240
63666 741.432102  221.15.201.254    147.83.130.144    TCP       9315 > citriximaclient [PSH, ACK] Seq=14764036 Ack=165567 Win=64240
63667 741.432145  147.83.130.144    221.15.201.254    TCP       citriximaclient > 9315 [ACK] Seq=165606 Ack=14764758 Win=65535 [TCP
63668 741.432208  221.15.201.254    147.83.130.144    TCP       9315 > citriximaclient [PSH, ACK] Seq=14764758 Ack=165567 Win=64240
63669 741.432327  221.15.201.254    147.83.130.144    TCP       9315 > citriximaclient [PSH, ACK] Seq=14766198 Ack=165567 Win=64240
63670 741.432362  147.83.130.144    221.15.201.254    TCP       citriximaclient > 9315 [ACK] Seq=165606 Ack=14767484 Win=65535 [TCP
63671 741.432425  221.15.201.254    147.83.130.144    TCP       9315 > citriximaclient [PSH, ACK] Seq=14767484 Ack=165567 Win=64240
63672 741.432498  221.15.201.254    147.83.130.144    TCP       9315 > citriximaclient [PSH, ACK] Seq=14768924 Ack=165567 Win=64240
63673 741.432532  147.83.130.144    221.15.201.254    TCP       citriximaclient > 9315 [ACK] Seq=165606 Ack=14769646 Win=65535 [TCP
63674 741.432614  221.15.201.254    147.83.130.144    TCP       9315 > citriximaclient [PSH, ACK] Seq=14769646 Ack=165567 Win=64240
63675 741.432745  221.15.201.254    147.83.130.144    TCP       9315 > citriximaclient [PSH, ACK] Seq=14771086 Ack=165567 Win=64240
63676 741.432775  147.83.130.144    221.15.201.254    TCP       citriximaclient > 9315 [ACK] Seq=165606 Ack=14772372 Win=62809 [TCP
```

```
> Frame 1 (54 bytes on wire, 54 bytes captured)
> Ethernet II, Src: CompalIn_a6:42:a2 (00:1e:ec:a6:42:a2), Dst: JuniperN_23:70:db (00:12:1e:23:70:db)
> Internet Protocol, Src: 147.83.130.144 (147.83.130.144), Dst: 202.112.20.103 (202.112.20.103)
v Transmission Control Protocol, Src Port: sns-dispatcher (2657), Dst Port: http (80), Seq: 1, Ack: 1, Len: 0
    Source port: sns-dispatcher (2657)
    Destination port: http (80)
    Sequence number: 1    (relative sequence number)
    Acknowledgement number: 1    (relative ack number)
```

```
0000  00 12 1e 23 70 db 00 1e  ec a6 42 a2 08 00 45 00   ...#p... ..B...E.
0010  00 28 25 fe 40 00 80 06  e0 16 93 53 82 90 ca 70   .(%.@... ...S...p
0020  14 67 0a 61 00 50 7a fa  69 de d1 69 16 d4 50 10   .g.a.Pz. i..i..P.
0030  ff ff f4 d5 00 00                                  ......
```

```
F...  Packets: 63676 Displayed: 63676 Marked: 0
```

Fig. 5.2: A screenshot of Wireshark work environment

Looking at the figure 5.2 helps to have an idea of the information that the analyzed traces bring:

- *frame information*: absolute arrival time (date and hour), time delta from previous captured and displayed frame (in seconds), timestamp (i.e. the time since reference or first frame, in seconds), frame number, frame length;
- *network protocol information*: protocol name (IPv4, IPv6, ICMP, IGMP, etc…), source and destination IP address, protocol version, header length, DS field, total length, identification (hexadecimal number), flags, fragment offset, TTL, header checksum;
- transport protocol information: protocol name (UDP, TCP, DCCP, etc…), source and port number, sequence number, acknowledgment number, header length, flags, window size, checksum, payload size (in byte);

The extraction of the information is carried out through some shell commands of **TShark**, the console-based version of Wireshark. Launching few Linux shell commands, it is easy extracting the relevant data from the traces of interest from time to time. Below there is the general structure of a TShark command launched from a Linux shell, in order to get some information from the traces:

```
tshark -r INPUT FILE -Tfields -e FIELD 1 -e FIELD 2 -e ... –e
FIELD N > OUTPUT FILE
```

Here `tshark` simply indicates that what follows has to been interpreted as a command of TShark; `-r` is the indicator of the input file in pcap format, i.e. the format of the traces captured by Wireshark; `-e` precedes a field specification.
Every field indicates a particular part of a trace, for example the IP source address, or the TCP port number, or the timestamp, etc… In the end, the symbol of `>`, followed by the output file name, serves to redirect the output of the command towards the text file indicated (it could be existent or not, in the former case the precedent content of the file will be subscribed, in the latter case the file will be created at that moment).
The redirection of the output is a very useful function for this work, because it allows the immediate creation of the wanted format, without others intermediate steps.
For instance the follow command:

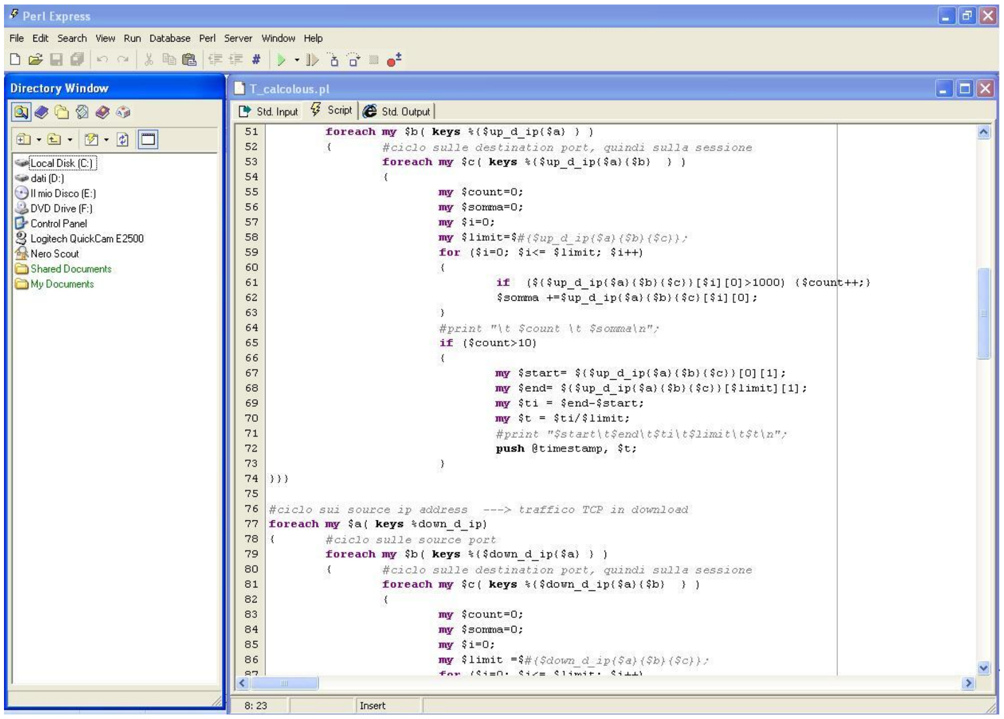
```
tshark -r pplive.florin.tcp.pcap  -Tfields -e ip.src -e
ip.dst -e tcp.srcport -e tcp.dstport -e ip.len -e
frame.time_relative > romania_T_tcp.txt
```

just takes in input the file `pplive.florin.tcp.pcap`, then extracts from it the following fields (in order): IP source address, IP destination address, TCP source port number, TCP destination port number, IP payload length and timestamp of the traces. Finally, the result is written into the text file `romania_T_tcp.txt`.

5.3 Data managing

The data managing is the second phase of the work process, and it regards the conversion of the text file obtained by Wireshark, in another text file containing the elaboration of these data. The aim is getting new information through the aggregation of the available data, in order to make considerations about the efficiency of the CoreCast protocol from several points of view.
Actually, this is the core part of the work, because the analysis and the calculations that will be made in the final third phase strictly depend from this step. So, in order to manage the data, it is necessary to use a software application able in managing text file, because both input and output file are textual (referring to figure 5.1).

For this reason the choice was the **Perl** (Practical Extraction and Report Language), one of the most popular programming language working with strings and text structures in general. The simplest operations that can be done in Perl are the *pattern matching* and the *pattern substitution* [25].
The selected platform is *Active Perl version 5.10.1* mounted on a Windows XP machine, and *Perl Express version 2.5* (figure 5.3) has been chosen as Integrated Development Environment (IDE).



```
        foreach my $b( keys %{$up_d_ip{$a} } )
        {       #ciclo sulle destination port, quindi sulla sessione
                foreach my $c( keys %{$up_d_ip{$a}{$b}  } )
                {
                        my $count=0;
                        my $somma=0;
                        my $i=0;
                        my $limit=$#{$up_d_ip{$a}{$b}{$c}};
                        for ($i=0; $i<= $limit; $i++)
                        {
                                if  (${$up_d_ip{$a}{$b}{$c}}[$i][0]>1000) {$count++;}
                                $somma +=$up_d_ip{$a}{$b}{$c}[$i][0];
                        }
                        #print "\t $count \t $somma\n";
                        if ($count>10)
                        {
                                my $start= ${$up_d_ip{$a}{$b}{$c}}[0][1];
                                my $end= ${$up_d_ip{$a}{$b}{$c}}[$limit][1];
                                my $ti = $end-$start;
                                my $t = $ti/$limit;
                                #print "$start\t$end\t$ti\t$limit\t$t\n";
                                push @timestamp, $t;
                        }
}}}

#ciclo sui source ip address  ---> traffico TCP in download
foreach my $a( keys %down_d_ip)
{       #ciclo sulle source port
        foreach my $b( keys %{$down_d_ip{$a} } )
        {       #ciclo sulle destination port, quindi sulla sessione
                foreach my $c( keys %{$down_d_ip{$a}{$b}  } )
                {
                        my $count=0;
                        my $somma=0;
                        my $i=0;
                        my $limit =$#{$down_d_ip{$a}{$b}{$c}};
                        for ($i=0; $i<= $limit; $i++)
```

Fig. 5.3: A screenshot of the Perl Express 2.5 work environment

The Perl language is very easy, for instance, to look for a particular value inside a text file, or to make aggregations of some kind of field. One of the most useful function of Perl is the *conversion of a text row in a data structure*.

Three or four lines of text code are enough in order to implement this conversion: the input text file is opened, then the `split` function "cuts" the current text row (managed as a string), obtaining as many as wanted variables.

The following extract of coding gives an example of implementation of the split operation:

```
open(IN,"< file path/file_name.txt");
while(<IN>) {
     chomp;
     (my $s_ip, my $d_ip, my $s_port, my $d_port, my
     $byte)=split;
     […]
     }
close(IN);
```

In other words, every row of the text file called `file_name.txt` is cut in order to obtain the variables `$s_ip, $d_ip, $s_port, $d_port` and `$byte`. The `chomp` command just deletes all $INPUT_RECORD_SEPARATOR (spaces) from the string in input.
The split is iterated through a cycle on the text input file implemented by the instruction `while(<IN>)`.

Another very useful feature of Perl language is the *hash data structure*. It is the third data type in Perl, after scalars and arrays, and it consists of an associative array. It means that the elements belonging to a hash can be pointed through an arbitrary scalar value, such as number or string [25].
As each different native data types in Perl, the hash has the symbol of `%` as special starting character that identifies the type of variable (scalar has `$` and array has `@`).
The following lines of Perl coding give~~s~~ an example of hash use:

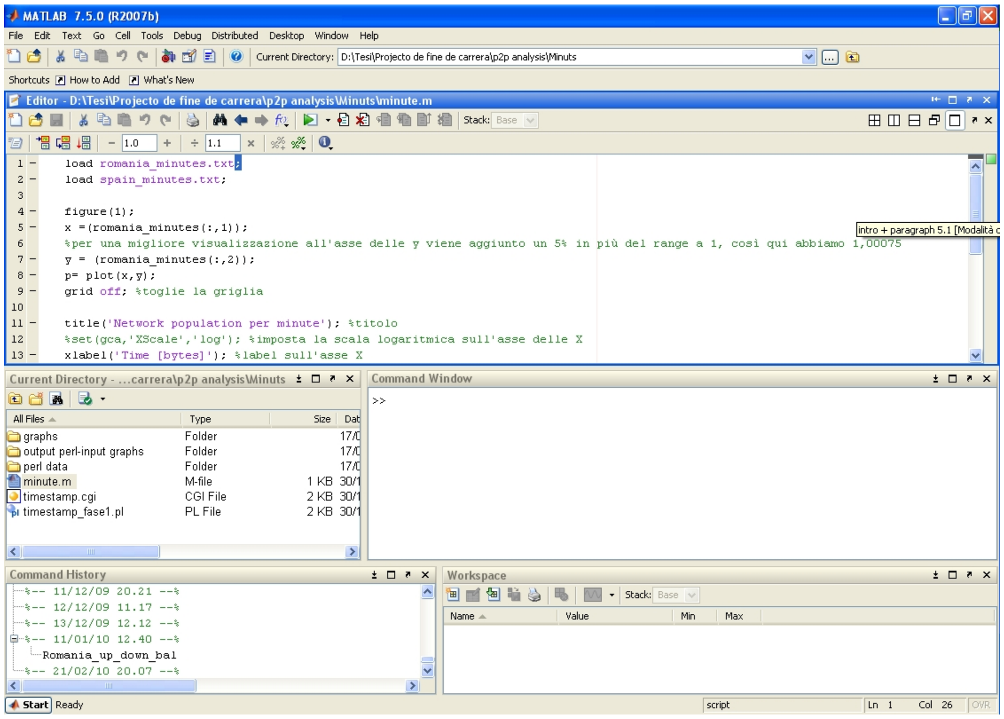
```
%tcp_s_h=();
open(IN, ,"< file path/file_name.txt");
while(<IN>) {
    chomp;
   (my $ip, my $s_port, my $d_port) = split;
   $tcp_s_h{$s_port}++;
}
close(IN);
```

The hash `%tcp_s_h` is instantiated as empty at line 1, then the input text file `file_name.txt` is opened and split (like in the first example). The variable `$s_port` becomes an element of the hash `%tcp_s_h` through the instruction at line 6.
Using hash and array data conjunctly, multidimensional data structure can be implemented, too.

5.4 Data representation and results interpretation

The last phase of the work process concerns the representation of the results through graphs. Converting data in graphical form makes their interpretation and understanding more intuitive and immediate.
Data extracted from the traces and elaborated by the Perl scripts are synthesized in a text file, that represents the ingress point of the present process of conversion (referring to figure 5.1). As mentioned before, *the output of the process will be the representation of the data in a graphical form*.

The instrument used in order to realize this reinterpretation is the popular **MATLAB** (Matrix Laboratory) version 7.5.0 (R2007b), a high level technical computing language, widely used in scientific and industries areas.
Its environment (shown in figure 5.4) lets users solve technical and mathematical problems faster than traditional programming languages [26].


MATLAB 7.5.0 (R2007b)
File Edit Text Go Cell Tools Debug Distributed Desktop Window Help
Current Directory: D:\Tesi\Projecto de fine de carrera\p2p analysis\Minuts
Shortcuts How to Add What's New
Editor - D:\Tesi\Projecto de fine de carrera\p2p analysis\Minuts\minute.m
Stack: Base
intro + paragraph 5.1 [Modalità c
Current Directory - ...carrera\p2p analysis\Minuts
Command Window
Command History
Workspace
Name Value Min Max
Start Ready
script Ln 1 Col 26 OVR

```
load romania_minutes.txt;
load spain_minutes.txt;

figure(1);
x =(romania_minutes(:,1));
%per una migliore visualizzazione all'asse delle y viene aggiunto un 5% in più del range a 1, così qui abbiamo 1,00075
y = (romania_minutes(:,2));
p= plot(x,y);
grid off; %toglie la griglia

title('Network population per minute'); %titolo
%set(gca,'XScale','log'); %imposta la scala logaritmica sull'asse delle X
xlabel('Time [bytes]'); %label sull'asse X
```

```
>>
```

All Files	Type	Size	Dat
graphs	Folder		17/0
output perl-input graphs	Folder		17/0
perl data	Folder		17/0
minute.m	M-file	1 KB	30/1
timestamp.cgi	CGI File	2 KB	30/1
timestamp_fase1.pl	PL File	2 KB	30/1

```
%-- 11/12/09 20.21 --%
%-- 12/12/09 11.17 --%
%-- 13/12/09 12.12 --%
%-- 11/01/10 12.40 --%
  Romania_up_down_bal
%-- 21/02/10 20.07 --%
```

Fig. 5.4: A screenshot of the MATLAB 7.5.0 (R2007b) work environment

MATLAB provides *a very large variety of applications*, such as communications, test and measurement, control design, image processing, managing code, mathematical function for algebra, statistics, Fourier analysis, filtering and numerical integration, tools for building custom graphical user interfaces, and so on.

Besides, using *several toolboxes*, the number of applications increases much more: there are some special additional modules implemented to resolve special classes of scientific and mathematical problems. Another strength point of MATLAB is the possibility to integrate its code with others languages.

Among this multitude of application that MATLAB offers, the one used in this work concerns the realization of graphics functions. The intent is the representation of the data derived from the elaboration of the PPlive traces.

In details, **M-files** have been used in order to decide all the characteristics that graphics should have from time to time. The M-files are a very powerful instrument, because they make possible collecting several instructions and executing all of them in one time. Actually the *M-file is a script* (but there is another type working as a function), executable simply keying its name (without its extension `.m`):

```
>> M-file_name
```

Just one condition: the M-file must be saved into the Current Directory of MATLAB, otherwise it cannot run.

Since all M-files realized into this work concern the rendering of some graphs, the `plot` function has been used in each of them. The following is an example of how the `plot` function works (these instructions belong to minutes.m, one of the M-files used during the present work):

```
x = (spain_minutes(:,1));
y = (spain_minutes(:,2));
p = plot(x,y);
```

The input parameter of the plot function are `x` and `y`, two vectors that represent respectively the x and y axis. The output of the function is saved into the variable `p`, in order to allow additional operations to personalize the graph rendering.

CHAPTER 6

Analyses and results

This last chapter is the core of all the thesis, as a matter of a fact it contains all analyses and experiments made during the realization of the present work.
Basically, it consists of six parts, each of which deals with a single analysis, studied in every aspect. In each part, it will be retraced the entire path (referring to figure 5.1), from the extraction of the data to the realization of the results. Finally the obtained results are commented and evaluated.

As mentioned in the previous chapter, all analysis derive from the elaboration of the traces relative to Michael Jackson funeral. The traffic has been captured at the Universitat Politècnica de Catalunya (UPC) of Barcelona and at the Universitatea Tehnica Cluj Napoca (UTCN) at the same time.
The traces are produced by PPlive, one of the most popular p2p video live streaming platform in the Internet.
Then, the traces have been saved into two .pcap files, of about 750 KB (Spain traces) and 1.5 GB (Romania traces). The different sizes of the traces are due to the fact that in Spanish network there are 3292 peers, instead in Romanias network there are 24463 peers (more than seven times bigger than in Spanish network).

In order to simplify our work, these two files have been previously divided into two parts, using the filtering function of Wireshark. The division regards the separation between UDP traffic and TCP traffic. In the end, four files have been

obtained, two of which representing UDP and TCP traces of Spain caught traffic, and other two representing UDP and TCP traces of Romania caught traffic.
This division makes easier the data managing, because it simplifies the characterization in details of the traffic, and, as a consequence, the analyses of all its features and details.

All presented analyses follow, in such a way, several experiments conducted by Thomas Silverston [2] and others researchers. Their purpose was characterizing and understanding the traffic properties and the peer behavior of a peer-to-peer community during a world-scale event.

These experiments are based on the traffic captured on the occasion of *2006 FIFA World Cup*. The traces have been captured from June 9th to July 9th from various kind of P2P IPTV applications (PPlive [19], PPStream, Sopcast, TVants). The capturing of the traces has taken a long time (one month), because the researchers thought that a long period of observation would have been more representative of the actual characteristic of the network and of the users, too.

6.1 Traffic analysis

Our first work aims to classify the traffic captured by Wireshark, in order to understand the nature of the traffic on the p2p network.
Since our first purpose is to evaluate the performance of CoreCast protocol [10], we are interested above all in knowing how much control traffic is there on the network, and then in trying to reduce it. For this reason, as already pointed, the p2p IPTV application chosen is PPlive [19]: this choice is justified by the experiments of Silverston, that reveal that PPlive produces less control traffic among the observed networks [2].

In order to separate signaling traffic and video traffic, it is necessary relying on a **heuristic algorithm**, because the protocol adopted from PPlive application is not open.

The first step of the algorithm consists in the individualization of the **sessions**. *A session is defined as a series of packets with the same IP source address, IP destination address, source port number, destination port number and protocol* [2]. After that, the algorithm has to decide if the individualized session is a signaling session or a data session.

As described in the precedent chapter, the first phase is the data extracting. So, in order to get the data necessary to individualize the sessions, the structure of the TShark commands is the following:

```
tshark -r pplive.tcp.pcap  -Tfields -e ip.src -e ip.dst -e
tcp.srcport -e tcp.dstport -e ip.len > TCP_only.txt
```

Among the commands used, just the input and the output files changes (there are four commands because there are four traces in input; for simplicity, from now on it will be referred just to the structure of the all one, regardless of the input and the output files).

Referring to the explanation given in the fifth chapter, in this command line it is possible to find all the five parameters necessary to individualize a session.

After the data extraction, we can pass to the data managing phase. Here the separation between the control and data traffic is implemented. In order to implement this division, it has been considered that in general data and control traffic have different characteristics:

- *Payload Size (PS)*, because data packets are much bigger than control ones;
- *Delay Constraints*, more stringent for data traffic. In order to evaluate this limits, it has been taking on account as parameter the *Inter Packet Time (IPT*), i.e. the time interval between two packets belonging to the same session .

It is supposed that video sessions are composed by big packets, transmitted in regular and short time intervals; on the contrary control session should be composed by small packets transmitted less frequently, comparing with data ones. In other words, video sessions should have big PS and short IPT, vice versa control session should have smaller PS and longer IPT than data chunks.
Based on this reasoning, the heuristic algorithm works counting the number of packets (hence this parameter will be called n), with a payload bigger than 1200 bytes. So, if $n \geq 10$, the session is considered a data one, otherwise it will be considered a supposed control one. Note that each session is treated like a "monolith", in the sense that it can be considered either the whole data session or the whole control session. Furthermore, some direct analysis on the traces, have revealed that the limit can be decreased to 1000 bytes (hence, this one will be the threshold taken into account for the following experiments).

Using these information deriving from the previous mentioned experiments [2], the traces have been processed by a Perl script, in order to obtain an aggregation of the traffic according to three distinctive features:

1. UDP traffic and TCP traffic, obtained through the previous division of the original trace file in an UDP file and a TCP file;
2. data traffic and control traffic, based on heuristic algorithm explained;
3. upload traffic and download traffic, obtained by comparison with the fix IP address of the capture point (IP address of UPC for spanish traces and IP address of UTCN for romanian traces).

The Perl script implementing this function, works according to the logic represented in figure 6.1:

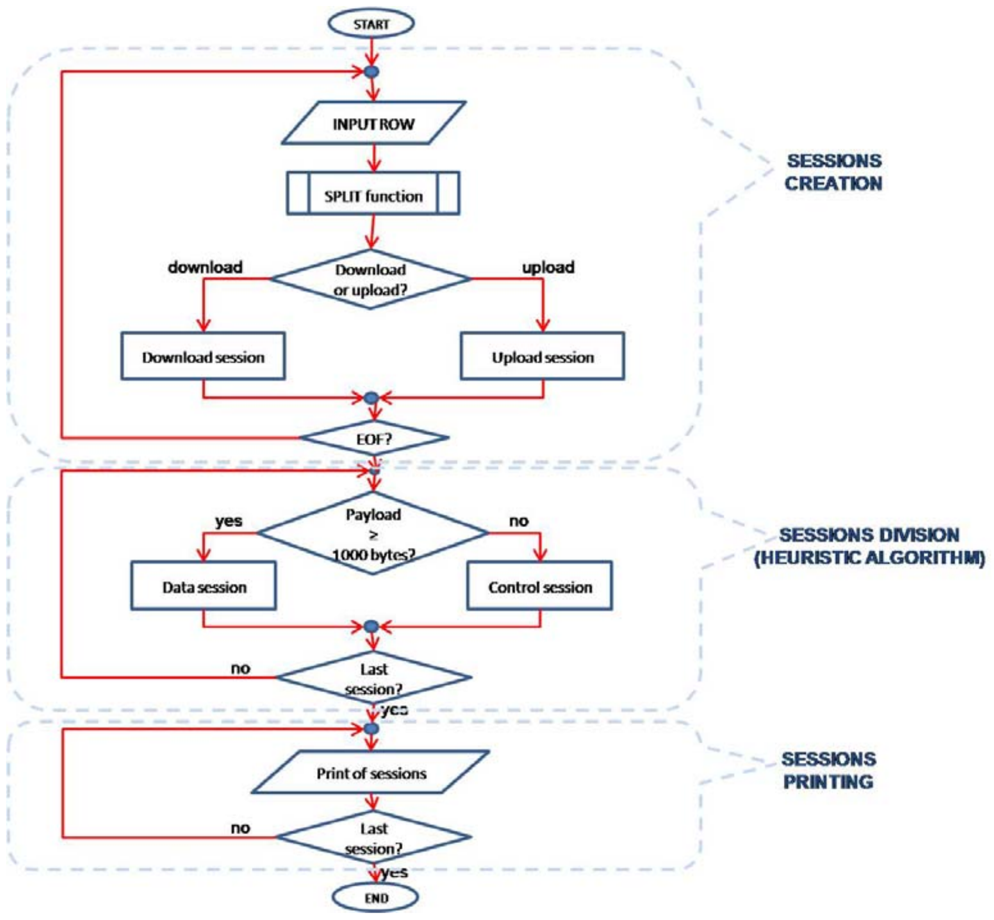


Fig. 6.1: Block diagram representing the logical process of the first Perl script

The algorithm is composed by two main phases. The session creation, during which the data structure (four-dimensional) thought to contain the sessions is instantiated during the reading of the input file row by row. Then, the session division phase deals with the separation of the data sessions and control sessions. In the end, the results are saved into 12 text files, one for each graph that will be drown. (later it will be explained why the graphs are 12).

After this process, it is possible to recognize 8 different categories of traffic:

- video download TCP traffic
- video download UDP traffic
- video upload TCP traffic

- video upload UDP traffic
- control download TCP traffic
- control download UDP traffic
- control upload TCP traffic
- control upload UDP traffic

The last phase of the work is the representation of the results. A proper MATLAB M-file has been constructed: it receives in input the text file from the Perl script, then it returns in output 12 graphs:

- 8 of these refers to the traffic category individualized through the Perl script;
- the other 4 arise from the aggregation of data and control traffic, so they represent the total upload TCP traffic, the total download TCP traffic, the total upload UDP traffic and the total download UDP traffic.

Each of these 12 graphs charts the trend of the CDF (Cumulative Distribution Function) of each node (identified by its IP address) of Spain and Romania traces (two lines) for the particular category of traffic which it refers to. Actually, each of this diagram has been saved in five different formats (.pdf, .png, .fig, .eps, .jpg), in order to allow eventual modifications and uses in different environments.
The following (table 6.1) is a summary of the analyzed traces, divided according to the characteristics of the traffic they represent:

Traffic kind		Number of sessions	Total payload
SPAIN	video download TCP traffic	78	632926384
	video download UDP traffic	46	29244528
	video upload TCP traffic	139	2289918
	video upload UDP traffic	173	17534038
	control download TCP traffic	0	0
	control download UDP traffic	0	0
	control upload TCP traffic	3	8075
	control upload UDP traffic	3170	8301392
ROMANIA	video download TCP traffic	30	47929668
	video download UDP traffic	34	8344177
	video upload TCP traffic	310	1013349
	video upload UDP traffic	308	1302585
	control download TCP traffic	634	464062616
	control download UDP traffic	1008	311543907
	control upload TCP traffic	21071	32567799
	control upload UDP traffic	22823	42198363

Table. 6.1: summary of traces characteristics

The table gives information about the number of sessions, individualized using the heuristic algorithm, and the total payload, obtained just summing the payloads of every session of the same kind.

Following, there are the graphs relative to every class of data, as reported in the table 6.1.

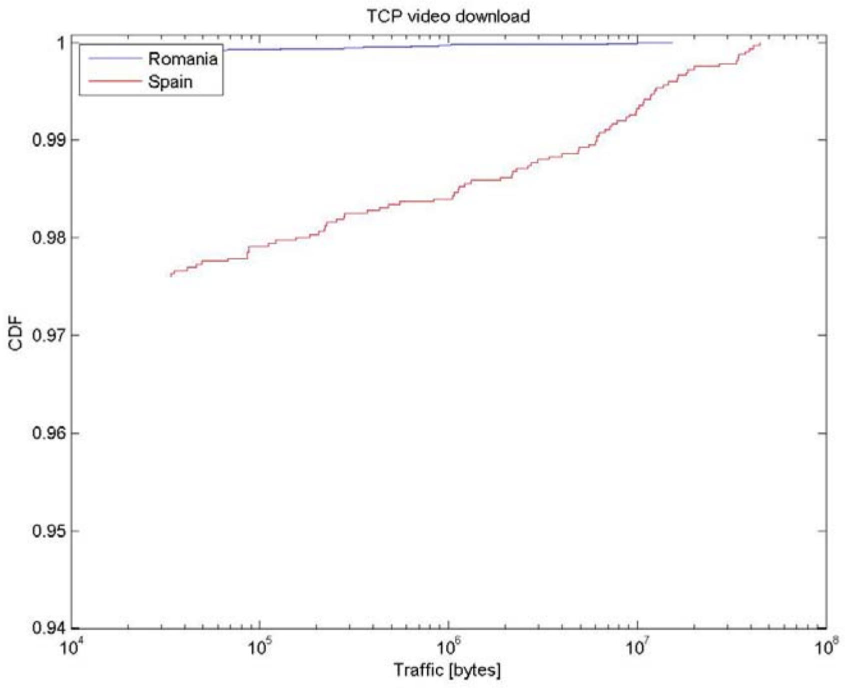


Fig. 6.2: CDF representations of spanish and romanian TCP video download traffic

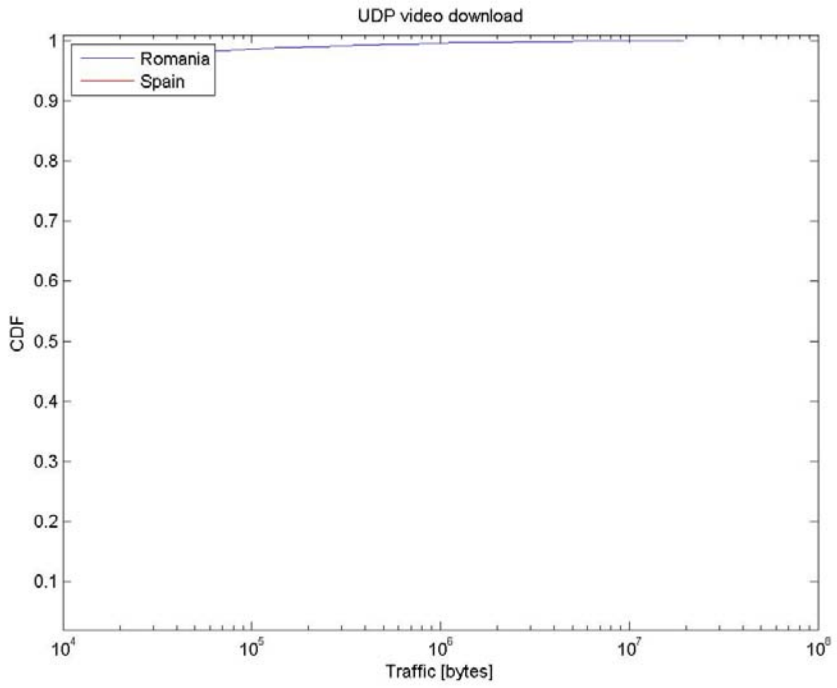


Fig. 6.3: CDF representations of spanish and romanian UDP video download traffic

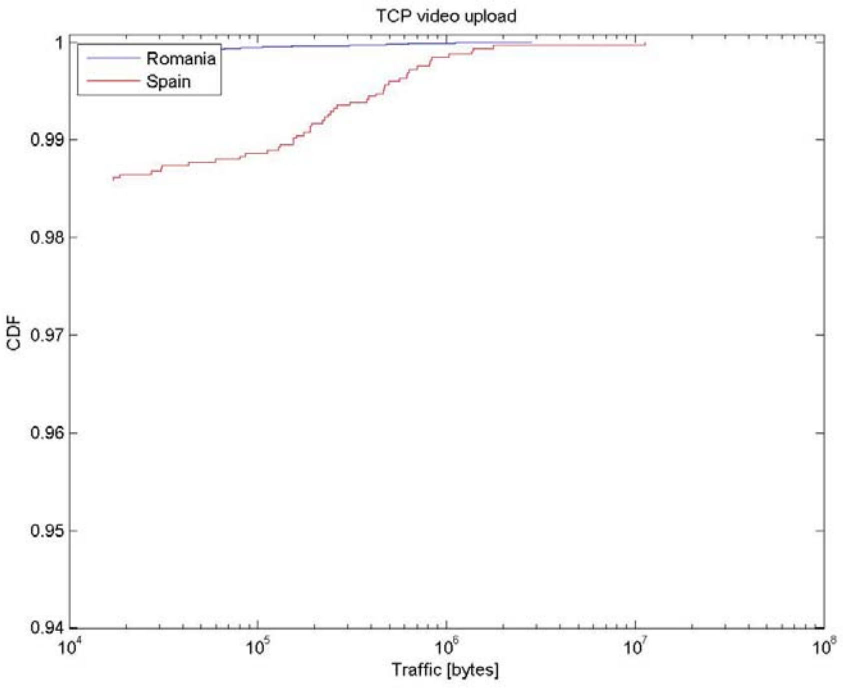


Fig. 6.4: CDF representations of spanish and romanian TCP video upload traffic

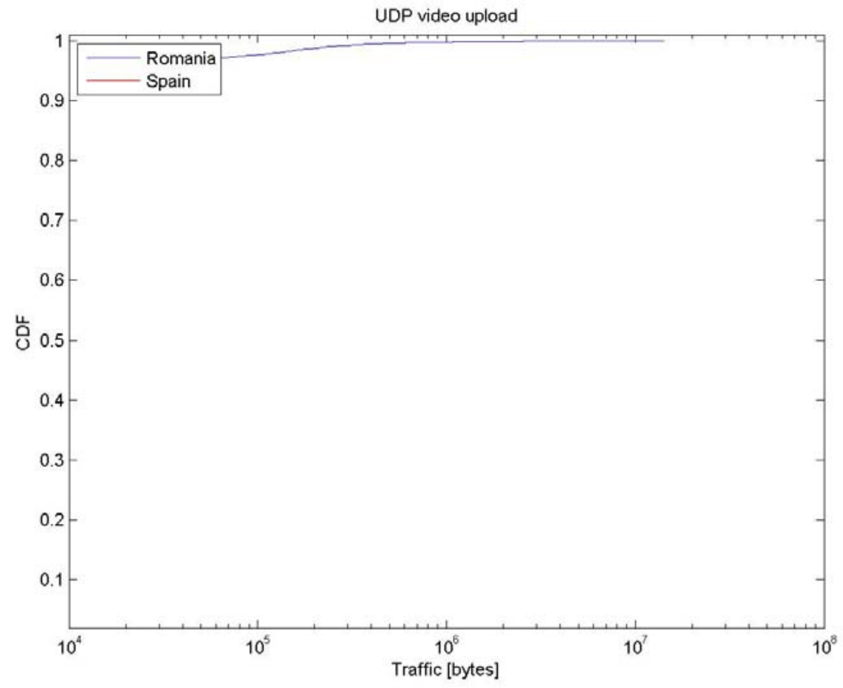


Fig. 6.5: CDF representations of spanish and romanian UDP video upload traffic

These results show that there is no UDP video traffic in Spain traces (figures 6.3 and 6.5). This is probably due to the presence of the firewall, as previously hypothesized.

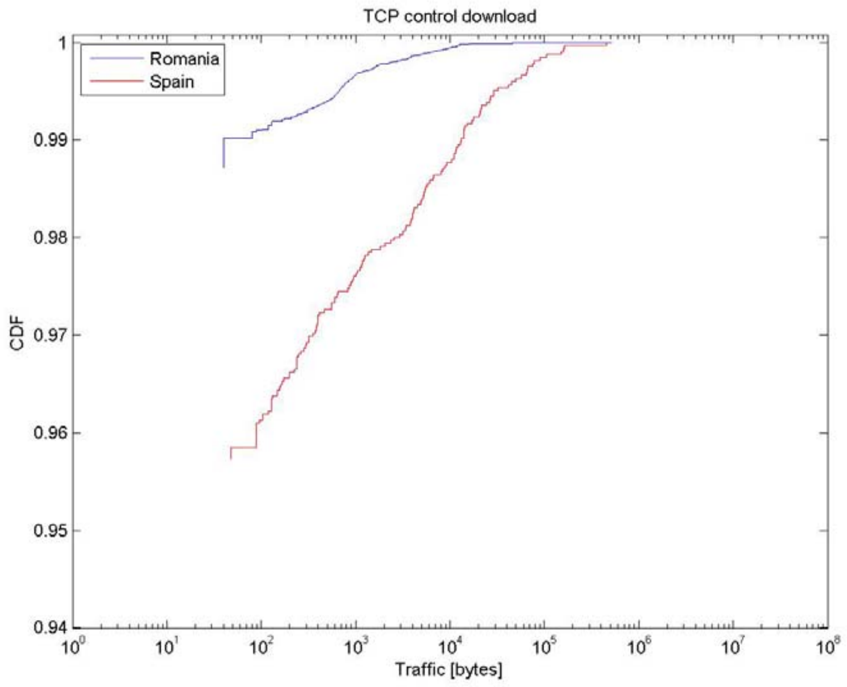


Fig. 6.6: CDF representations of spanish and romanian TCP control download traffic

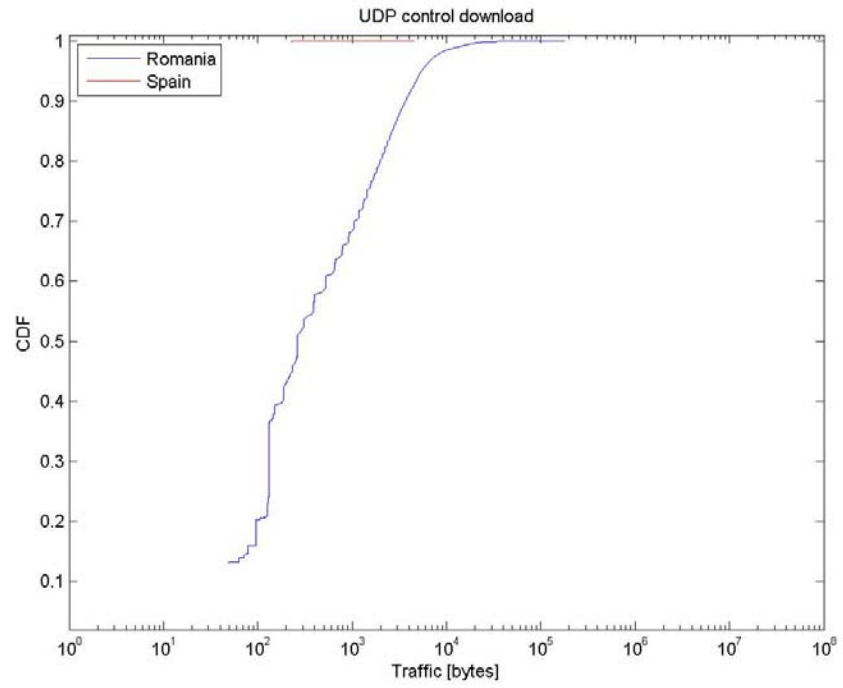


Fig. 6.7: CDF representations of spanish and romanian UDP control download traffic

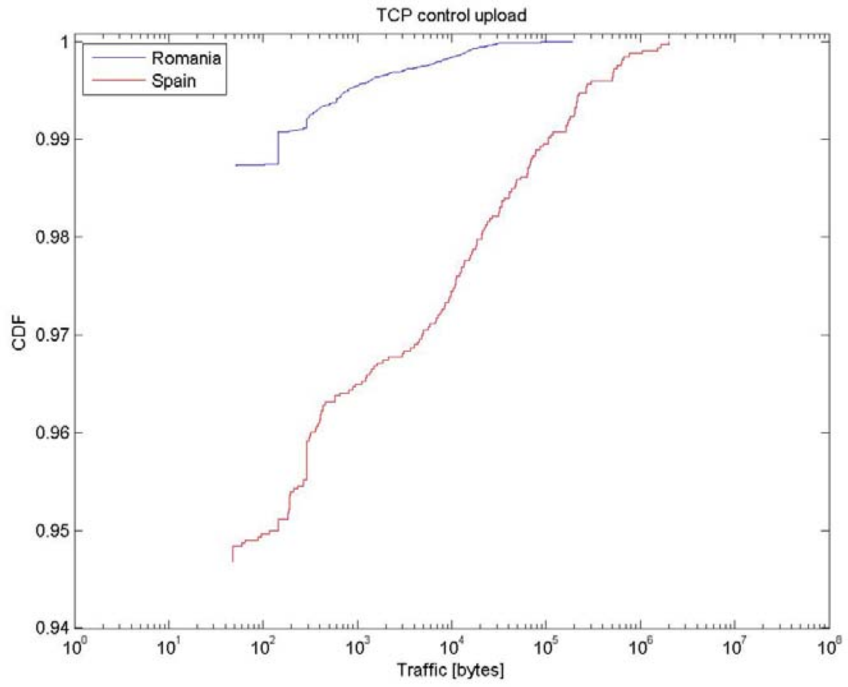


Fig. 6.8: CDF representations of spanish and romanian TCP control upload traffic

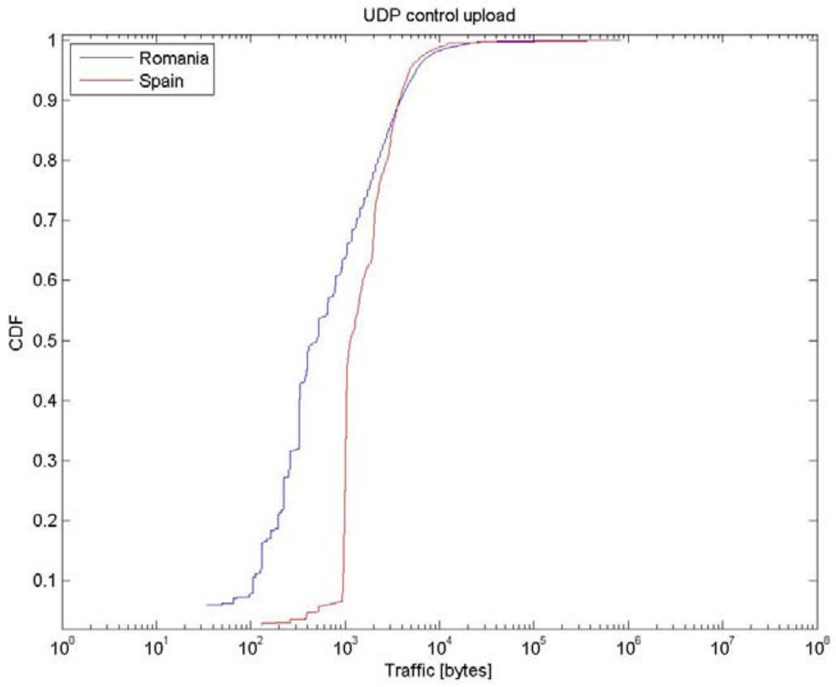


Fig. 6.9: CDF representations of spanish and romanian UDP control upload traffic

As a confirmation of this theory, in the figures 6.7 and 6.9 some UDP control traffic appears. Analyzing these graphs, too, is evident that the control TCP traffic (lower limit 98.5 %) is less distributed than the control UDP one (lower limit 18.5 %): this percentages also confirm the idea that TCP traffic is almost all video type, for this reason it is not so sparse among the peers. Maybe UDP control packets have a payload limited to a certain threshold.

As a conclusion of this experiment, below there are the graphs about total traffic:

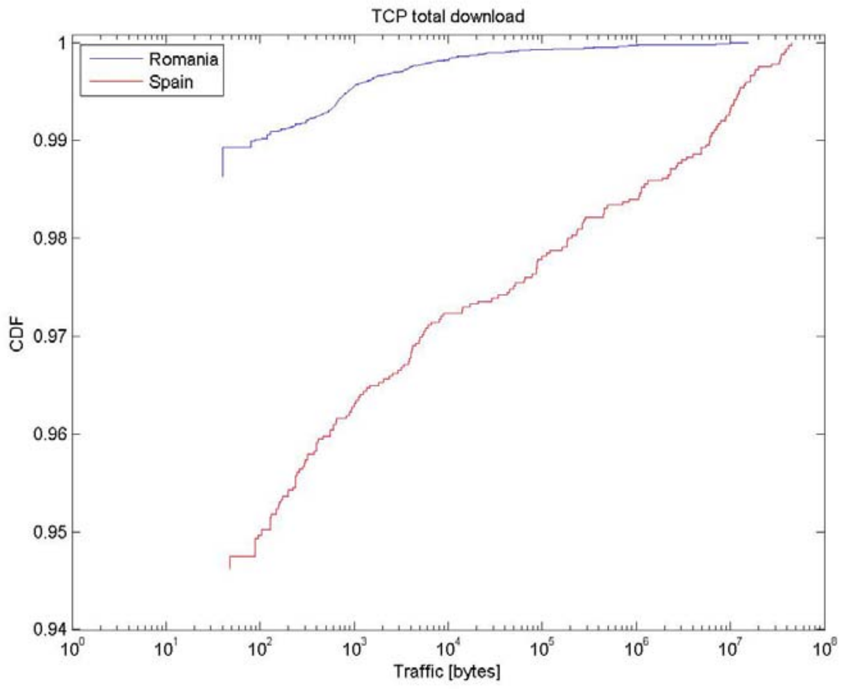


Fig. 6.10: CDF representations of spanish and romanian TCP total download traffic

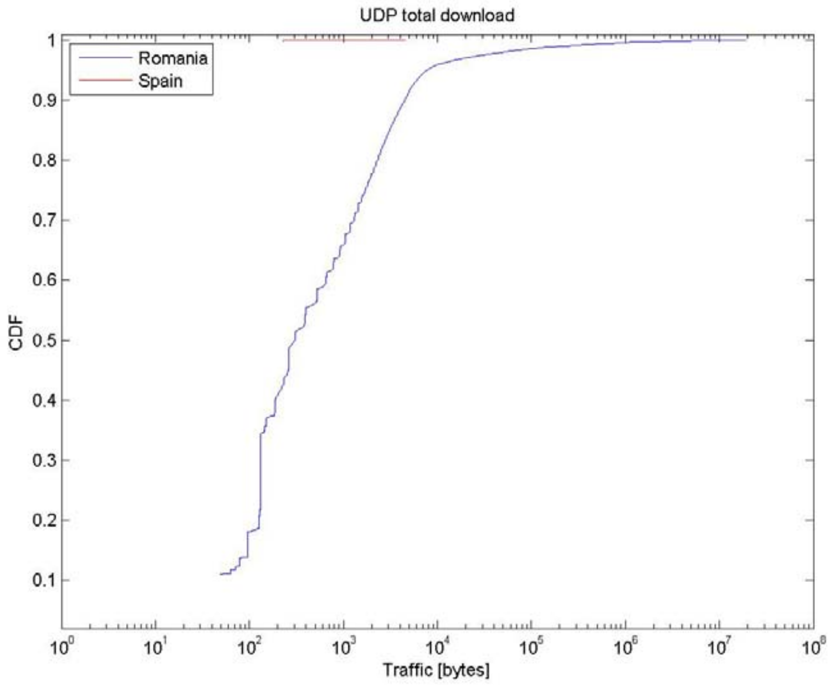


Fig. 6.11: CDF representations of spanish and romanian UDP total download traffic

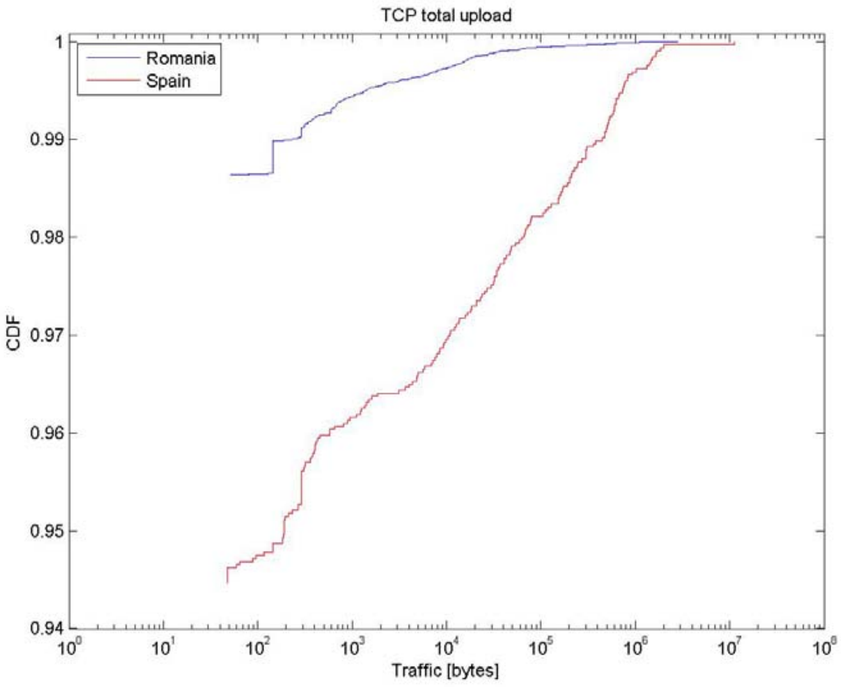


Fig. 6.12: CDF representations of spanish and romanian TCP total upload traffic

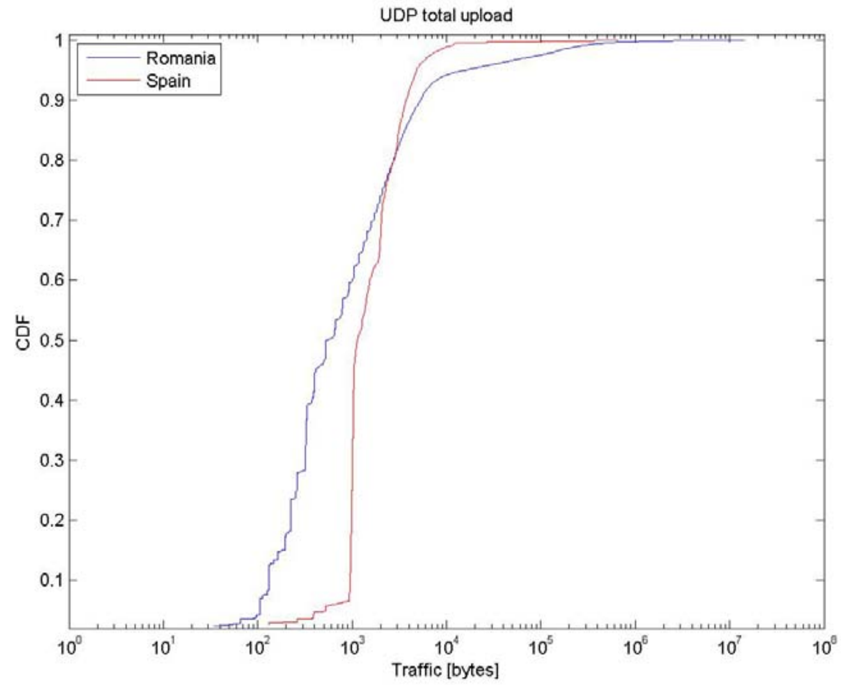


Fig. 6.13: CDF representations of spanish and romanian UDP total upload traffic

Viewing this summarizing four graphs (figures 6.10 - 6.11 - 6.12 - 6.13) it can be said that generally the Romania traffic converges to 1 more quickly than Spain traffic: it means that the traffic is distributed among more peers in Romania than it is in Spain. Furthermore, the download traffic in Spain (figures 6.10 and 6.11) is bigger than the download traffic in Romania (above all UDP); on the contrary the upload traffic is larger in Romania than in Spain.
As shown, the download traffic is created from the interactions among the peers, instead the upload traffic regards the interactions between the capture point and the peers. A possible explanation to this phenomena could be the presence of a firewall into the Spain scenario.

This hypothesis is born out also by the statement of fact that there is more control traffic in download than in upload (figures 6.6 - 6.7 - 6.8 - 6.9): it means that control traffic doesn't concern the traffic between peers, but only the direct connections with the capture point.

Finally, confronting the volume of UDP and TCP traffic in Romania and in Spain scenarios, it has been resulted that there is prevalence of TCP traffic in Spain and prevalence of UDP traffic in Romania. This can be due to the very high quantity of peers (more than eight times bigger than the number of peers in the Spain network) that produce more control traffic among themselves (it is UDP, as already pointed).

6.2 Download/upload balance

Continuing with the reasoning previously described, in this section the theme of download/upload balance is faced. The term balance is defined as the *proportion between the volume of the download traffic and the volume of upload traffic*. This ratio is very useful in order to reveal if the peers use more their download bandwidth, taking the video content directly from the capture point (or

in alternatively, from other peers), than their upload bandwidth, sharing their resource in order to let other peers download from them some video chunks.
This experiment would not have sense if the peers used an asymmetrical bandwidth, but in our case it is significant, because the capture point (both in Romania and in Spain) has a **symmetrical bandwidth** in order to obtain more equal results. The data extracting and the data managing steps have been skipped, because all necessary data are already available from the traffic analysis showed in paragraph 6.1.

So, only the result representation phase is necessary, as a consequence a second M-file has been written. The input data of this one are the same of the previous M-file, but the output consists of **12 graphs** in total: 6 for the Romanian traces and 6 for the Spanish ones. This time, Romania and Spain data are drown separately, because seeing this kind of representation of both in just one graph could be not very intelligible. Like in all experiments, every graph has been saved in five different formats.

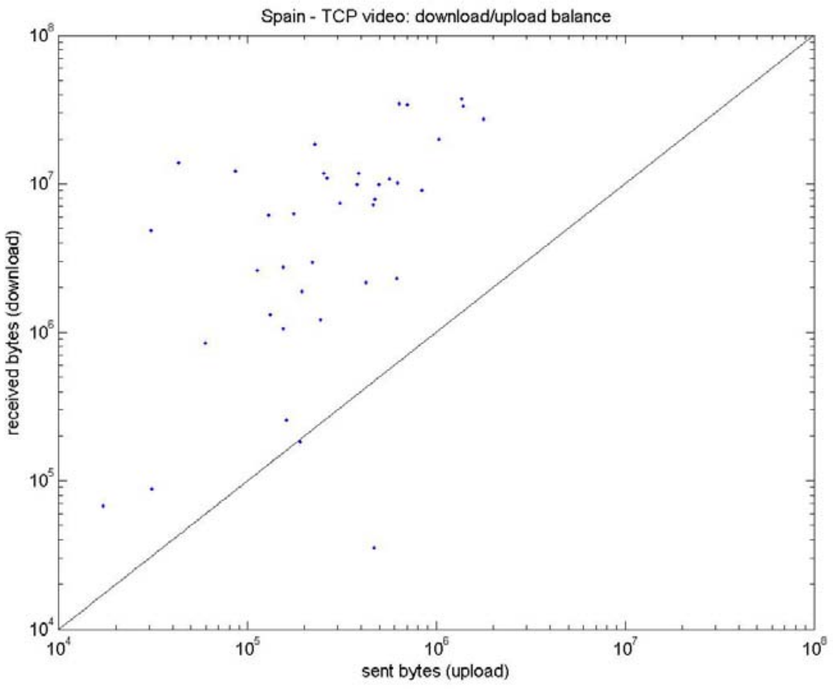


Fig. 6.14: Download/upload balance representation of TCP video traffic in Spain

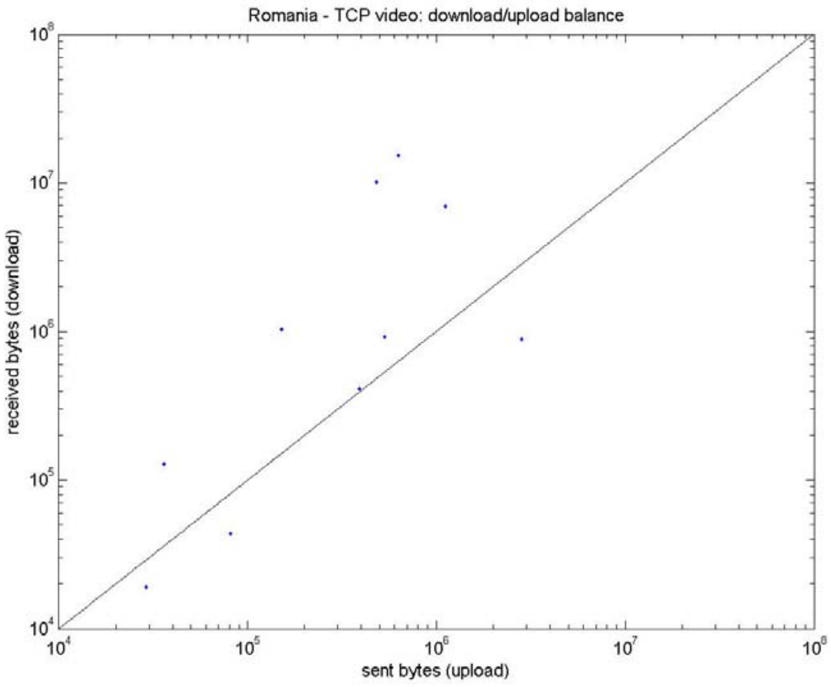


Fig. 6.15: Download/upload balance representation of TCP video traffic in Romania

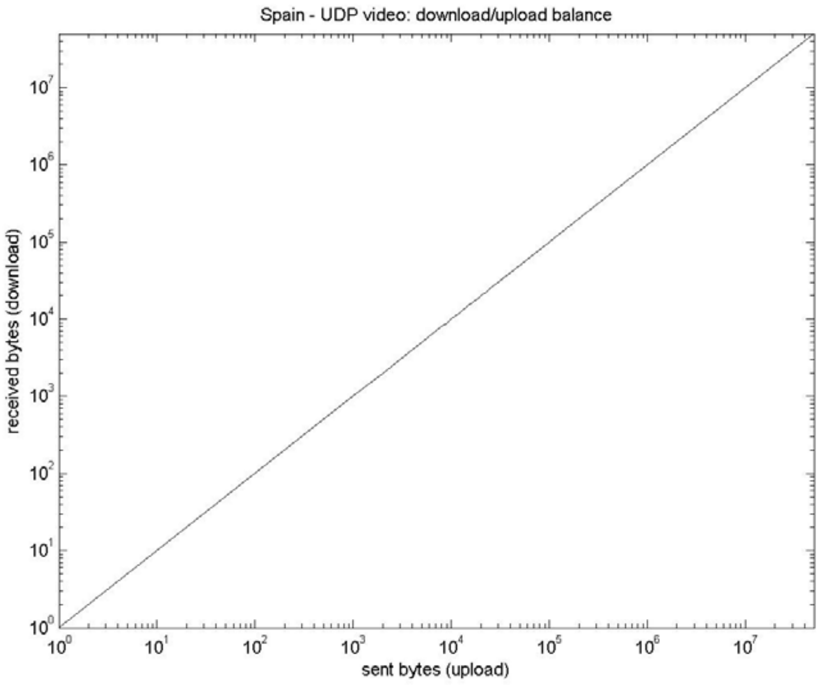


Fig. 6.16: Download/upload balance representation of UDP video traffic in Spain

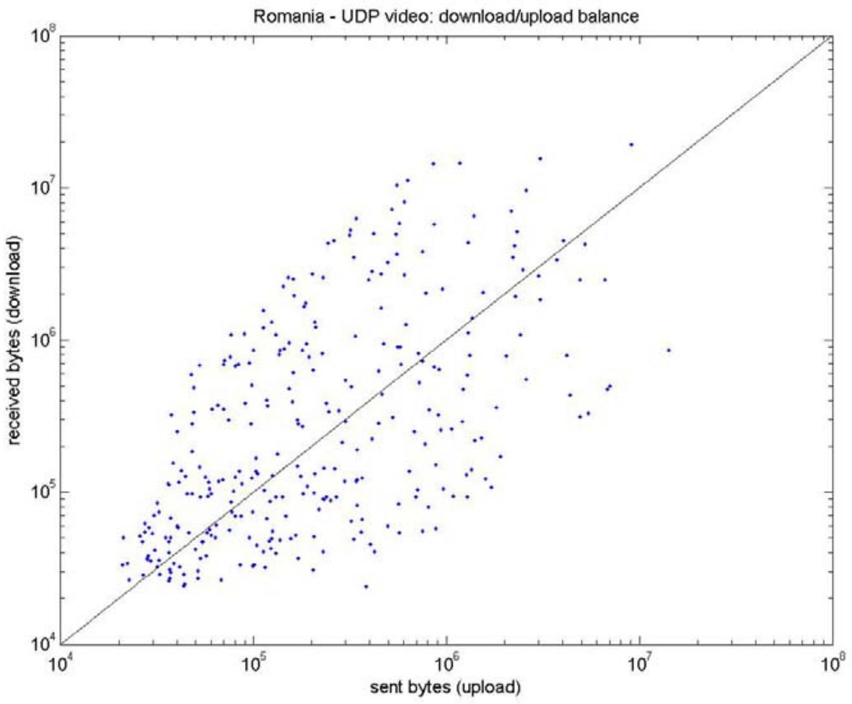


Fig. 6.17: Download/upload balance representation of UDP video traffic in Romania

With referring to the figures 6.14 – 6.15 – 6.15 – 6.17, the axis represents the number of bytes received (y axis) and transmitted (x axis). Every point of the diagram corresponds to a peer. A **bisector line** represents a situation of perfect balancing between upload and download traffic. Of course, it is just an *ideal situation*, as a matter of a fact there are a lot of points quite distant from this line: if the point is upper than the line, it means that it receives more data than it sends, vice versa if the point is under the line. The plot has logarithmic scale, because the considered values cover multiple orders of magnitude.

As to TCP video traffic (figures 6.14 and 6.15), in both networks the download one is more relevant if compared to the upload one. The situation changes analyzing the UDP video traffic graphs: into the graph representing Romania network, (figure 6.17) all points are quite close to the bisector line; it means that the situation is more balanced; instead the UDP data traffic graph referred to the Spain network (figure 6.16) is empty, there are no points . This is because the volume of UDP video traffic in upload (with reference to the figures 6.3 and 6.5) and in download (with reference to the figures 6.2 and 6.4) is too low to be represented with a logarithmic scale like the one used in these graphs.

Thus, it can be deduced that TCP video traffic is more concentrated into the download side (upper than the bisector line), with the meaning that peers do not use a large part of their upload bandwidth, although they have the same capacity than they have in downloading. The situation is more balanced in UDP video traffic, maybe because UDP protocol is not very used to transmit data packets (and this consideration can also explain the situation of the figure 6.16).

The control traffic situation in Spain and Romania networks is represented in the following four figures 6.18 – 6.19 – 6.20 - 6.21:

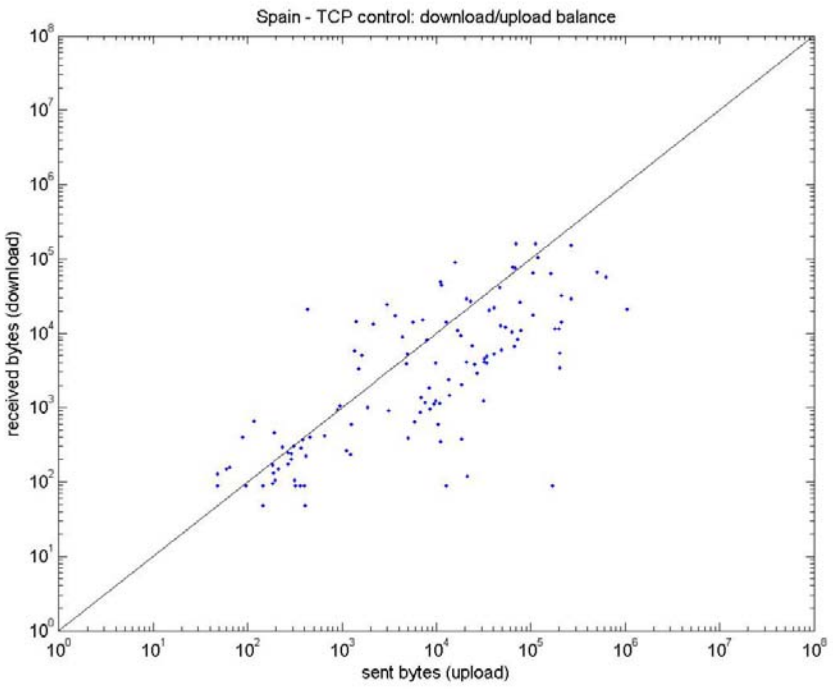


Fig. 6.18: Download/upload balance representation of TCP control traffic in Spain

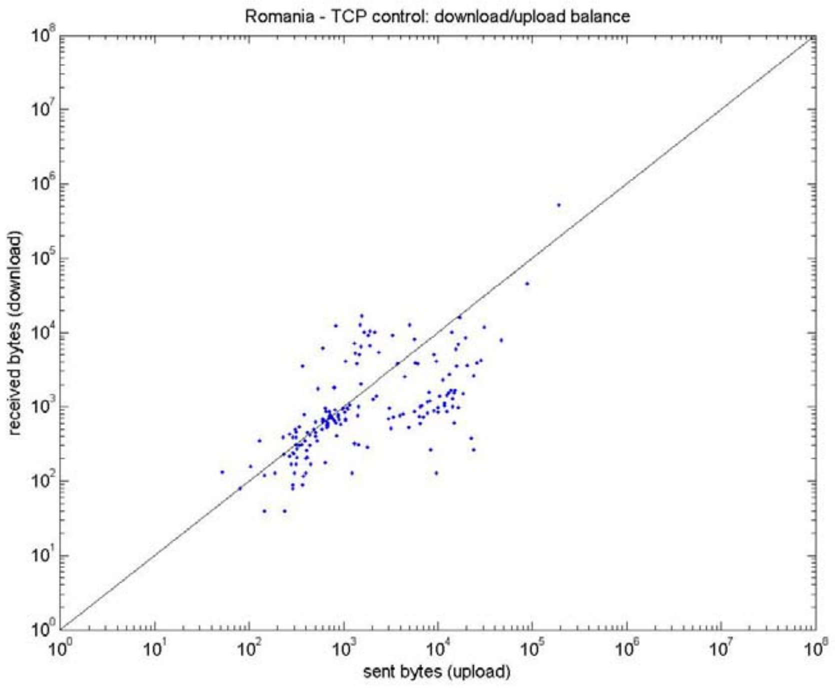


Fig. 6.19: Download/upload balance representation of TCP control traffic in Romania

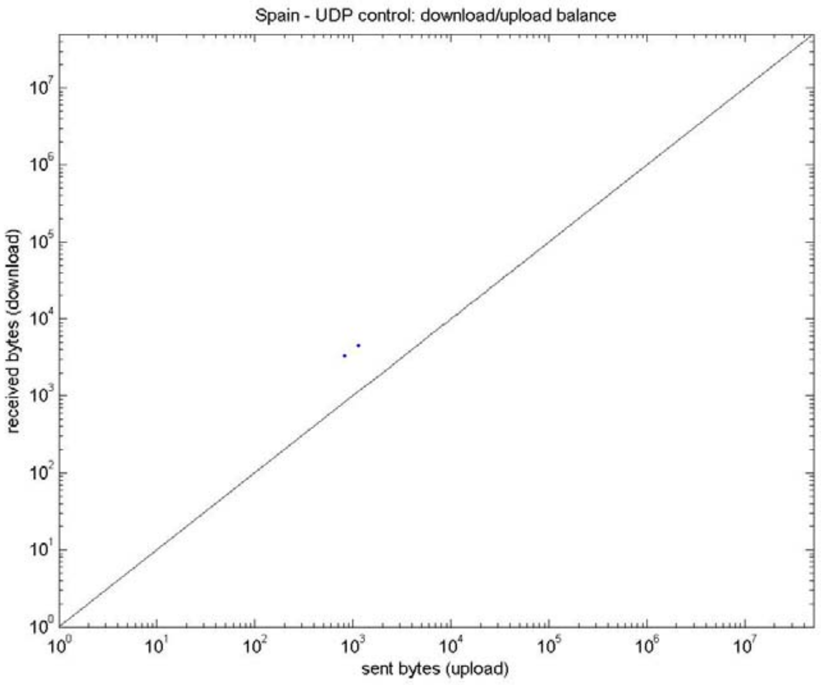


Fig. 6.20: Download/upload balance representation of UDP control traffic in Spain

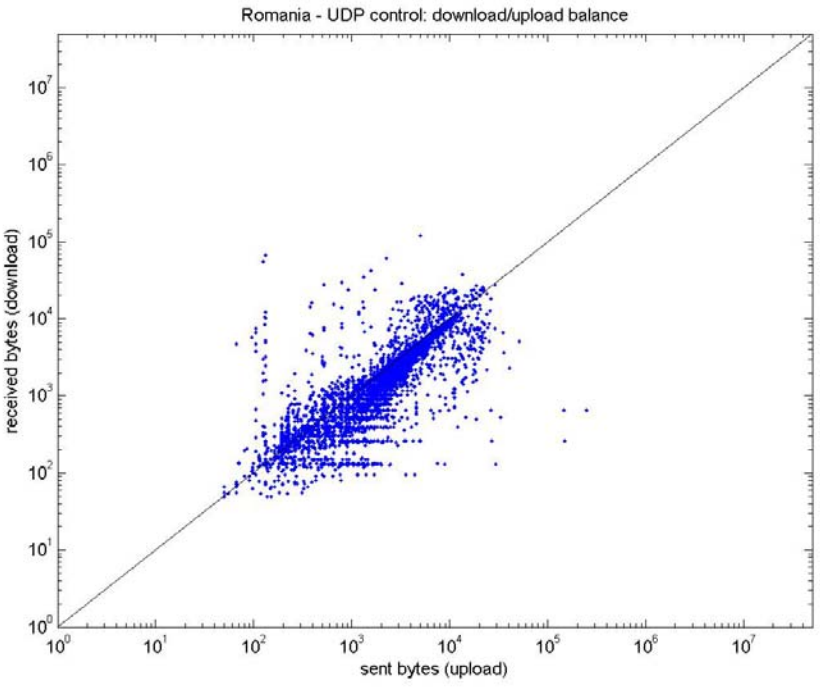


Fig. 6.21: Download/upload balance representation of UDP control traffic in Romania

Analyzing the control traffic, it results evident that the volume of TCP data is much lower than the volume of UDP data, with the exception of the figure 6.20, that shows almost no point. This graph supports the idea that there is a small volume of upload and download UDP traffic into the UPC network, as the figures 6.7 and 6.9 has already revealed. Since the UDP traffic is almost referred to signaling, the lower number of peers in this network can be a plausible cause of this phenomena. Anyhow, the general trend of these graphs confirms that there is more traffic UDP than TCP in signalizing. The trend is more emphasized comparing the figures 6.19 to the figure 6.21: in the latter a large amount of points compose a sort of cloud into the graph. Probably, it is caused by the large amount of peers of the UTCN network, that, interacting among them, create a big volume of control traffic.

It is also interesting considering the distribution of the points: the graphs referred to the Romania scenario (figures 6.19 and 6.21) have the majority of the points concentrated in the lower-left region of the diagram (sent and received payloads smaller than 10^4 bytes). It just confirms that control payloads are smaller than the data payloads, as a matter of a fact the figures 6.15 and 6.17 show a concentration of points in the higher-right part of the diagram. This characteristic is more accentuated in the Romania scenario because of the bigger amount of peers. Finally, the graphs relative to the aggregation of video and control traffic are shown in the following figures:

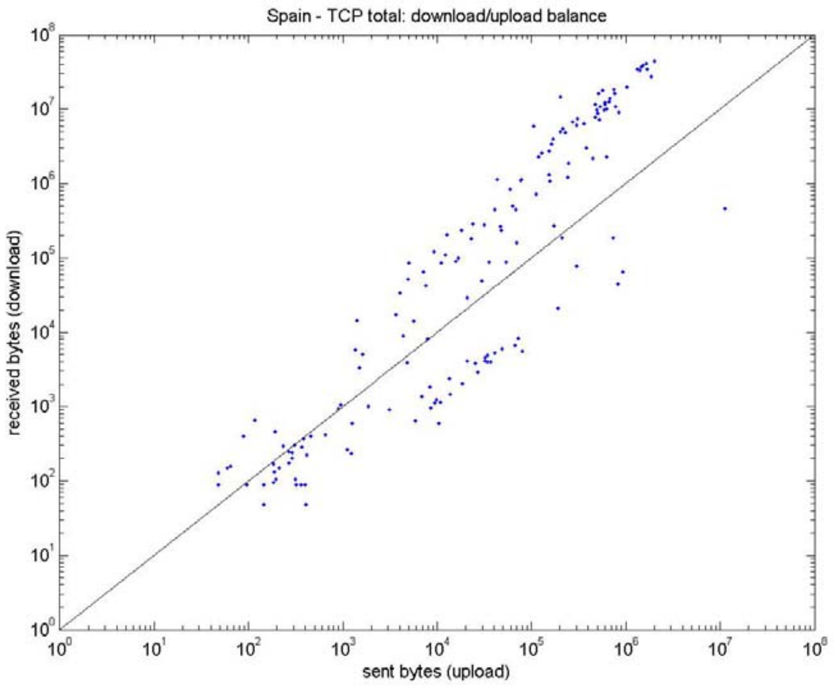


Fig. 6.22: Download/upload balance representation of TCP total traffic in Spain

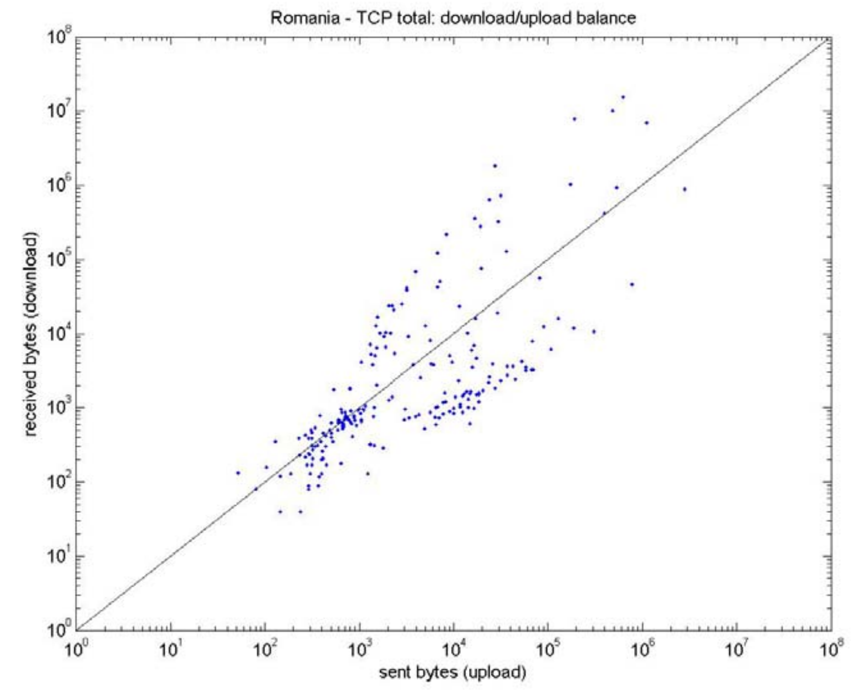


Fig. 6.23: Download/upload balance representation of TCP total traffic in Romania

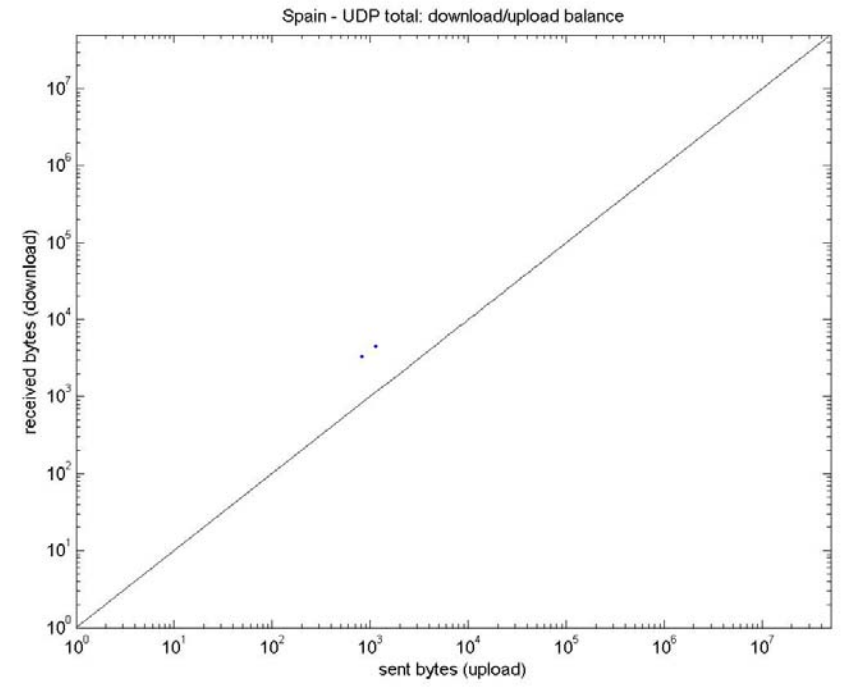


Fig. 6.24: Download/upload balance representation of UDP total traffic in Spain

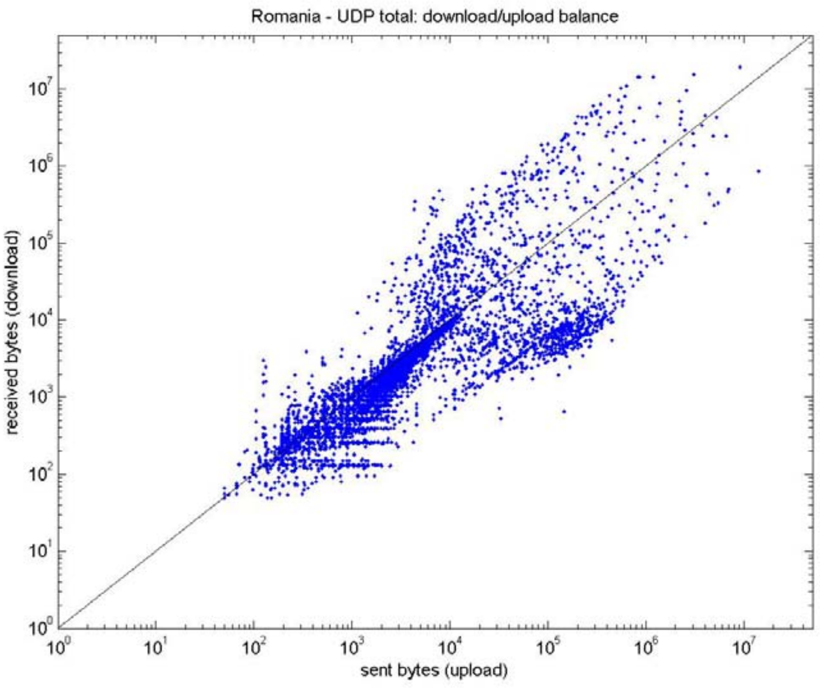


Fig. 6.25: Download/upload balance representation of UDP total traffic in Romania

With the exception of the figure 6.24, that represents a logic consequence derived from the sum of the diagrams of figures 6.16 and 6.20, the general distribution of the peers is not very sparse, so the major part of the points is quite close to the bisector line. This could mean that the PPlive application tries to maintain a certain compensation between download and upload traffic.

As a conclusion, we can affirm that, generally, the download/upload balance is quite good in all the considered scenarios.

6.3 Evolution of network population

In this section the network population has been evaluated, considering its evolution minute by minute. The focus is on the time, because it could be interesting to understand how many users per minute are connected to the peer-to-peer network watching the video. The number of users, corresponds to the number of different IP addresses appeared into the traces. Since in our experiment the video content transmitted is the Michael Jackson funerals one, it can be expected that a lot of peers have been connecting for the whole duration of the event.

In order to get these information, it is necessary extracting from the traces some data about the time. In this case, the time is represented by the **timestamp** of each trace. The timestamp is defined as the *time interval (relative time) elapsed since the receiving of the first trace of the video content until the arrival of current trace*.

Since during the previous experiments no data time have been involved, in the first phase, previous called data extracting, another Linux shell command of TShark has been used:

```
tshark -r pplive.florin.pcap  -Tfields -e ip.src -e ip.dst -
e frame.time_relative > romania_timestamp.txt
```

Actually the commands are two, that the first extracts the information from Spanish network's traces, the second (shown above) extracts the information from Romanian network's traces. The required fields allowing the processing of data in the second phase are the IP source address (`ip.src`), the IP destination address (`ip.dst`) and the timestamp (`frame.time_relative`). The former is expressed in seconds through a real number with nine decimal digits.

Finally, the extracted results are redirected to the text file `romania_timestamp.txt`.

At this juncture, all data are available and the data managing phase can start. It regards three specific operations: the *loading of the data* in an opportune data structure, the *IP addresses clustering in every minute* and the *peers counting* in every minute. Two Perl script are used, one implementing the first function and another implementing the others two functions in cascade. Anyway, the data flow logic can be represented by a single block datagram (figure 6.34).

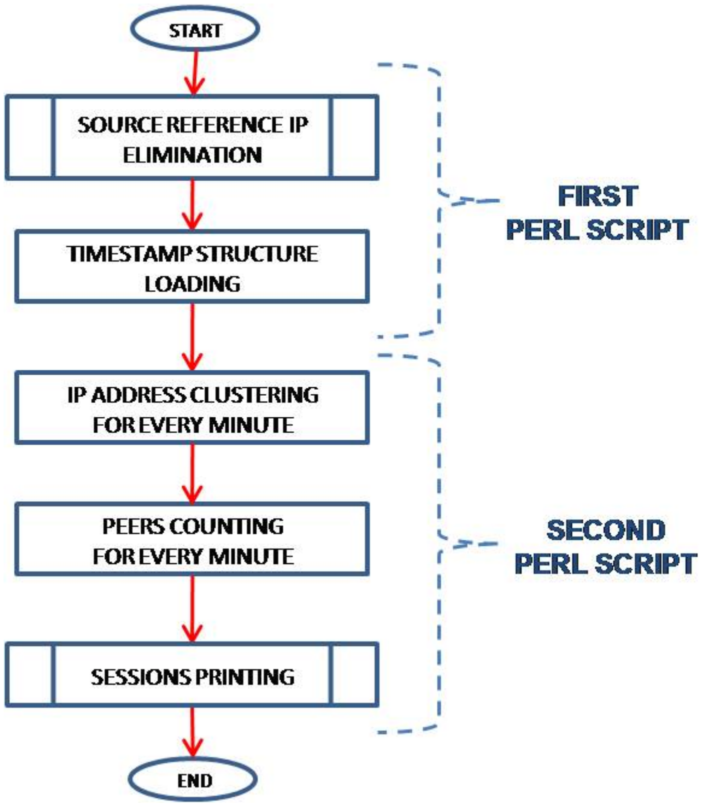

START
SOURCE REFERENCE IP ELIMINATION
TIMESTAMP STRUCTURE LOADING
IP ADDRESS CLUSTERING FOR EVERY MINUTE
PEERS COUNTING FOR EVERY MINUTE
SESSIONS PRINTING
END
FIRST PERL SCRIPT
SECOND PERL SCRIPT

Fig. 6.34: Block diagram representing the logical process of the fourth and the fifth Perl scripts implemented to obtain the evolution of peers population in the network minute by minute

The first step of the diagram is constituted by the same function used in the third Perl script (with reference to the figure 6.29), so here it is simply reported in a single function block (code reusing). Afterward, the timestamp structure is loaded without the reference IP address of the capture point.

Now, the control passes to the second Perl script. First, the peers are clustered according to the minute of belonging, in order to convert the data from a seconds granularity to a minutes granularity.
This discretization is justified by the fact that the peers joins and leaves the network in every second. This dynamicity rends an analysis, considering the second as minimum unit of measure not very significant. It is better counting all the different IP addresses appeared during a minute of observation. In the end, the session printing function is adopted (like in all the other analyses done), in order to save the results in two text file (one for each network).

Subsequently, the result representation phase starts through the editing of an M-file. The data are represented in a single graph, including the Spanish and the Romanian results (in order red line and blue line in figure 6.35).

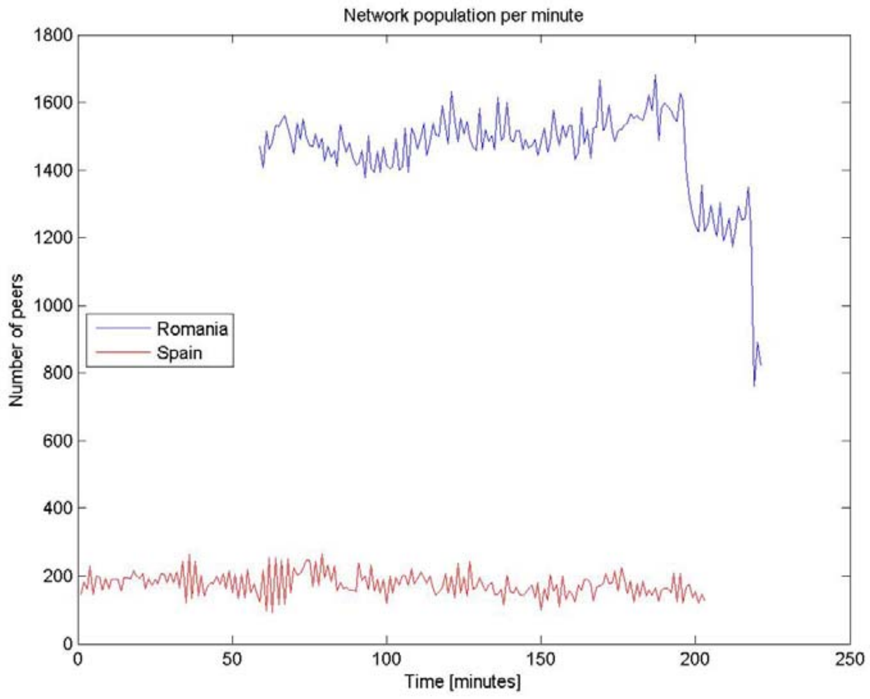


Fig. 6.35: representation of the evolution of network population minute by minute

The plot is realized using normal scalar scale on both axes. Note that the two lines start and end in different minutes, because the streaming of the event covers different time intervals. The Spain traces are captured exactly 58 minutes before (since about 18.00 h considering the BST time zone) than the Romania traces. Then, since the minute 59 until the minute 203, both networks are active (the main part of the Michael Jackson funerals covers this lapse of time, as a matter of a fact the event started at 19.00 h in BST). Afterward, since the 204-th minute until the 221-th minute, only the Romanian network works.

The general trend of the two lines is quite regular, but the red line looks more linear than the blue one. As expected, because of the large spread between the total number of peers of the two networks, the average number of different peers counting in every minute into the Romania network (about 1400) is bigger than in Spain network (about 200 different peers).
It's significant to see that only a low percentage (about 10%) of the peers is active at the same time.

6.4 Ports' utilization

Studying the traces, it has been noted some recursive characters, such as the amount of packet payloads or the number of port used. The former could be an interesting clue in order to understand the peers behavior or to catch some particular feature of the network traffic.
In this experiment, too, the data extraction phase is useless, because the information about both UDP and TCP port number, are already present in the output files obtained after the first analysis (with reference to the paragraph 6.2).
So, it can be left out in this dissertation, passing to the data managing phase. The aim of this evaluation is counting the occurrences of each port, i.e. the times a certain port is used. The figure 6.29 shows the block datagram scheme of the Perl script implemented to achieve the requested counting:

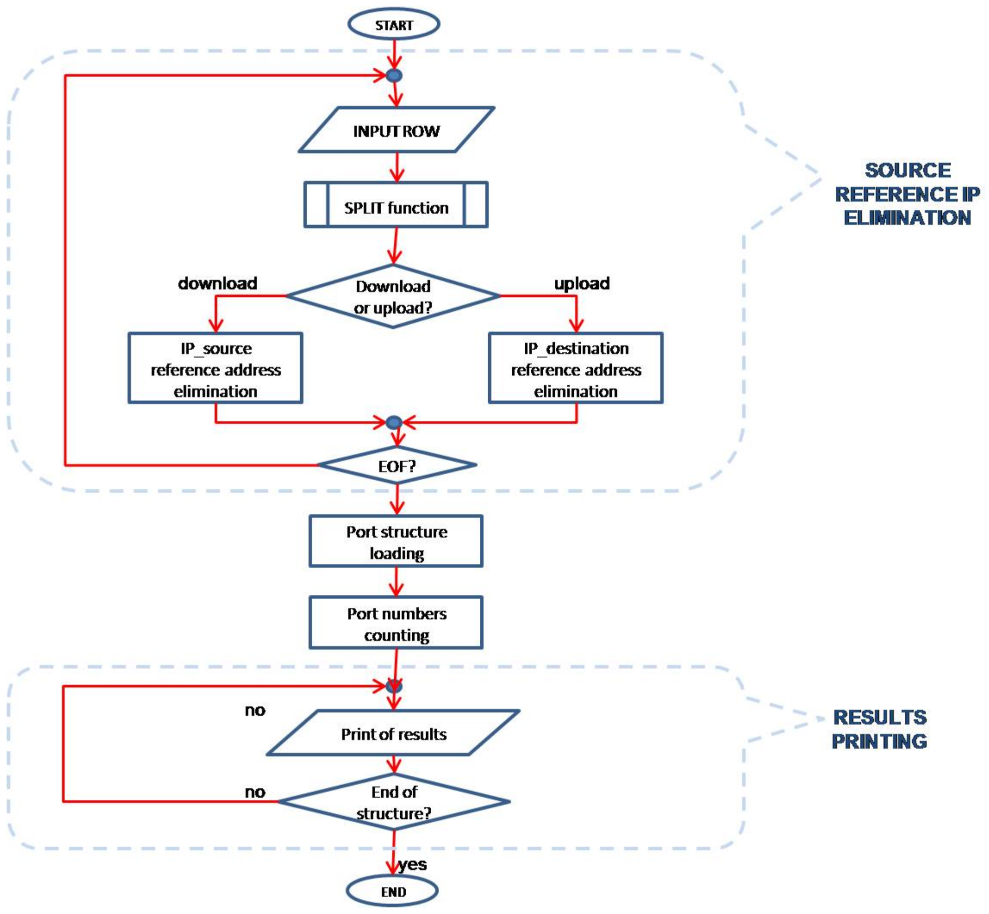


Fig. 6.29: Block diagram representing the logical process of the third Perl script implemented to obtain the information about ports utilization

The program is essentially composed of two parts. The first one, the **reference IP elimination**, has as objective the deleting of the IP number of the video capture point, that can be the IP address of the UPC machine (147.83.130.144) or of the UTCN machine (192.168.1.101) playing as source into the relative networks. This apparently useless operation makes simpler the managing of data in the second phase of the program, because it allows the managing of the data using just one data structure. Afterwards, in the **ports counting** phase, making a simple control (pattern matching based) on the mentioned data structure, the occurrences of every port number has been counted. As usual, finally the obtained results are printed (**session printing** phase). In this case the output consists of four text files, two for the source port counting and two for the destination port counting (for Romania and Spain scenarios).

Passing to the results representation phase, also in this analysis, a new M-file has been written: it accepts in input the four text files created by the Perl script and returns in output as many graphs.

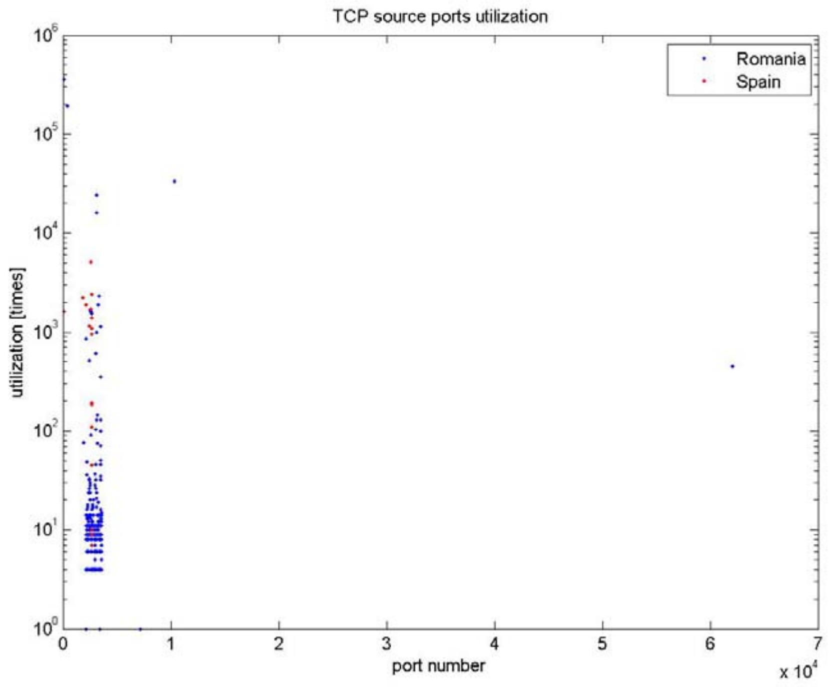


Fig. 6.30: distribution of TCP source port numbers used in Spain and Romania networks

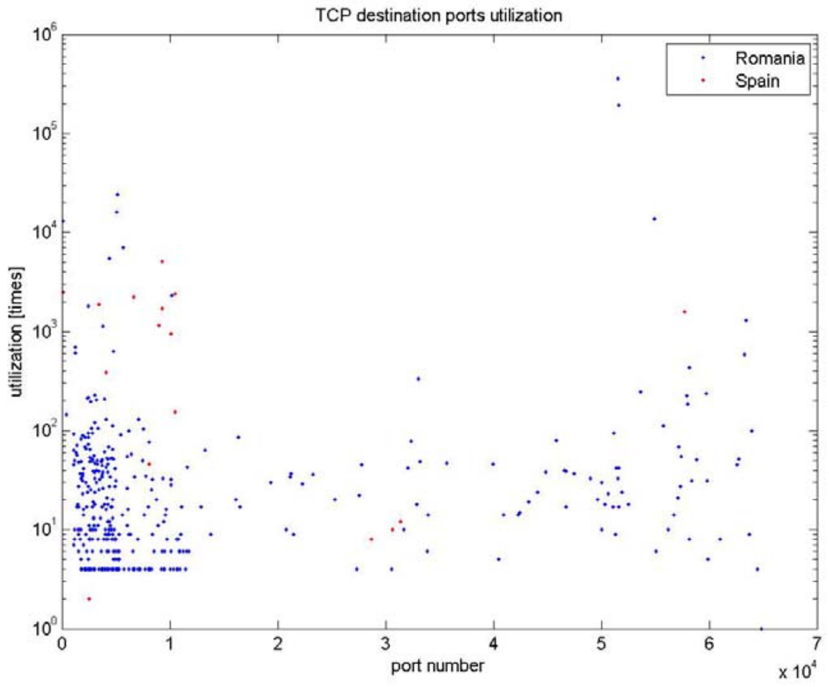


Fig. 6.31: distribution of TCP destination port numbers used in Spain and Romania networks

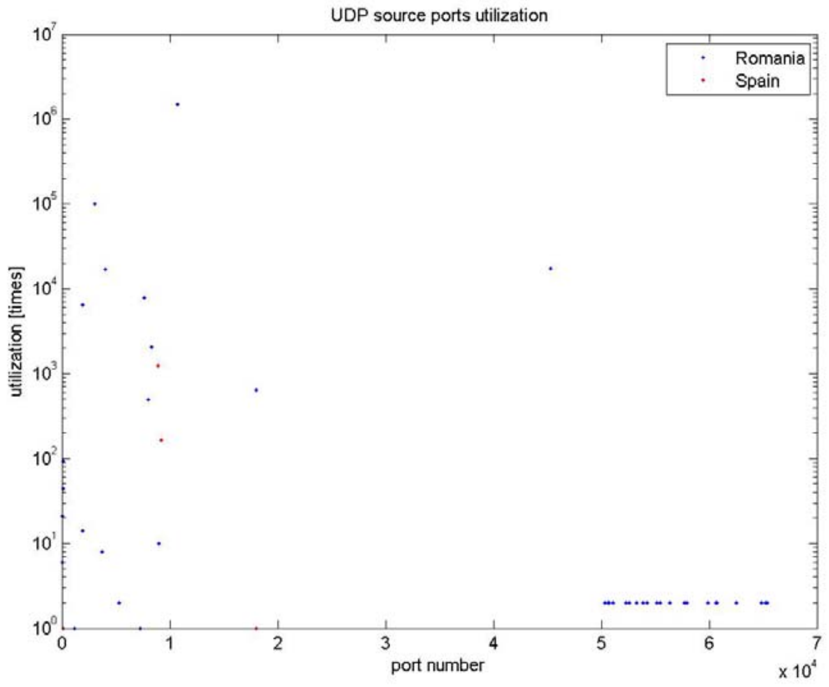


Fig. 6.32: distribution of UDP source port numbers used in Spain and Romania networks

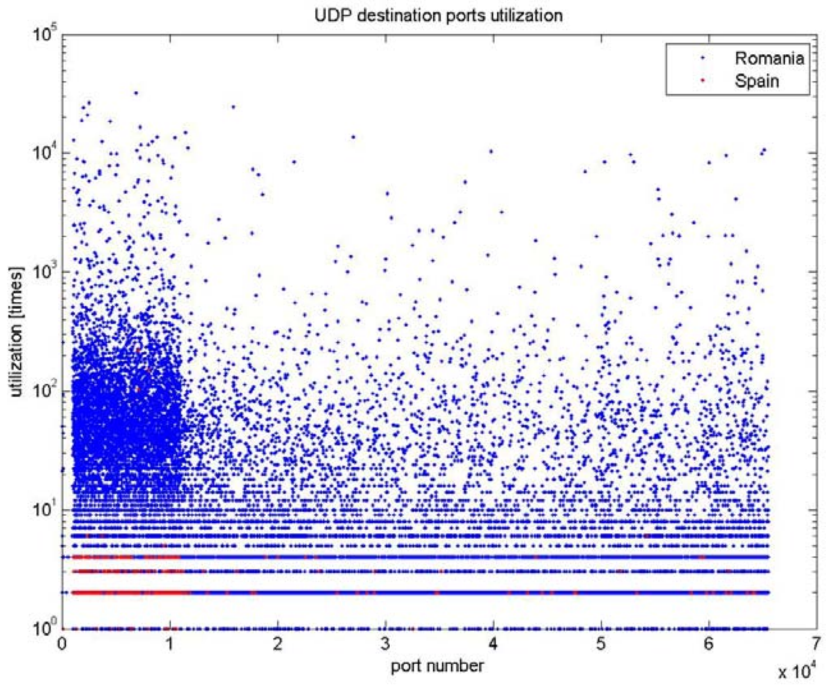


Fig. 6.33: distribution of UDP destination port numbers used in Spain and Romania networks

The plot has logarithmic scale on y-axis because the considered values range across several orders of magnitude. The data of Romania and Spain scenarios are jointly represented into the same graph.

At first sight, it can be noted that the destination ports distribution (figures 6.31 and 6.33) has a more sparse behavior compared with the source ports distribution one (figures 6.30 and 6.32). The deepest confront could be done with the number of UDP source ports used in Spanish scenario (4 port numbers: `8898, 9224, 18010, 138`) and the number of UDP destination ports used in Romanian scenario (22425 different port numbers).

Analyzing the graphs in details, some relevant data has been discovered:

- the most used TCP destination ports into the Romanian networks are bound into a range from number 1066 to 11529;
- the most used UDP destination ports into the Romanian networks are bound into a range from number 1038 to 10654;

- the most used UDP destination ports into the Spanish networks are bound into a range from number 1122 to 10048.

These quantitative data, along with the consideration that there are more used destination ports in UDP traffic in both networks, let reach the conclusion that, *generally, the ports are elected randomly*. Besides, the Romania situation presents always a sparser port numbers distribution than in Spain: this difference can be due to the fact that there are more nodes in the Romanian network, so the random behavior with which the ports are selected is more marked.

Even if some results are interesting and curious, the general random nature of the ports utilization distribution (as already expected) induces to not consider this parameter very indicative and reliable to identify the properties of the PPlive network and its traffic.

6.5 Inter and intra-domain traffic

Since the main aim of these researches is the evaluation of the performance of CoreCast protocol in reducing the inter-domain traffic, it is necessary studying a method to further characterize the captured traces, dividing them in intra-domain and inter-domain traces.
In order to implement this separation, it is necessary knowing the **AS (Autonomous System),** which every IP address belongs to. The belonging to an AS is symbolized through an **AS number**.
Of course, the AS number of the UPC and the UTCN are already known. In order to obtain other nodes AS number, a particular Perl script has been used. It takes in input the IP address, and looks for the correspondenting AS number, finally gives in output the list of the IP address along with the relative AS number in a two columns text file.

After obtaining the AS number of all peers, another Perl script is necessary to implement the division between the intra-domain and inter-domain traffic, also maintaining the characterizations already reached in the first experiment (paragraph 6.1). Note that the aim is *adding information to the traces, without touching the information already got*. It means that after this experiment, the traffic can be described from four points of view:

1. **TCP and TCP traffic**
2. **download and upload traffic**
3. **video and control traffic**
4. **intra-domain and inter-domain traffic**

The algorithm implemented through this second Perl Script is based on the first one used, and just another function is added. The following block diagram (in figure 6.26) explains in details its functionality:

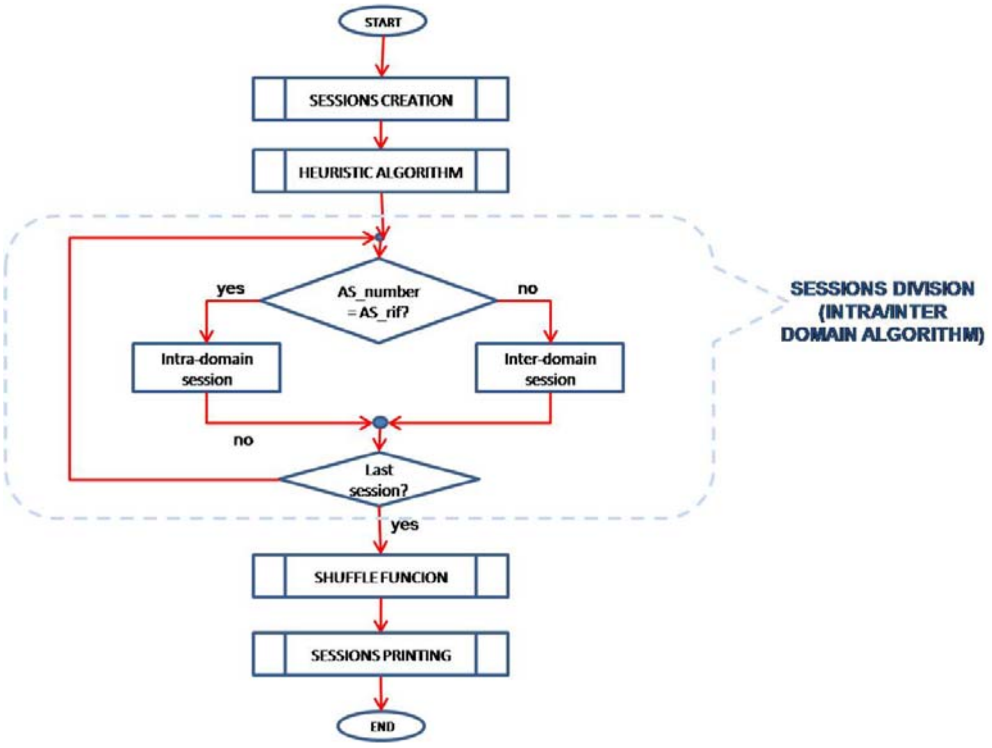


Fig. 6.26: Block diagram representing the logical process of the second Perl script implemented to obtain the division between intra-domain and inter-domain traffic

As expected, the block diagram in figure 6.26 is quite similar to the figure 6.1 one. The session creation and the session printing phases are exactly the same in both algorithms (reusing of code). So a single block has been drawn as substitution for each of these functions, in order to make the representation more compact. The session division phase in this case is composed by three sub-steps: the heuristic algorithm, already used in the first script, then the *intra/inter-domain discrimination algorithm* and the *shuffle function* are added in cascade.

In the second sub-step, the traffic is divided into intra-domain and inter-domain parts through a **pattern matching method**, i.e. comparing the AS number of the peer with the AS number of the capture point node (UPC for Spain network and UTCN for the Romania network).

Actually, the script ends in this point, and the results are saved in a text file. Later on, another Perl script loads this text file row by row and starts the last part of the session division. Since the separation of the Perl script into two different is due just to memory limits, and to render the computation faster, from a logical point of view it can be considered like a single script.

In order to run a realistic simulation with intra end inter domain traffic, a "realistic" traffic matrix from the real trace has to be constructed.

Now it is necessary considering that the differentiation between intra-domain and inter-domain traffic needs another information besides the AS number of every peer: the *amount of directly exchanged data among the peers*.

Since the available traces only give information concerning the traffic from the network point of view , i.e. the capture point one, it is necessary another heuristic algorithm in order to obtain the data regarding every peers (latter information required).

Basing our implementation on the hypothesis that *the distribution of the traffic is equal in each node of the network* [2], changing the order of the elements of this distribution, the final result does not change .

In term of Perl code, all data referred to the traffic between the capture point and each other peer of the network is contained in an array structure (figure 6.27).

	Peer #1	Peer #2	Peer #3	…	Peer #i	…	Peer #N-1	Peer #N
Capture point	500	65	1020	…	456	…	89	1230

Fig. 6.27: Structure of a vector in Perl code, containing the data relative to the traffic exchanged between the capture point and every peer of the network

In figure 6.27 N indicates the number of peers into the network; the numbers into the second row of the picture refer to the total number of bytes exchanged between the capture point and the i-th peer (where $0< i \leq N$ and i is an integer number).

There are 8 vectors like this one into the Perl code, one for each category of traffic:

- TCP upload video vector
- TCP upload control vector
- TCP download video vector
- TCP download control vector
- UDP upload video vector
- UDP upload control vector
- UDP download video vector
- UDP download control vector

So, during the third sub-step of the session division, every vector is passed in input to a **shuffle function** (`fisher_yates_shuffle`). This function shuffles the vector randomly, then gives in output as results another shuffled vector. Every vector is shuffled for N times, so in the end 8 NxN matrixes are obtained (more details and the implementation of this function can be found into the appendix A).

After running the Perl script, the necessary data in order to compose some representations are available. From the point of view of inter-domain and intra-domain traffic, the interest does not concern all of traffic types, but just the **TCP video upload traffic**. The choice is motivated by the fact that this is the only kind

of traffic referred to the utilization of the peers bandwidth in transmission of video chunks among themselves, without direct download from the capture point.
The TCP video upload inter-domain traffic is considered the most expensive, for this reason it is supposed that reducing it, outstanding bandwidth saving and delay decreasing might be obtained. This is also one of the primary aims of CoreCast protocol [10].

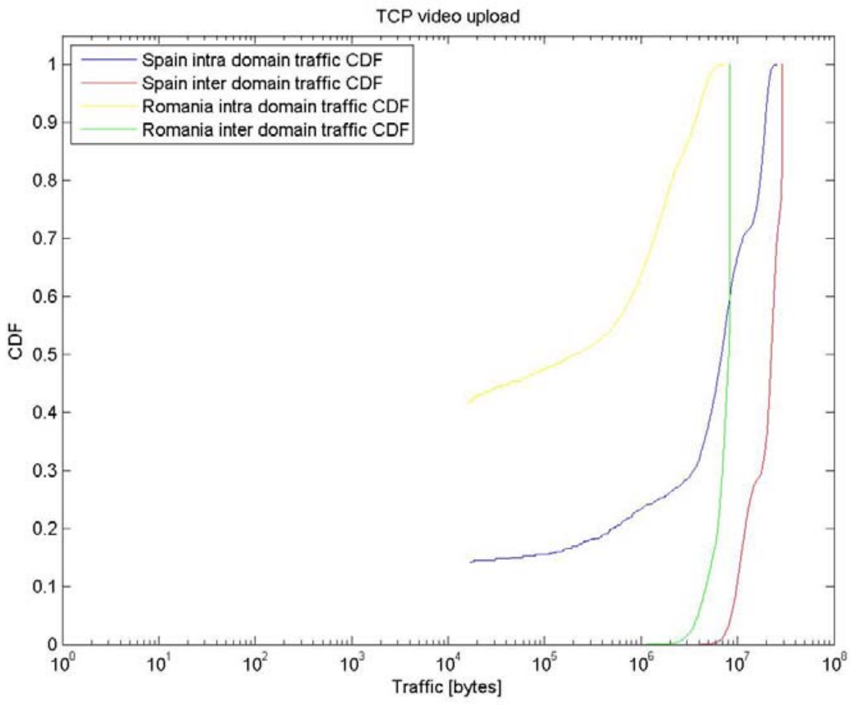


Fig. 6.27: CDF of the TCP video upload traffic, divided into intra-domain and inter-domain traffic for Spain and Romania networks

The graph in figure 6.27 represents the CDF of the TCP video upload intra-domain and inter-domain traffic for both spanish and romanian scenarios (four lines). On the x-axis, a logarithmic scale has been used, in order to cover a range of several orders of magnitude.
Observing the diagram, it can be noted that lines referred to the Romania traffic (the yellow one and the green one) are always at the left of the Spain traffic lines

(the blue one and the red one): it means that the TCP video upload traffic is bigger in Spain.

By the way, observing the lines referred to the inter-domain traffic (the red one and the green one), it can be affirmed that the romanian performance is better, because the network is able to maintain a relative low volume of inter-domain traffic, because the number of peers is much bigger than the spanish network.

6.6 P2P and CoreCast performance comparison

This last analysis differs a little bit from the others, because its aim is making a comparison between the performance of a normal p2p network and the performance of a p2p network using the CoreCast protocol. With details, the comparison regards only the download video traffic, both UDP and TCP, separated in intra-domain and inter-domain traffic.

The meaning of this evaluation is *testing if actually CoreCast acts as a intra-domain traffic reducer*. In order to answer this question, it is necessary calculating the intra-domain and the inter-domain traffic for a normal p2p network and for a CoreCast network. The data about a "classical" p2p network could be simply extracted from the output file of the second Perl script, implementing the split between intra-domain sessions and inter-domain sessions (paragraph 6.3).

Instead, the data about a CoreCast network have to be calculated using the formulas (3) and (4) shown in paragraph 4.4 (reported below), necessary to calculate the intra-domain and the inter-domain bandwidth.

$$BW_{\mathrm{int}er} = \frac{1}{T}\left(\left(LISP + CCP\right)\cdot j + \left(LISP + CCH\right)\cdot k\right) \quad (3)$$

$$BW_{\text{int}ra} = \frac{1}{T}\left(\left(IP + DATA\right)\cdot k\right) \tag{4}$$

In the end, just multiplying this two bandwidths with the total length of the traces (obtained through a subtraction between the timestamp of the first and the last packet of the session), it is possible to derive the relative amounts of intra and inter domain traffic.

Starting from the beginning, the first phase is the data extracting. In this contest, in addition to the usual information of traces always used during the previous analyses (IP source and destination addresses, the UDP and TCP source and destination port number and the payload), it is also necessary the timestamp (as in the precedent paragraph). So the TShark command structure is the following:

```
tshark -r pplive.udp.pcap  -Tfields -e ip.src -e ip.dst -e
udp.srcport -e udp.dstport -e ip.len -e frame.time_relative
> spain_T_udp
```

This time, the launched commands are four: two for UDP and TCP traffic of Spanish traces, two for UDP and TCP traffic of Romanian traces.
In order to execute the formulas (3) and (4), just two parameters are missing:

1. **T**, the average duration time of one session, expressed in seconds;
2. **DATA**, the average payload size.

For each of this parameters, a different Perl script has been implemented. The following block diagram (figure 6.36) refers to the estimation of T:

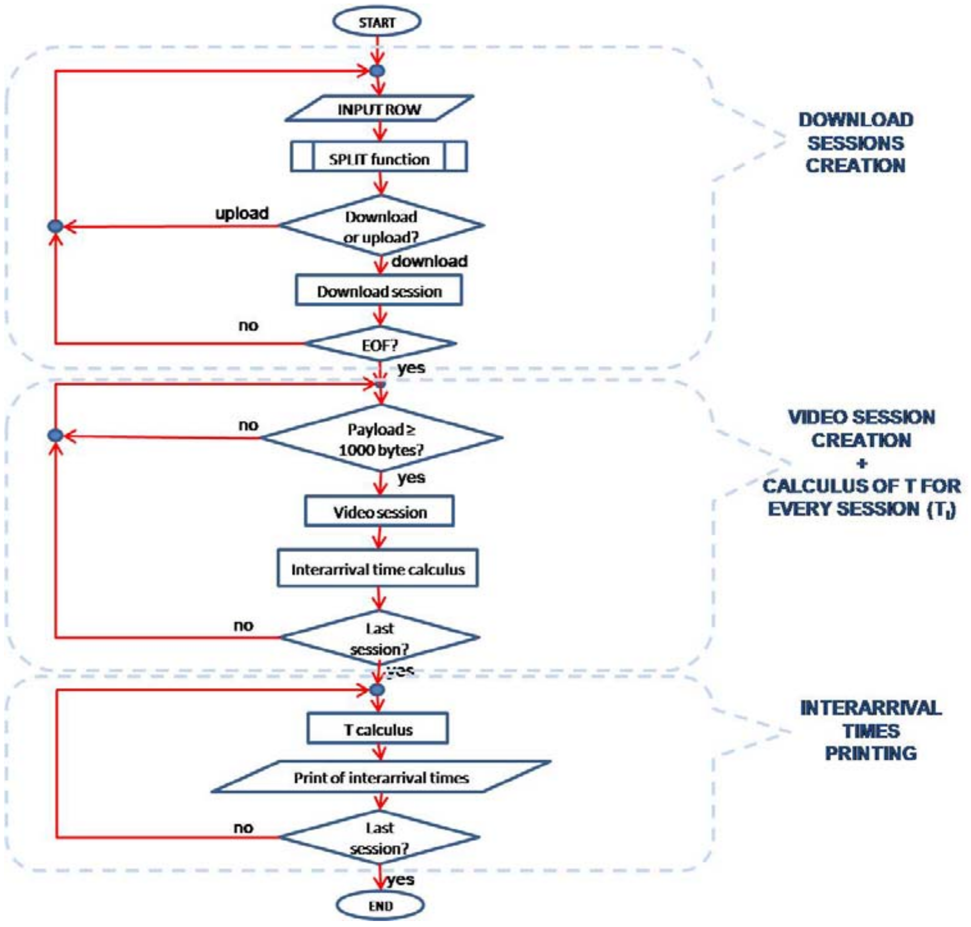


Fig. 6.36: Block diagram representing the logical process of the sixth Perl script implemented to calculate the T value

The first part of the diagram differs a little bit from the usual session creation function, because in this case only the video download sessions are required. As a matter of a fact, the upload sessions are already discarded during this first step.

Afterwards, the signaling sessions are also discarded during the second phase, in which the data structure, acted to contain the video sessions, is created. The duration time relative to the session (T_i) is calculated for every composed session. Then, all T_i are used to calculate the average duration time T.

In order to have a graphic representation of the calculated T (T =0.025 seconds), in the final part of the algorithm all the T_i are saved in a text file. Using the former, it has been plotted the graph shown in figure 6.37, representing the CDF of the T_i.

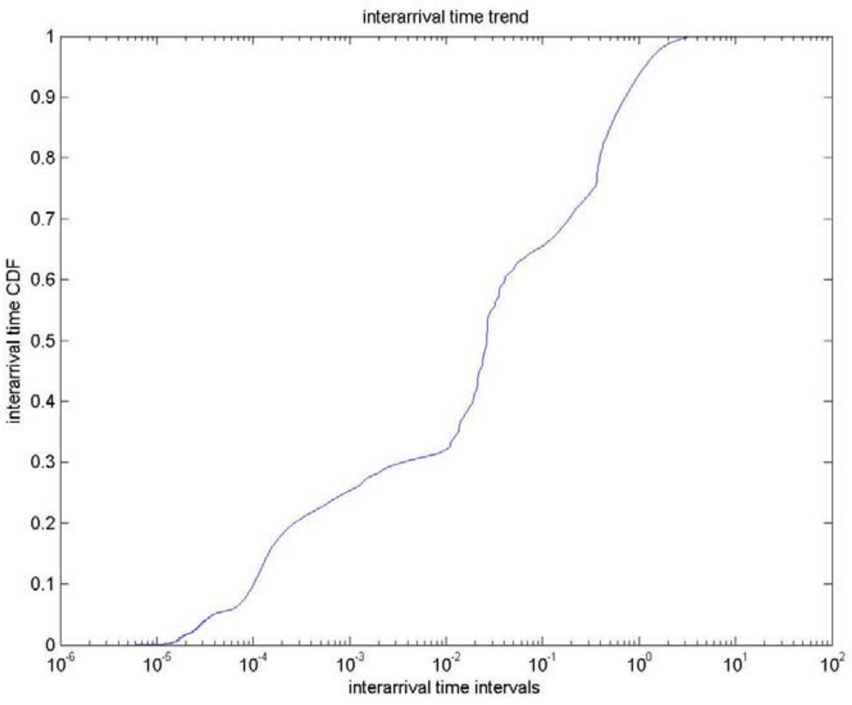


Fig. 6.37: representation of the CDF of the T_i values

The figure has a logarithmic scale on x-axis referring to all the interarrival time intervals T_i; on the y-axis there is the CDF of the T_i. The value on the x-axis correspondent to a CDF = 0.5 is just the value $T_i = 0.025$. It means that there is at most the 50% of the samples referring to the indicated value: for this reason the value 0.025 seconds represents the median of T_i, i.e. the T value. As expected, the graphic representation confirms the data found analytically through the Perl script.

Once T is calculated, the DATA parameter should be calculated, so, also for it, a proper Perl script has been implemented, according to the flow diagram in figure 6.38.

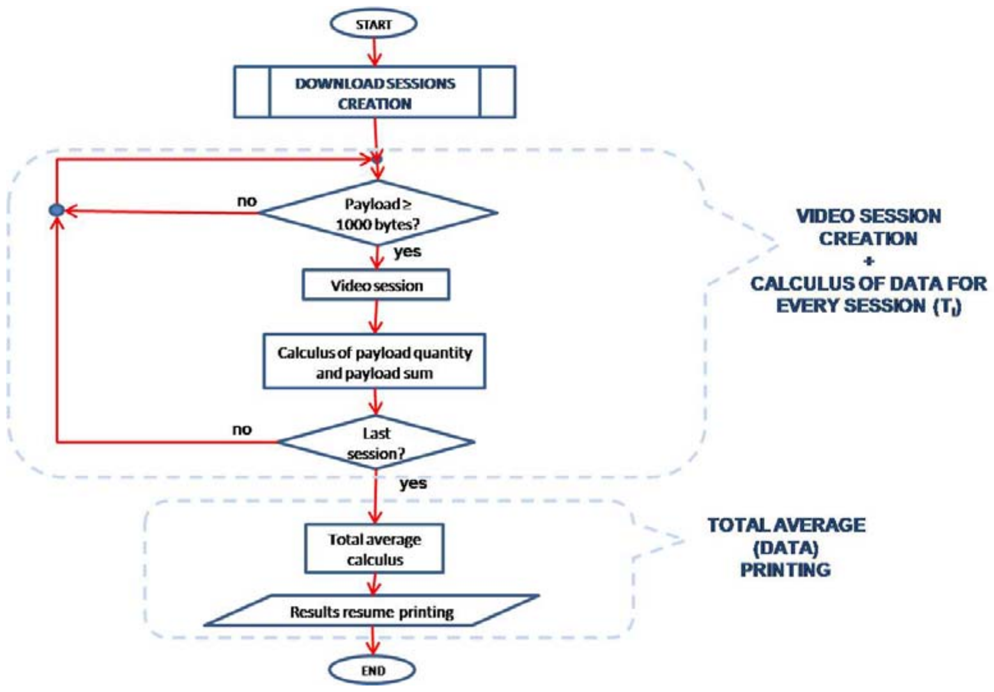

START
DOWNLOAD SESSIONS CREATION
no
Payload ≥ 1000 bytes?
yes
Video session
Calculus of payload quantity and payload sum
no
Last session?
yes
VIDEO SESSION CREATION + CALCULUS OF DATA FOR EVERY SESSION (Ti)
Total average calculus
Results resume printing
TOTAL AVERAGE (DATA) PRINTING
END

Fig. 6.38: Flow chart representing the logical process of the seventh Perl script implemented to calculate the DATA value

The first phase is identical to the previous diagram block (figure 6.36), for this reason just a function block has been reported.

The structure of the following phase, concerning the calculus of the *number of the packets* and the *payload sum* for a session, has the same structure of the second phase of the diagram in figure 6.36. Just the calculated parameters change.

After this step, using the data calculated for every session, the DATA value is obtained just making an average operation, because it represents the average payload size.

Finally, a resume of the results got is printed in a text file. It contains the sum of the payloads of all session, the total number of the packet and the value of DATA. All of this three parameters is calculated for both TCP and UDP traffic, then the sum of these is used into the formula (4) to obtain the intra-domain bandwidth.

At this stage of the game, all necessary parameters to use both formulas (3) and (4) are available. The results returned by the formulas are reassumed in the following tables:

SPANISH NETWORK: INTRA-DOMAIN AND INTER-DOMAIN TRAFFIC CALCULUS FOR A CORECAST NETWORK

INPUT DATA

k	3252
j	200
T	0,025
LISP	20
CCP	882,751898237040
CCH	24
IP	20
DATA TCP	860,751898237040
DATA UDP	0,000000000000
DATA total	860,751898237040

LENGTH OF TRACE

TCP		
last packet's timestamp	12.219,671825	-
first packet's timestamp	0	=
	12219,67183	
UDP		
last packet's timestamp	0	-
first packet's timestamp	0	=
	0	

$$BW_{inter} = \frac{1}{T}((LISP+CCP)\cdot j+(LISP+CCH)\cdot k)$$ = 12945535,19 bytes/seconds

$$BW_{intra} = \frac{1}{T}(IP + DATA)\cdot k$$

= 114568206,9	bytes/seconds	TCP
2601600	bytes/seconds	UDP
114568206,9	bytes/seconds	Total

CORECAST INTER DOMAIN TRAFFIC	= (BW_{inter} ·LENGTH OF TRACE)	=	**147,3260965 Gbytes**
CORECAST INTRA DOMAIN TRAFFIC	= (BW_{intra_TCP} ·LENGTH OF TRACE)	=	1303,8384636620 Gbytes
	= (BW_{intra_UDP} ·LENGTH OF TRACE)	=	29,6073949150 Gbytes
	= (BW_{intra_TOTAL} ·LENGTH OF TRACE)	=	**1303,8384636620 Gbytes**

Table 6.2: resume of CoreCast's results for the Spanish scenario

ROMANIAN NETWORK: INTRA-DOMAIN AND INTER-DOMAIN TRAFFIC CALCULUS FOR A CORECAST NETWORK

INPUT DATA

k	24253
j	1313
T	0,025
LISP	20
CCP	1222
CCH	24
IP	20
DATA TCP	1.002,003544872900
DATA UDP	878,967990996103
DATA total	888,899329491387

LENGTH OF TRACE

TCP		
last packet's timestamp	9086,85421	-
first packet's timestamp	0	=
	9086,85421	
UDP		
last packet's timestamp	9.807	-
first packet's timestamp	0	=
	9807,302584	

$$BW_{inter} = \frac{1}{T}((LISP+CCP)\cdot j+(LISP+CCH)\cdot k)$$ = 107915120 bytes/seconds

$$BW_{intra} = \frac{1}{T}(IP + DATA)\cdot k$$

= 991466079	bytes/seconds	TCP
872106827,4	bytes/seconds	UDP
881741417,5	bytes/seconds	Total

CORECAST INTER DOMAIN TRAFFIC	= (BW_{inter} ·LENGTH OF TRACE)	=	**985,6710539 Gbytes**
CORECAST INTRA DOMAIN TRAFFIC	= (BW_{intra_TCP} ·LENGTH OF TRACE)	=	9055,8154862889 Gbytes
	= (BW_{intra_UDP} ·LENGTH OF TRACE)	=	7965,6164554233 Gbytes
	= (BW_{intra_TOTAL} ·LENGTH OF TRACE)	=	**8053,6165111925 Gbytes**

Table 6.3: resume of CoreCast's results for the Romanian scenario

As pointed at the beginning of this paragraph, our purpose is comparing these results with the results extracted directly from the output file implementing the division between intra-domain and inter-domain traffic (paragraph 6.3). the data are reassumed here below:

SPANISH NETWORK: INTRA-DOMAIN AND INTER-DOMAIN TRAFFIC CALCULUS FOR A PEER-TO-PEER NETWORK		
PEER-TO-PEER INTER DOMAIN TRAFFIC	=	**167,023710206151 Gbytes**
PEER-TO-PEER INTRA DOMAIN TRAFFIC	=	**1926,24570011347 Gbytes**

ROMANIAN NETWORK: INTRA-DOMAIN AND INTER-DOMAIN TRAFFIC CALCULUS FOR A PEER-TO-PEER NETWORK		
PEER-TO-PEER INTER DOMAIN TRAFFIC	=	**2622,61816918849 Gbytes**
PEER-TO-PEER INTRA DOMAIN TRAFFIC	=	**17844,6774439569 Gbytes**

Table 6.4: resume of P2P results for Spain and Romania scenarios

The comparison between the performance of a classical p2p network and a CoreCast network is all in favor of the former, because CoreCast succeeds in its aim of reducing the inter domain traffic.

Spain network's inter-domain video downlaoad traffic		
P2P	167,02371 Gbytes	-
CoreCast	147,3261 Gbytes	=
SAVING	**19,697614 Gbytes**	

Romania network's inter-domain video downlaoad traffic		
P2P	2622,6182 Gbytes	-
CoreCast	985,67105 Gbytes	=
SAVING	**1636,9471 Gbytes**	

Table 6.5: saving obtained in inter-domain video download traffic through CoreCast using

The obtained saving is very considerable, and it is much bigger in Romania network. This is probably due to the ratio $\frac{k}{j}$, as table 6.6 shows:

	Spain scenario	Romania scenario
$\frac{k}{j}$	16,26	18,47144

Table 6.6: k/j ratios comparing in both considered scenarios

As explained in the fourth chapter, the parameter *j* represents the number of ASes in the network, and the parameter *k* is the number of the users. So, this ratio is very significant in CoreCast content, because the efficiency of the protocol is strictly connected with this parameter. It represents the distribution of the peers among the ASes of the network: if it is high, it means that the clients are located among few ASes. This is a good situation for CoreCast, because under this condition it is able to gain a big bandwidth saving (for this reason in Romania scenario CoreCast saves a huge amount of traffic).
Furthermore, positives results have been achieved in intra-domain traffic too, as the table 6.7 shows.

Spain network's intra-domain video downlaoad traffic			
P2P	1926,2457	Gbytes	-
CoreCast	1303,8384636620	Gbytes	=
SAVING	**622,4072365**	**Gbytes**	

Romania network's intra-domain video downlaoad traffic			
P2P	17844,67744	Gbytes	-
CoreCast	8053,6165111925	Gbytes	=
SAVING	**9791,060933**	**Gbytes**	

Table 6.7: saving obtained in intra-domain video download traffic through CoreCast using

Also in this case, more considerable advantages are obtained in Romania network. All of these results confirm that the CoreCast protocol can bring several advantages to the current p2p networks, improving the quality and reducing the costs in video streaming service.

CONCLUSIONS

During this work, the video delivery topic has been faced from several points of view, analyzing its characteristics, studying its applications and all the architecture providing this service.

The growing of the users interest in multimedia applications, such as Video on Demand, videoconferencing, or real-time video streaming of popular event, has pushed the researches in looking for more efficient methods, to allow the delivery to a constantly increasing number of applicants.

Generally, one source host (but also several sources) sends a video content to a multitude of recipients. There are several delivery methods, such as unicasting and broadcasting. As demonstrated during the first chapter, these two methods aren't very efficient, because they realize a bandwidth wasting, other than problems of delay. Even if multicasting represents the best way to allow a video live-streaming service, it is also very expensive, because it implies the maintaining of the knowledge of network topology in every router. Some scalability problems also affect multicasting, because as the numbers of multicast groups members increases, the data that routers have to store grow, too.

For this reason, the attention to the Peer-to-peer world has increased more and more, even that today a lot of multimedia applications have been implemented on p2p networks. Of course managing a video delivery of peer-to-peer network is cheaper than using a multicasting approach, but also in this contest, some problems about QoS parameters and bandwidth consumption still appear. Among all these problems, this work has pointed the attention on the intra-ISP and inter-ISP peer link availability, just because it can lead the formation of some bottlenecks in the proximity of ISP boundaries.

In order to improve the video streaming service on p2p networks, a new network-layer live streaming protocol has been thought. CoreCast is built on top of the LISP, and its most strong point is the reduction of inter-domain traffic (it can be defined as an ISP-friendly protocol). As a matter of fact, this work has demonstrated that a remarkable saving of inter domain bandwidth (and of intra-domain bandwidth too) is obtained using CoreCast in transmitting a very popular video content through a large scale p2p network, like PPlive. Note that CoreCast does not imply a lot of modifications of the existing p2p framework, furthermore it works on LISP, that is considered the most promising implementation of the Loc/ID split. In future, CoreCast could be implied on the top of other structures implementing the Loc/ID split, because its adoption is not bounded to the LISP structure.

During these work a lot of evaluations have been made on a big amount of data captured during the Michael Jackson funerals. The results can give several analytical information about the traffic characteristic, such as the distinction between download and upload data, UDP and TCP traffic, data or control and finally inter-domain and intra-domain. All of these information are particularly interesting and useful, because their study allows understanding the p2p video delivery mechanisms, the peers behavior and the network characteristics.

Furthermore, this work has analytically demonstrated that CoreCast can be considered a reliable solution to the problem of intra-domain traffic. It gives a more satisfactory performance, compared with a “traditional” p2p streaming-live service. Since the analyzed traces can give information about network only from the point of view of the source node, it is impossible to achieve a general idea of the network viewed from all peers. Despite this lack, the heuristic algorithm used in order to simulate the distribution of the traffic among all peers, has turned out to be very useful to separate data traffic to signaling.

Finding a way to obtain the data about the transmission from every peer of the network, can be an interesting start point for future analyses of the peer-to-peer networks.

Other network features can be understood, for instance, studying the trend of the population (i.e. the number of different peers that populate the network in every moment) considering several channels (in this work just one channel has been considered). Besides, a load-balancing investigation could be interesting in order to understand the distribution of the workloads among the peers, than to try to implement some load-balance algorithms aimed to homogenize the traffic distribution.

Used acronyms’ list

AAA = Authentication, Authorization, and Accounting

AAC = Advanced Audio Coding

ADSL = Asymmetric Digital Subscriber Line

ARP = Address Resolution Protocol

AS = Autonomous System

AS = Autonomous System

ASM = Any Source Multicast

BSR = Bootstrap Router

BST = British Summer Time

CDF = Cumulative Distribution Function

ChanDstList = Channel Destination List

CPU = Central Processing Unit

DCCP = Datagram Congestion Control Protocol

DFZ = Default Free Zone

DS = Differentiated Services

EID = Endpoint Identifier

EOF = End of File

ETR = the Egress Tunnel Router

FF = Fixed-Filter

FIFO = First In First Out

FTP = File Transfer Protocol

GIMP = GNU Image Manipulation Program

GNU = GNU is Not Unix

GPL = General Public License

GTK = GIMP ToolKit

HTTP	= Hyper Text Transfer Protocol
ICMP	= Internet Control Message Protocol
IDE	= Integrated Development Environment
IEEE	= Institute of Electrical and Electronic Engineers
IETF	= Internet Engineering Task Force
IGMP	= Internet Group Management Protocol
IP	= Internet Protocol
IPT	= Inter Packet Time
IPTV	= Internet Protocol TeleVision
IPv4	= Internet Protocol Version 4
IPv6	= Internet Protocol Version 6
ISP	= Internet Service Provider
ITR	= Ingress Tunnel Router
LAN	= Local Area Network
LISP	= Locator/ID Separation Protocol
Loc/ID	= Locator/Identifier
MAC	= Media Access Control
MATLAB	= Matrix Laboratory
MPEG	= Moving Picture Experts Group
NVoD	= Near Video on Demand
OSPF	= Open Shortest Path First
P2P	= Peer-to-Peer
Perl	= Practical Extraction and Report Language
PS	= Payload Size
QoS	= Quality of Service
RAM	= Random Access Memory
RG	= Routing Goop
RIP	= Routing Information Protocol
RLOC	= Routing Locator
RMVB	= Real Media Variable Bitrate
RP	= Rendezvous Point

RPB = Reverse-Path Broadcasting

RPF = Reverse-Path Forwarding

RPM = Reverse-Path Multicasting

RR = Receiver Report

RS - ECC = Reed Solomon Error Correcting Code

RSVP = Resource Reservation Protocol

RTCP = Real Time Control Protocol

RTP = Real Time Protocol

SE = Shared-Explicit

SLA = Service Layer Agreement

SR = Sender Report

SSM = Single Source Multicast

TCP = Transmission Control Protocol

TRPB = Truncated Reverse-Path Broadcasting

TTL = Time to Live

TV = TeleVision

TVoD = True Video on Demand

UDP = User Datagram Protocol

UVoD = Unified Video on Demand

VoD = Video on Demand

WAN = Wide Area Network

WAR = Write After Read

WE = Wildcard-Filter

WMV = Windows Media Video

APPENDIX A

Implemented Perl scripts

In this section all Perl scripts produced during this work are reported according to the order of their using in the sixth chapter. Some comments are writing in order to help the comprehension.

1. UDP-TCP, Upload-Download, Control-Data traffic division

```
#!/usr/bin/perl
#reference source IP address
$ip_rif = '147.83.130.144';    #Spain
#$ip_rif = '192.168.1.101';    #Romania

my %ip; #it keeps all IP addresses of the network
#hash structures for printing
my %TCP_up_video; #ip and payload of TCP upload video traffic
my %TCP_up_control;#ip and payload of TCP upload control traffic
my %TCP_down_video; #ip and payload of TCP download video traffic
my %TCP_down_control; #ip and payload of TCP download control
traffic
my %UDP_up_video; #ip and payload of UDP upload video traffic
my %UDP_up_control; #ip and payload of UDP upload control traffic
my %UDP_down_video; #ip and payload of UDP download video traffic
my %UDP_down_control; #ip and payload of UDP download control
traffic

# ***************** TCP *****************
#TCP sessions hash definition
my %up_d_ip; #contains all IP destination address in upload
my %down_d_ip; #contains all IP source address in download
#TCP packets file opening
open(IN, "< C:/Program Files/Perl Express/Input/TCP_only.txt") ||
die "It’s impossible to open the file\n\n";
while(<IN>) {
chomp;
(my $s_ip, my $d_ip, my $s_port, my $d_port, my $byte) = split;
#control instruction
if(  (!($s_ip)) || (!($d_ip)) || (!($s_port)) || (!($d_port)) ||
(!($byte)) || ( ($s_ip) eq '0.0.0.0') || ( ($d_ip) eq '0.0.0.0')
|| ( ($s_ip) eq 'NA') || ( ($d_ip) eq 'NA') ) { next;}
# hash loading
if($s_ip eq $ip_rif)
      { #upload session
        $up_d_ip{$d_ip}{$s_port}{$d_port}= []unless exists
        $up_d_ip{$d_ip}{$s_port}{$d_port};
        push @{$up_d_ip{$d_ip}{$s_port}{$d_port}}, $byte;
        $ip{$d_ip}=1;
      }
else
      {  #download session
      $down_d_ip{$s_ip}{$s_port}{$d_port}=[]unless exists
      $down_d_ip{$s_ip}{$s_port}{$d_port};
      push @{$down_d_ip{$s_ip}{$s_port}{$d_port}}, $byte;
      $ip{$s_ip}=1;
      }
}
#file closing
close(IN);

#split of video and control traffic
```

```
foreach my $a( keys %up_d_ip)
{       #cicle of source ports
      foreach my $b( keys %{$up_d_ip{$a} } )
        {       #cicle on destination ports
            foreach my $c( keys %{$up_d_ip{$a}{$b}  } )
                {
                  my $count=0;
                  my $somma=0;
                  my $i=0;
                  for ($i=0; $i<= $#{$up_d_ip{$a}{$b}{$c}}; $i++)
                        {
                        If (${$up_d_ip{$a}{$b}{$c}}[$i]>1000)
                              {$count++;}
                        $somma +=$up_d_ip{$a}{$b}{$c}[$i];
                        }
                  if ($count>10) {$TCP_up_video{$a}+=$somma;}
                  else {$TCP_up_control{$a}+=$somma;}}}}

foreach my $a( keys %down_d_ip)
{       #cicle on source ports
      foreach my $b( keys %{$down_d_ip{$a} } )
        {       #cicle on destination ports
            foreach my $c( keys %{$down_d_ip{$a}{$b}  } )
                {
                  my $count=0;
                  my $somma=0;
                  my $i=0;
                  for ($i=0; $i<= $#{$down_d_ip{$a}{$b}{$c}};$i++)
                        {
                        If (${$down_d_ip{$a}{$b}{$c}}[$i]>1000)
                              {$count++;}
                        $somma +=$down_d_ip{$a}{$b}{$c}[$i];
                        }
                  if ($count>10) {$TCP_down_video{$a}+=$somma;}
                        else {$TCP_down_control{$a}+=$somma;}}}}

# **************** UDP ****************
```

```
#UDP sessions hash definition
my %up_d_ip; #contains all IP destination address in upload
my %down_d_ip; #contains all IP source address in download
#UDP packets file opening
open(IN, "< C:/Program Files/Perl Express/Input/UDP_only.txt") ||
die "It’s impossible to open the file\n\n";
while(<IN>) {
chomp;
(my $s_ip, my $d_ip, my $s_port, my $d_port, my $byte) = split;
#control instruction
if(  (!($s_ip)) || (!($d_ip)) || (!($s_port)) || (!($d_port)) ||
(!($byte)) || ( ($s_ip) eq '0.0.0.0') || ( ($d_ip) eq '0.0.0.0')
|| ( ($s_ip) eq 'NA') || ( ($d_ip) eq 'NA') ) { next;}
# hash loading
if($s_ip eq $ip_rif)
      { #upload session
        $up_d_ip{$d_ip}{$s_port}{$d_port}= []unless exists
        $up_d_ip{$d_ip}{$s_port}{$d_port};
        push @{$up_d_ip{$d_ip}{$s_port}{$d_port}}, $byte;
        $ip{$d_ip}=1;
      }
else
      {  #download session
      $down_d_ip{$s_ip}{$s_port}{$d_port}=[]unless exists
      $down_d_ip{$s_ip}{$s_port}{$d_port};
      push @{$down_d_ip{$s_ip}{$s_port}{$d_port}}, $byte;
      $ip{$s_ip}=1;
      }
}
#file closing
close(IN);

#split of video and control traffic
foreach my $a( keys %up_d_ip)
{       #cicle of source ports
      foreach my $b( keys %{$up_d_ip{$a} } )
        {       #cicle on destination ports
```

```
            foreach my $c( keys %{$up_d_ip{$a}{$b}  } )
                {
                  my $count=0;
                  my $somma=0;
                  my $i=0;
                  for ($i=0; $i<= $#{$up_d_ip{$a}{$b}{$c}}; $i++)
                        {
                        If (${$up_d_ip{$a}{$b}{$c}}[$i]>1000)
                              {$count++;}
                        $somma +=$up_d_ip{$a}{$b}{$c}[$i];
                        }
                  if ($count>10) {$UDP_up_video{$a}+=$somma;}
                  else {$UDP_up_control{$a}+=$somma;}}}}

foreach my $a( keys %down_d_ip)
{       #cicle on source ports
      foreach my $b( keys %{$down_d_ip{$a} } )
        {       #cicle on destination ports
            foreach my $c( keys %{$down_d_ip{$a}{$b}  } )
                {
                  my $count=0;
                  my $somma=0;
                  my $i=0;
                  for ($i=0; $i<= $#{$down_d_ip{$a}{$b}{$c}};$i++)
                        {
                        If (${$down_d_ip{$a}{$b}{$c}}[$i]>1000)
                              {$count++;}
                        $somma +=$down_d_ip{$a}{$b}{$c}[$i];
                        }
                  if ($count>10) {$UDP_down_video{$a}+=$somma;}
                        else {$UDP_down_control{$a}+=$somma;}}}}

#********************** PRINTING ********************************

open(IN, "> C:/Program Files/Perl Express/Input/total_output.txt")
|| die "It’s impossible to open the file\n\n";
```

```
#human labels printing
print "       IP \t\t|\t\t\t VIDEO
\t\t\t\t\t|\t\t\tCONTROL\t\t\t\t\t|\t\t\t TOTAL \t\t\t\t\t|\n";
print
"\t\t\t|\t\tupload\t\t|\t\tdownload\t|\t\tupload\t\t|\t\tdownload\
t|\t\tupload\t\t|\t\tdownload\t|\n";
print "\t\t\t|  TCP  \t|  UDP  \t|  TCP  \t|  UDP  \t|  TCP  \t|
UDP  \t|  TCP  \t|  UDP  \t|  TCP  \t|  UDP  \t|  TCP  \t|  UDP
\t| \n\n";
#pc labels printing
printf IN "       IP \t\t|\t VIDEO \t\t|\tCONTROL\t\t|\t TOTAL
\t\t|\n";
printf IN
"\t\t\t|upload\t|download\t|upload\t|download\t|upload\t|download\
t|\n";
printf IN
"\t\t\t|TCP\t|UDP\t|TCP\t|UDP\t|TCP\t|UDP\t|TCP\t|UDP\t|TCP\t|UDP\
t|TCP\t|UDP\t|\n\n";

#values printing
foreach $k(keys %ip)
{
if( !($TCP_up_video{$k})) {$TCP_up_video{$k}= 0;}
if( !($TCP_up_control{$k})) {$TCP_up_control{$k}= 0;}
if( !($TCP_down_video{$k})) {$TCP_down_video{$k}= 0;}
if( !($TCP_down_control{$k})) {$TCP_down_control{$k}= 0;}
if( !($UDP_up_video{$k})) {$UDP_up_video{$k}= 0;}
if( !($UDP_up_control{$k})) {$UDP_up_control{$k}= 0;}
if( !($UDP_down_video{$k})) {$UDP_down_video{$k}= 0;}
if( !($UDP_down_control{$k})) {$UDP_down_control{$k}= 0;}
$s1= $TCP_up_video{$k} + $TCP_up_control{$k};
$s2= $UDP_up_video{$k} + $UDP_up_control{$k};
$s3= $TCP_down_video{$k} + $TCP_down_control{$k};
$s4= $UDP_down_video{$k} + $UDP_down_control{$k};

#***************** HUMAN PRINTING  **********************
print
"$k\t\t$TCP_up_video{$k}\t\t$UDP_up_video{$k}\t\t$TCP_down_video{$
k}\t\t$UDP_down_video{$k}\t\t$TCP_up_control{$k}\t\t$UDP_up_contro
l{$k}\t\t$TCP_down_control{$k}\t\t$UDP_down_control{$k}\t\t$s1\t\t
$s2\t\t$s3\t\t$s4\n";

#******************** PC PRINTING  **********************



printf IN
"%s\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",$k,$TCP_up_
video{$k},$UDP_up_video{$k},$TCP_down_video{$k},$UDP_down_video{$k
},$TCP_up_control{$k},$UDP_up_control{$k},$TCP_down_control{$k},$U
DP_down_control{$k},$s1,$s2,$s3,$s4;
}
close(IN);
```

2. Intra-domain and inter-domain traffic division

```
#!/usr/bin/perl
#reference source IP address
$ip_rif = '147.83.130.144';    #Spain
#$ip_rif = '192.168.1.101';    #Romania

my %ip; #it keeps all IP addresses of the network
#hash structures for printing
my %TCP_up_video; #ip and payload of TCP upload video traffic
my %TCP_up_control;#ip and payload of TCP upload control traffic
my %TCP_down_video; #ip and payload of TCP download video traffic
my %TCP_down_control; #ip and payload of TCP download control
traffic
my %UDP_up_video; #ip and payload of UDP upload video traffic
my %UDP_up_control; #ip and payload of UDP upload control traffic
my %UDP_down_video; #ip and payload of UDP download video traffic
my %UDP_down_control; #ip and payload of UDP download control
traffic

# ***************** TCP *****************
#TCP sessions hash definition
my %up_d_ip; #contains all IP destination address in upload
my %down_d_ip; #contains all IP source address in download
#TCP packets file opening
open(IN, "< C:/Program Files/Perl Express/Input/TCP_only.txt") ||
die "It’s impossible to open the file\n\n";
while(<IN>) {
chomp;
(my $s_ip, my $d_ip, my $s_port, my $d_port, my $byte) = split;
#control instruction
if(  (!($s_ip)) || (!($d_ip)) || (!($s_port)) || (!($d_port)) ||
(!($byte)) || ( ($s_ip) eq '0.0.0.0') || ( ($d_ip) eq '0.0.0.0')
|| ( ($s_ip) eq 'NA') || ( ($d_ip) eq 'NA') ) { next;}
# hash loading
if($s_ip eq $ip_rif)
      { #upload session
        $up_d_ip{$d_ip}{$s_port}{$d_port}= []unless exists
        $up_d_ip{$d_ip}{$s_port}{$d_port};
        push @{$up_d_ip{$d_ip}{$s_port}{$d_port}}, $byte;
        $ip{$d_ip}=1;
      }
else
      {  #download session
      $down_d_ip{$s_ip}{$s_port}{$d_port}=[]unless exists
      $down_d_ip{$s_ip}{$s_port}{$d_port};
      push @{$down_d_ip{$s_ip}{$s_port}{$d_port}}, $byte;
      $ip{$s_ip}=1;
      }
}
#file closing
close(IN);
```

```
#split of video and control traffic
foreach my $a( keys %up_d_ip)
{       #cicle of source ports
      foreach my $b( keys %{$up_d_ip{$a} } )
        {       #cicle on destination ports
            foreach my $c( keys %{$up_d_ip{$a}{$b}  } )
                {
                  my $count=0;
                  my $somma=0;
                  my $i=0;
                  for ($i=0; $i<= $#{$up_d_ip{$a}{$b}{$c}}; $i++)
                        {
                        If (${$up_d_ip{$a}{$b}{$c}}[$i]>1000)
                              {$count++;}
                        $somma +=$up_d_ip{$a}{$b}{$c}[$i];
                        }
                  if ($count>10) {$TCP_up_video{$a}+=$somma;}
                  else {$TCP_up_control{$a}+=$somma;}}}}

foreach my $a( keys %down_d_ip)
{       #cicle on source ports
      foreach my $b( keys %{$down_d_ip{$a} } )
        {       #cicle on destination ports
            foreach my $c( keys %{$down_d_ip{$a}{$b}  } )
                {
                  my $count=0;
                  my $somma=0;
                  my $i=0;
                  for ($i=0; $i<= $#{$down_d_ip{$a}{$b}{$c}};$i++)
                        {
                        If (${$down_d_ip{$a}{$b}{$c}}[$i]>1000)
                              {$count++;}
                        $somma +=$down_d_ip{$a}{$b}{$c}[$i];
                        }
                  if ($count>10) {$TCP_down_video{$a}+=$somma;}
                        else {$TCP_down_control{$a}+=$somma;}}}}
```

```
# ***************** UDP *****************
#UDP sessions hash definition
my %up_d_ip; #contains all IP destination address in upload
my %down_d_ip; #contains all IP source address in download
#UDP packets file opening
open(IN, "< C:/Program Files/Perl Express/Input/UDP_only.txt") ||
die "It’s impossible to open the file\n\n";
while(<IN>) {
chomp;
(my $s_ip, my $d_ip, my $s_port, my $d_port, my $byte) = split;
#control instruction
if(  (!($s_ip)) || (!($d_ip)) || (!($s_port)) || (!($d_port)) ||
(!($byte)) || ( ($s_ip) eq '0.0.0.0') || ( ($d_ip) eq '0.0.0.0')
|| ( ($s_ip) eq 'NA') || ( ($d_ip) eq 'NA') ) { next;}
# hash loading
if($s_ip eq $ip_rif)
      { #upload session
        $up_d_ip{$d_ip}{$s_port}{$d_port}= []unless exists
        $up_d_ip{$d_ip}{$s_port}{$d_port};
        push @{$up_d_ip{$d_ip}{$s_port}{$d_port}}, $byte;
        $ip{$d_ip}=1;
      }
else
      {  #download session
      $down_d_ip{$s_ip}{$s_port}{$d_port}=[]unless exists
      $down_d_ip{$s_ip}{$s_port}{$d_port};
      push @{$down_d_ip{$s_ip}{$s_port}{$d_port}}, $byte;
      $ip{$s_ip}=1;
      }
}
#file closing
close(IN);

#split of video and control traffic
foreach my $a( keys %up_d_ip)
{       #cicle of source ports
      foreach my $b( keys %{$up_d_ip{$a} } )
```

```
        {       #cicle on destination ports
            foreach my $c( keys %{$up_d_ip{$a}{$b}  } )
                {
                  my $count=0;
                  my $somma=0;
                  my $i=0;
                  for ($i=0; $i<= $#{$up_d_ip{$a}{$b}{$c}}; $i++)
                        {
                        If (${$up_d_ip{$a}{$b}{$c}}[$i]>1000)
                              {$count++;}
                        $somma +=$up_d_ip{$a}{$b}{$c}[$i];
                        }
                  if ($count>10) {$UDP_up_video{$a}+=$somma;}
                  else {$UDP_up_control{$a}+=$somma;}}}}

foreach my $a( keys %down_d_ip)
{       #cicle on source ports
      foreach my $b( keys %{$down_d_ip{$a} } )
        {       #cicle on destination ports
            foreach my $c( keys %{$down_d_ip{$a}{$b}  } )
                {
                  my $count=0;
                  my $somma=0;
                  my $i=0;
                  for ($i=0; $i<= $#{$down_d_ip{$a}{$b}{$c}};$i++)
                        {
                        If (${$down_d_ip{$a}{$b}{$c}}[$i]>1000)
                              {$count++;}
                        $somma +=$down_d_ip{$a}{$b}{$c}[$i];
                        }
                  if ($count>10) {$UDP_down_video{$a}+=$somma;}
                        else {$UDP_down_control{$a}+=$somma;}}}}

#*************** IP AND AS NUMBER LOADING **********************
my %ip_as; #definition of the hash collecting all IP address and
AS number
```

```
open(IN, "< C:/Program Files/Perl Express/Input/ip2asn.dat") ||
die "It’s impossible to open the file\n\n";
while(<IN>)
{
      chomp;
      (my $as, my $barra, my $ip) = split;
      if(  (!($ip)) || (!($as)) || (($as) eq 'NA') )
            { next;}
      $ip_as{$ip}=$as;
}
close (IN);

#************************** PRINTING *************************
open(IN, "> C:/Program Files/Perl
Express/Input/spain_pc_output.txt") || die " It’s impossible to
open the file\n\n";

#human labels printing
print "       IP \t\t|  AS\t|\t\t\t VIDEO
\t\t\t\t\t|\t\t\tCONTROL\t\t\t\t\t|\t\t\t TOTAL \t\t\t\t\t|\n";
print
"\t\t\t|\t|\t\tupload\t\t|\t\tdownload\t|\t\tupload\t\t|\t\tdownlo
ad\t|\t\tupload\t\t|\t\tdownload\t|\n";
print "\t\t\t|\t|  TCP  \t|  UDP  \t|  TCP  \t|  UDP  \t|  TCP
\t|  UDP  \t|  TCP  \t|  UDP  \t|  TCP  \t|  UDP  \t|  TCP  \t|
UDP  \t| \n\n";

#pc labels printing
printf IN "       IP \t\t| AS\t|\t VIDEO \t\t|\tCONTROL\t\t|\t
TOTAL \t\t|\n";
printf IN
"\t\t\t|\t|upload\t|download\t|upload\t|download\t|upload\t|downlo
ad\t|\n";
printf IN
"\t\t\t|\t|TCP\t|UDP\t|TCP\t|UDP\t|TCP\t|UDP\t|TCP\t|UDP\t|TCP\t|U
DP\t|TCP\t|UDP\t|\n\n";
#value printing
```

```
foreach $k(keys %ip_as)
{
if( !($TCP_up_video{$k})) {$TCP_up_video{$k}= 0;}
if( !($TCP_up_control{$k})) {$TCP_up_control{$k}= 0;}
if( !($TCP_down_video{$k})) {$TCP_down_video{$k}= 0;}
if( !($TCP_down_control{$k})) {$TCP_down_control{$k}= 0;}
if( !($UDP_up_video{$k})) {$UDP_up_video{$k}= 0;}
if( !($UDP_up_control{$k})) {$UDP_up_control{$k}= 0;}
if( !($UDP_down_video{$k})) {$UDP_down_video{$k}= 0;}
if( !($UDP_down_control{$k})) {$UDP_down_control{$k}= 0;}
$s1= $TCP_up_video{$k} + $TCP_up_control{$k};
$s2= $UDP_up_video{$k} + $UDP_up_control{$k};
$s3= $TCP_down_video{$k} + $TCP_down_control{$k};
$s4= $UDP_down_video{$k} + $UDP_down_control{$k};

#**********************  HUMAN PRINTING  **********************
print
"$k\t\t$ip_as{$k}\t$TCP_up_video{$k}\t\t$UDP_up_video{$k}\t\t$TCP_
down_video{$k}\t\t$UDP_down_video{$k}\t\t$TCP_up_control{$k}\t\t$U
DP_up_control{$k}\t\t$TCP_down_control{$k}\t\t$UDP_down_control{$k
}\t\t$s1\t\t$s2\t\t$s3\t\t$s4\n";


#**********************  PC PRINTING  **********************
printf IN
"%s\t%s\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",$k,$ip_
as{$k},$TCP_up_video{$k},$UDP_up_video{$k},$TCP_down_video{$k},$UD
P_down_video{$k},$TCP_up_control{$k},$UDP_up_control{$k},$TCP_down
_control{$k},$UDP_down_control{$k},$s1,$s2,$s3,$s4;
}
close(IN);
```

3. Ports' utilization

```
#!/usr/bin/perl
#reference source IP elimination

#insert the source IP according to the traces that you are going
to analize
#$ip_rif = '147.83.130.144';  #Spain
$ip_rif = '192.168.1.101';    #Romania

#TCP files opening
open(IN, "< C:/Program Files/Perl Express/Input/Romania_tcp.txt")
|| die "It’s impossible to open the file\n\n";
open(OUT, "> C:/Program Files/Perl
Express/Input/Romania_ports_tcp.txt") || die " It’s impossible to
open the file\n\n";

while(<IN>) {
chomp;
(my $s_ip, my $d_ip, my $s_port, my $d_port, my $byte) = split;
#control instruction
if(  (!($s_ip)) || (!($d_ip)) || (!($s_port)) || (!($d_port)) ||
(!($byte)) || ( ($s_ip) eq '0.0.0.0') || ( ($d_ip) eq '0.0.0.0')
|| ( ($s_ip) eq 'NA') || ( ($d_ip) eq 'NA') ) { next;}
#output writing
if($s_ip eq $ip_rif)
      { #upload
        print OUT "$d_ip\t$s_port\t$d_port\n"; }
else
      {  #download
        print OUT "$s_ip\t$d_port\t$s_port\n"; }
}
#TCP files closing
close(IN);
close (OUT);
```

```
open(IN, "< C:/Program Files/Perl Express/Input/Romania_udp.txt")
|| die "Impossibile aprire il file\n\n";
open(OUT, "> C:/Program Files/Perl
Express/Input/Romania_ports_udp.txt") || die "It’s impossible to
open the file\n\n";

while(<IN>) {
chomp;
(my $s_ip, my $d_ip, my $s_port, my $d_port, my $byte) = split;
if(  (!($s_ip)) || (!($d_ip)) || (!($s_port)) || (!($d_port)) ||
(!($byte)) || ( ($s_ip) eq '0.0.0.0') || ( ($d_ip) eq '0.0.0.0')
|| ( ($s_ip) eq 'NA') || ( ($d_ip) eq 'NA') ) { next;}
#hashes loading
if($s_ip eq $ip_rif)
      { #siamo in upload
        print OUT "$d_ip\t$s_port\t$d_port\n"; }
else
      {  #siamo in download
        print OUT "$s_ip\t$d_port\t$s_port\n"; }
}
#UDP files closing
close(IN);
close (OUT);

#ROMANIA
#*************************  TCP  *********************************
my %tcp_s_h; #it keeps all IP destination address in upload
my %tcp_d_h; #it keeps all IP source address in download

#TCP file opening
open(IN, "< C:/Program Files/Perl
Express/Input/romania_ports_tcp.txt") || die "It’s impossible to
open the file\n\n";
while(<IN>) {
chomp;
(my $ip, my $s_port, my $d_port) = split;
if(!($ip)) { next;}
#hash loading
$tcp_s_h{$s_port}++;
$tcp_d_h{$d_port}++;
}
close(IN);

#************************  UDP  ********************************
my %udp_s_h; #it keeps all IP destination address in upload
my %udp_d_h; # it keeps all IP source address in download

#UDP file opening
open(IN, "< C:/Program Files/Perl
Express/Input/romania_ports_udp.txt") || die "It’s impossible to
open the file\n\n";
while(<IN>) {
chomp;
(my $ip, my $s_port, my $d_port) = split;
if(!($ip)) { next;}
#hashes loading
$udp_s_h{$s_port}++;
$udp_d_h{$d_port}++;
}
close(IN);

#************************* PRINTING ***************************
open(OUT, "> C:/Program Files/Perl
Express/Input/romania_tcp_sourceport_count.txt") || die " It’s
impossible to open the file\n\n";
foreach $k(keys %tcp_s_h) {print OUT "$k\t$tcp_s_h{$k}\n";}
close(OUT);

open(OUT, "> C:/Program Files/Perl
Express/Input/romania_tcp_destport_count.txt") || die " It’s
impossible to open the file\n\n";
foreach $k(keys %tcp_d_h) {print OUT "$k\t$tcp_d_h{$k}\n";}
close(OUT);
```

```
open(OUT, "> C:/Program Files/Perl
Express/Input/romania_udp_sourceport_count.txt") || die " It’s
impossible to open the file\n\n";
foreach $k(keys %udp_s_h) {print OUT "$k\t$udp_s_h{$k}\n";}
close(OUT);

open(OUT, "> C:/Program Files/Perl
Express/Input/romania_udp_destport_count.txt") || die " It’s
impossible to open the file\n\n";
foreach $k(keys %udp_d_h) {print OUT "$k\t$udp_d_h{$k}\n";}
close(OUT);

#hashes emptying
%tcp_s_h=();
%tcp_d_h=();
%udp_s_h=();
%udp_d_h=();

#SPAIN
#*************************  TCP  ********************************
my %tcp_s_h; #it keeps all IP destination address in upload
my %tcp_d_h; #it keeps all IP source address in download

#TCP file opening
open(IN, "< C:/Program Files/Perl
Express/Input/spain_ports_tcp.txt") || die "It’s impossible to
open the file\n\n";
while(<IN>) {
chomp;
(my $ip, my $s_port, my $d_port) = split;
if(!($ip)) { next;}
#hash loading
$tcp_s_h{$s_port}++;
$tcp_d_h{$d_port}++;
}
close(IN);
```

```
#************************  UDP  ********************************
my %udp_s_h; #it keeps all IP destination address in upload
my %udp_d_h; # it keeps all IP source address in download

#UDP file opening
open(IN, "< C:/Program Files/Perl
Express/Input/spain_ports_udp.txt") || die "It’s impossible to
open the file\n\n";
while(<IN>) {
chomp;
(my $ip, my $s_port, my $d_port) = split;
if(!($ip)) { next;}
#hashes loading
$udp_s_h{$s_port}++;
$udp_d_h{$d_port}++;
}
close(IN);

#************************* PRINTING ****************************
open(OUT, "> C:/Program Files/Perl
Express/Input/spain_tcp_sourceport_count.txt") || die " It’s
impossible to open the file\n\n";
foreach $k(keys %tcp_s_h) {print OUT "$k\t$tcp_s_h{$k}\n";}
close(OUT);

open(OUT, "> C:/Program Files/Perl
Express/Input/spain_tcp_destport_count.txt") || die " It’s
impossible to open the file\n\n";
foreach $k(keys %tcp_d_h) {print OUT "$k\t$tcp_d_h{$k}\n";}
close(OUT);

open(OUT, "> C:/Program Files/Perl
Express/Input/spain_udp_sourceport_count.txt") || die " It’s
impossible to open the file\n\n";
foreach $k(keys %udp_s_h) {print OUT "$k\t$udp_s_h{$k}\n";}
close(OUT);
```

```
open(OUT, "> C:/Program Files/Perl
Express/Input/spain_udp_destport_count.txt") || die "It’s
impossible to open the file\n\n";
foreach $k(keys %udp_d_h) {print OUT "$k\t$udp_d_h{$k}\n";}
close(OUT);
```

4. Network population for every minute

FIRST SCRIPT

```
#!/usr/bin/perl

#this script aims the elimination of the reference source IP
address

#insert the source IP according to the traces that you are going
to eliminate
#$ip_rif = '147.83.130.144';  #Spain
$ip_rif = '192.168.1.101';    #Romania

#file opening
open(IN, "< C:/Program Files/Perl
Express/Input/romania_timestamp.txt") || die "It’s impossible to
open the file\n\n";
open(OUT, "> C:/Program Files/Perl
Express/Input/romania_timestamp2.txt") || die "It’s impossible to
open the file\n\n";

while(<IN>) {
chomp;
(my $s_ip, my $d_ip, my $timestamp) = split;
#control instruction
if(  (!($s_ip)) || (!($d_ip)) || (!($timestamp)) || ( ($s_ip) eq
'0.0.0.0') || ( ($d_ip) eq '0.0.0.0') || ( ($s_ip) eq 'NA') || (
($d_ip) eq 'NA') ) { next;}
#hashes loading
if($s_ip eq $ip_rif)
      { #upload
        print OUT "$d_ip\t$timestamp\n"; }
else
      {  #download
        print OUT "$s_ip\t$timestamp\n"; }
}
#files closing
close(IN);
close (OUT);
```

SECOND SCRIPT

```
#!/usr/bin/perl
my %minute; #it keeps all IP destination address appeareed during
one minute
my $time=60; #it keeps the time session, it has increased of 60
units per time
my $i=59;  #minuts counter referred to the Romania’s traces

open(IN, "< C:/Program Files/Perl
Express/Input/romania_timestamp2.txt") || die "It’s impossible to
open the file\n\n";
open(OUT, "> C:/Program Files/Perl
Express/Input/romania_minutes.txt") || die "It’s impossible to
open the file\n\n";
while(<IN>) {
chomp;
(my $ip, my $timestamp) = split;
if(  (!($ip)) || (!($timestamp)) || ( ($ip) eq '0.0.0.0') || (
($ip) eq 'NA') ) { next;}
if ($timestamp<= $time){
      $minute{$ip}= 1;
}
else {
      $distinti= scalar keys %minute; #calculus of the hash
      elements
      print OUT "$i\t$distinti\n";
      $time+=60;
      $i++;
      %minute= (); #the temporary hash is emptied
}}
#files closing
close(IN);
close(OUT);
```

5. T calculus

```
#!/usr/bin/perl

#insert the source IP according to the traces that you are going
to analyze
$ip_rif = '147.83.130.144';  #Spain
#$ip_rif = '192.168.1.101';  #Romania

my @timestamp; #it keeps the duration times for all sessions
my @interarrivo; #it contains all Ti

#*************************  TCP  *************************
#TCP hash definition
my %down_d_ip; #it keeps all IP source address in download

#TCP files opening
```

```
open(IN, "< C:/Program Files/Perl Express/Input/spain_T_tcp.txt")
|| die "It’s impossible to open the file\n\n";
#open(IN, "< C:/Program Files/Perl
Express/Input/romania_T_tcp.txt") || die ""It’s impossible to open
the file\n\n";

while(<IN>) {
chomp;
(my $s_ip, my $d_ip, my $s_port, my $d_port, my $byte, my $tmp) =
split;
#control instruction
if(  (!($s_ip)) || (!($d_ip)) || (!($s_port)) || (!($d_port)) ||
(!($byte)) || ( ($s_ip) eq '0.0.0.0') || ( ($d_ip) eq '0.0.0.0')
|| ( ($s_ip) eq 'NA') || ( ($d_ip) eq 'NA') ) { next;}

#hashes loading
my @vettore;
push @vettore, $byte;
push @vettore, $tmp;
my $punt = \@vettore;

if(!($s_ip eq $ip_rif))
      {  # download
      $down_d_ip{$s_ip}{$s_port}{$d_port}=[]unless exists
      $down_d_ip{$s_ip}{$s_port}{$d_port};
      push @{$down_d_ip{$s_ip}{$s_port}{$d_port}}, $punt;
      $ip{$s_ip}=1;
      }
}
close(IN);

#ciclo sui source ip address  ---> TCP download traffic
foreach my $a( keys %down_d_ip)
{       #cicle on source ports
      foreach my $b( keys %{$down_d_ip{$a} } )
        {       #cicle on destination ports
            foreach my $c( keys %{$down_d_ip{$a}{$b}  } )
```

```
                {
                  my $count=0;
                  my $somma=0;
                  my $limit =$#{$down_d_ip{$a}{$b}{$c}};
                  for (my $i=0; $i<= $limit; $i++)
                        {
                         if(${$down_d_ip{$a}{$b}{$c}}[$i][0]>1000)
                              {$count++;}
                         $somma +=$down_d_ip{$a}{$b}{$c}[$i][0];
                        }
                        if ($count>10)
                        {
                        my @inter;
            for (my $i=0; $i<=$limit; $i++)
            {
            $inter[$i] = ${$down_d_ip{$a}{$b}{$c}}[$i][1]}
                        my $limit2=$#inter;
                        my $start = $inter[0];
                        my $end = $inter[$limit2];
                        my $ti = $end-$start;
                        my $t = $ti/$limit;
                        push @timestamp, $t;
                        my $int=0;
                        for (my $i=1; $i<=$limit; $i++)
                                {
                                  $int= $inter[$i]-$inter[$i-1];
                                  push @interarrivo, $int;
}}}}}

#***************************  UDP  ***************************
#UDP hash definition
my %down_d_ip; #it keeps all IP source address in download

#UDP files opening
open(IN, "< C:/Program Files/Perl Express/Input/spain_T_udp.txt")
|| die "It’s impossible to open the file\n\n";
```

```
#open(IN, "< C:/Program Files/Perl
Express/Input/romania_T_udp.txt") || die "It’s impossible to open
the file\n\n";

while(<IN>) {
chomp;
(my $s_ip, my $d_ip, my $s_port, my $d_port, my $byte, my $tmp) =
split;
#control instruction
if(  (!($s_ip)) || (!($d_ip)) || (!($s_port)) || (!($d_port)) ||
(!($byte)) || ( ($s_ip) eq '0.0.0.0') || ( ($d_ip) eq '0.0.0.0')
|| ( ($s_ip) eq 'NA') || ( ($d_ip) eq 'NA') ) { next;}

#hashes loading
my @vettore;
push @vettore, $byte;
push @vettore, $tmp;
my $punt = \@vettore;

if(!($s_ip eq $ip_rif))
      {  #download
      $down_d_ip{$s_ip}{$s_port}{$d_port}=[]unless exists
      $down_d_ip{$s_ip}{$s_port}{$d_port};
      push @{$down_d_ip{$s_ip}{$s_port}{$d_port}}, $punt;
      $ip{$s_ip}=1;
      }
}
close(IN);

#ciclo sui source ip address  ---> UDP download traffic
foreach my $a( keys %down_d_ip)
{       #cicle on source ports
      foreach my $b( keys %{$down_d_ip{$a} } )
        {       #cicle on destination ports
            foreach my $c( keys %{$down_d_ip{$a}{$b}  } )
                {
                  my $count=0;
                  my $somma=0;
                  my $limit =$#{$down_d_ip{$a}{$b}{$c}};
                  for (my $i=0; $i<= $limit; $i++)
                        {
                         if(${$down_d_ip{$a}{$b}{$c}}[$i][0]>1000)
                              {$count++;}
                         $somma +=$down_d_ip{$a}{$b}{$c}[$i][0];
                        }
                        if ($count>10)
                        {
                        my @inter;
            for (my $i=0; $i<=$limit; $i++)
            {
            $inter[$i] = ${$down_d_ip{$a}{$b}{$c}}[$i][1]}
                        my $limit2=$#inter;
                        my $start = $inter[0];
                        my $end = $inter[$limit2];
                        my $ti = $end-$start;
                        my $t = $ti/$limit;
                        push @timestamp, $t;
                        my $int=0;
                        for (my $i=1; $i<=$limit; $i++)
                                {
                                  $int= $inter[$i]-$inter[$i-1];
                                  push @interarrivo, $int;
}}}}}

*************************** PRINTING ***************************
my $t_sum;
my $elements= scalar @timestamp;
foreach $k (@timestamp) { $t_sum+=$k;}
my $T = $t_sum/$elements;
print "duration time of a session: T =  $T\n\n";
open(OUT, "> C:/Program Files/Perl
Express/Input/Spain_interarrivo.txt") || die “It’s impossible to
open the file\n\n";
```

```
#open(OUT, "> C:/Program Files/Perl
Express/Input/Romania_interarrivo.txt") || die "It’s impossible to
open the file\n\n";
foreach my $d (@interarrivo) { print OUT "$d\n";}
```

6. DATA calculus

```
#!/usr/bin/perl

#insert the source IP according to the traces that you are going
to analyze
$ip_rif = '147.83.130.144 ';   #Spain
#$ip_rif = '192.168.1.101';    #Romania

my $header=34;
my %ip; #tiene traccia di tutti gli ip

#*************************** TCP  *************************
#TCP hashes definition
my %down_d_ip; #it keeps all IP source address in download

open(IN, "< C:/Program Files/Perl Express/Input/TCP_only.txt") ||
die "It’s impossible to open the file\n\n";
while(<IN>) {
chomp;
(my $s_ip, my $d_ip, my $s_port, my $d_port, my $byte) = split;
if(  (!($s_ip)) || (!($d_ip)) || (!($s_port)) || (!($d_port)) ||
(!($byte)) || ( ($s_ip) eq '0.0.0.0') || ( ($d_ip) eq '0.0.0.0')
|| ( ($s_ip) eq 'NA') || ( ($d_ip) eq 'NA') ) { next;}
#hash loading
if(!($s_ip eq $ip_rif))
      {  #download
      $down_d_ip{$s_ip}{$s_port}{$d_port}=[]unless exists
      $down_d_ip{$s_ip}{$s_port}{$d_port};
      push @{$down_d_ip{$s_ip}{$s_port}{$d_port}}, $byte;
      $ip{$s_ip}=1;
      }
}
close(IN);

#****************** PRINTING OF ALL IP ADDRESSES ***************
my $TCP_cont_media=0;
my $TCP_sum_media=0;
#cicle on IP address  ---> TCP download traffic
foreach my $a( keys %down_d_ip)
{       #cicle on source ports
      foreach my $b( keys %{$down_d_ip{$a} } )
        {       #cicle on destination ports
            foreach my $c( keys %{$down_d_ip{$a}{$b}  } )
                {
                 my $count=0;
                 my $somma=0;
                 my $i=0;
                 for ($i=0; $i<= $#{$down_d_ip{$a}{$b}{$c}}; $i++)
                        {
                        if  (${$down_d_ip{$a}{$b}{$c}}[$i]>1000)
                             {$count++;}
                        $somma +=$down_d_ip{$a}{$b}{$c}[$i];
                        }
                  if ($count>10) {
                        $TCP_sum_media+=$somma;
                        $TCP_cont_media+=$i;
}}}}

#************************** UDP  ************************
#UDP hashes definition
my %down_d_ip; #it keeps all IP source address in download
open(IN, "< C:/Program Files/Perl Express/Input/UDP_only.txt") ||
die "It’s impossible to open the file\n\n";
while(<IN>) {
chomp;
```

```
(my $s_ip, my $d_ip, my $s_port, my $d_port, my $byte) = split;
if(  (!($s_ip)) || (!($d_ip)) || (!($s_port)) || (!($d_port)) ||
(!($byte)) || ( ($s_ip) eq '0.0.0.0') || ( ($d_ip) eq '0.0.0.0')
|| ( ($s_ip) eq 'NA') || ( ($d_ip) eq 'NA') ) { next;}
#hash loading
if(!($s_ip eq $ip_rif))
      {  #download
      $down_d_ip{$s_ip}{$s_port}{$d_port}=[]unless exists
      $down_d_ip{$s_ip}{$s_port}{$d_port};
      push @{$down_d_ip{$s_ip}{$s_port}{$d_port}}, $byte;
      $ip{$s_ip}=1;
      }
}
close(IN);

#******************* PRINTING OF ALL IP ADDRESSES ***************
my $UDP_cont_media=0;
my $UDP_sum_media=0;
#cicle on IP address  ---> UDP download traffic
foreach my $a( keys %down_d_ip)
{       #cicle on source ports
      foreach my $b( keys %{$down_d_ip{$a} } )
        {       #cicle on destination ports
            foreach my $c( keys %{$down_d_ip{$a}{$b}  } )
                {
                 my $count=0;
                 my $somma=0;
                 my $i=0;
                 for ($i=0; $i<= $#{$down_d_ip{$a}{$b}{$c}}; $i++)
                        {
                        if  (${$down_d_ip{$a}{$b}{$c}}[$i]>1000)
                              {$count++;}
                        $somma +=$down_d_ip{$a}{$b}{$c}[$i];
                        }
                  if ($count>10) {
                        $UDP_sum_media+=$somma;
                        $UDP_cont_media+=$i;
}}}}


#************************ PRINTING ***************************
open(IN, "> C:/Program Files/Perl Express/Input/spain_media.txt")
|| die "It’s impossible to open the file\n\n";

my $TCP_average;
if ($TCP_cont_media==0){$TCP_average=0;}
else {$TCP_average= ($TCP_sum_media/$TCP_cont_media)-$header;}
my $UDP_average;
if ($UDP_cont_media==0) {$UDP_average=0;}
else {$UDP_average= ($UDP_sum_media/$UDP_cont_media)-$header;}
my $total_sum= $UDP_sum_media+$TCP_sum_media;
my $total_count= $TCP_cont_media+$UDP_cont_media;
my $total_average;
if ($total_count==0) {$total_average=0;}
else {$total_average= ($total_sum/$total_count)-$header;}

print IN "SPAIN: Calculus about video download traffic\n\n";
print IN "TCP\n sum = $TCP_sum_media\n num_packet =
$TCP_cont_media\n average = $TCP_average\n\n";
print IN "UDP\n sum = $UDP_sum_media\n num_packet =
$UDP_cont_media\n average = $UDP_average\n\n";
print IN "TOTAL\n sum = $total_sum\n num_packet = $total_count\n
average = $total_average";
close(IN);

open(IN, "> C:/Program Files/Perl
Express/Input/romania_media.txt") || die "It’s impossible to open
the file\n\n";

my $TCP_average;
if ($TCP_cont_media==0){$TCP_average=0;}
else {$TCP_average= ($TCP_sum_media/$TCP_cont_media)-$header;}
my $UDP_average;
if ($UDP_cont_media==0) {$UDP_average=0;}
```

```
else {$UDP_average= ($UDP_sum_media/$UDP_cont_media)-$header;}
my $total_sum= $UDP_sum_media+$TCP_sum_media;
my $total_count= $TCP_cont_media+$UDP_cont_media;
my $total_average;
if ($total_count==0) {$total_average=0;}
else {$total_average= ($total_sum/$total_count)-$header;}

print IN "ROMANIA: Calculus about video download traffic\n\n";
print IN "TCP\n sum = $TCP_sum_media\n num_packet =
$TCP_cont_media\n average = $TCP_average\n\n";
print IN "UDP\n sum = $UDP_sum_media\n num_packet =
$UDP_cont_media\n average = $UDP_average\n\n";
print IN "TOTAL\n sum = $total_sum\n num_packet = $total_count\n
average = $total_average";
close(IN);
```

APPENDIX B

Implemented M-files

In this second appendix all M-files produced during this work are reported according to the order of their using in the sixth chapter. Some comments are writing in order to help the comprehension.

1. UDP-TCP, Upload-Download, Control-Data traffic division

```
%loading of TCP text files
%Romania
load video_download_TCP.txt;
load video_upload_TCP.txt;
load control_download_TCP.txt;
load control_upload_TCP.txt;
load total_download_TCP.txt;
load total_upload_TCP.txt;
%Spain
load TCP_down_video.txt;
load TCP_up_video.txt;
load TCP_down_control.txt;
load TCP_up_control.txt;
load total_down_TCP.txt;
load total_up_TCP.txt;

figure(1);
p=cdfplot(video_download_TCP);
%a 5% of the range is added on y-axis in order to obtain a better
visualization, so it is written 1.00075
```

```
axis([10000 100000000 0.94 1.00075]);
grid off; %it takes off the grid
set(gca,'XScale','log'); %it sets a logarithmic scale on x-axis
set(p,'Color','Blue'); %line colour
hold on;
p=cdfplot(TCP_down_video);
set(p,'Color','Red');
grid off;
legend('Romania','Spain','Location','NorthWest');
title('TCP video download');
xlabel('Traffic [bytes]'); %it sets a label on x-axis
ylabel('CDF'); %it sets a label on y-axis

saveas(gcf,'t-video_download_TCP.pdf'); %save as pdf
saveas(gcf,'t-video_download_TCP.png'); %save as png
saveas(gcf,'t-video_download_TCP.fig'); %save as fig
saveas(gcf,'t-video_download_TCP.eps'); %save as eps
saveas(gcf,'t-video_download_TCP.jpg'); %save as jpg

figure(2);
p=cdfplot(video_upload_TCP);
axis([10000 100000000 0.94 1.00075]);
grid off;
set(gca,'XScale','log');
set(p,'Color','Blue');
hold on;
p=cdfplot(TCP_up_video);
set(p,'Color','Red');
grid off;
legend('Romania','Spain','Location','NorthWest');
title('TCP video upload');
xlabel('Traffic [bytes]');
ylabel('CDF');
saveas(gcf,'t-video_upload_TCP.pdf');
saveas(gcf,'t-video_upload_TCP.png');
saveas(gcf,'t-video_upload_TCP.fig');
saveas(gcf,'t-video_upload_TCP.eps');
saveas(gcf,'t-video_upload_TCP.jpg');

figure(3);
p=cdfplot(control_download_TCP);
axis([1 100000000 0.94 1.00075]);
set(p,'Color','Blue');
grid off;
set(gca,'XScale','log');
hold on;
p=cdfplot(TCP_down_control);
set(p,'Color','Red');
grid off;
legend('Romania','Spain','Location','NorthWest');
title('TCP control download');
xlabel('Traffic [bytes]');
ylabel('CDF');
saveas(gcf,'t-control_download_TCP.pdf');
saveas(gcf,'t-control_download_TCP.png');
saveas(gcf,'t-control_download_TCP.fig');
saveas(gcf,'t-control_download_TCP.eps');
saveas(gcf,'t-control_download_TCP.jpg');

figure(4);
p=cdfplot(control_upload_TCP);
axis([1 100000000 0.94 1.00075]);
set(p,'Color','Blue');
grid off;
set(gca,'XScale','log');
hold on;
p=cdfplot(TCP_up_control);
set(p,'Color','Red');
grid off;
legend('Romania','Spain','Location','NorthWest');
title('TCP control upload');
xlabel('Traffic [bytes]');
ylabel('CDF');
saveas(gcf,'t-control_upload_TCP.pdf');
saveas(gcf,'t-control_upload_TCP.png');
saveas(gcf,'t-control_upload_TCP.fig');
saveas(gcf,'t-control_upload_TCP.eps');
saveas(gcf,'t-control_upload_TCP.jpg');

figure(5);
p=cdfplot(total_download_TCP);
axis([1 100000000 0.94 1.00075]);
set(p,'Color','Blue');
grid off;
set(gca,'XScale','log');
hold on;
p=cdfplot(total_down_TCP);
set(p,'Color','Red');
grid off;
legend('Romania','Spain','Location','NorthWest');
title('TCP total download');
xlabel('Traffic [bytes]');
ylabel('CDF');
saveas(gcf,'t-total_download_TCP.pdf');
saveas(gcf,'t-total_download_TCP.png');
saveas(gcf,'t-total_download_TCP.fig');
saveas(gcf,'t-total_download_TCP.eps');
saveas(gcf,'t-total_download_TCP.jpg');

figure(6);
p=cdfplot(total_upload_TCP);
axis([1 100000000 0.94 1.00075]);
set(p,'Color','Blue');
grid off;
set(gca,'XScale','log');
hold on;
p=cdfplot(total_up_TCP);
set(p,'Color','Red');
```

```
grid off;
legend('Romania','Spain','Location','NorthWest');
title('TCP total upload');
xlabel('Traffic [bytes]');
ylabel('CDF');
saveas(gcf,'t-total_upload_TCP.pdf');
saveas(gcf,'t-total_upload_TCP.png');
saveas(gcf,'t-total_upload_TCP.fig');
saveas(gcf,'t-total_upload_TCP.eps');
saveas(gcf,'t-total_upload_TCP.jpg');


%loading of UDP text files
%Romania
load video_download_UDP.txt;
load video_upload_UDP.txt;
load control_download_UDP.txt;
load control_upload_UDP.txt;
load total_download_UDP.txt;
load total_upload_UDP.txt;
%Spain
load UDP_down_video.txt;
load UDP_up_video.txt;
load UDP_down_control.txt;
load UDP_up_control.txt;
load total_down_UDP.txt;
load total_up_UDP.txt;

figure(7);
p=cdfplot(video_download_UDP);
%a 5% of the range is added on y-axis in order to obtain a better
visualization, so it is written 1.00075
axis([10000 100000000 0.94 1.00075]);
grid off; %it takes off the grid
set(gca,'XScale','log'); %it sets a logarithmic scale on x-axis
set(p,'Color','Blue'); %line colour
hold on;
p=cdfplot(UDP_down_video);
set(p,'Color','Red');
grid off;
legend('Romania','Spain','Location','NorthWest');
title('UDP video download');
xlabel('Traffic [bytes]'); %it sets a label on x-axis
ylabel('CDF'); %it sets a label on y-axis

saveas(gcf,'t-video_download_UDP.pdf'); %save as pdf
saveas(gcf,'t-video_download_UDP.png'); %save as png
saveas(gcf,'t-video_download_UDP.fig'); %save as fig
saveas(gcf,'t-video_download_UDP.eps'); %save as eps
saveas(gcf,'t-video_download_UDP.jpg'); %save as jpg


figure(8);
p=cdfplot(video_upload_UDP);
```

```
axis([10000 100000000 0.94 1.00075]);
grid off;
set(gca,'XScale','log');
set(p,'Color','Blue');
hold on;
p=cdfplot(UDP_up_video);
set(p,'Color','Red');
grid off;
legend('Romania','Spain','Location','NorthWest');
title('UDP video upload');
xlabel('Traffic [bytes]');
ylabel('CDF');
saveas(gcf,'t-video_upload_UDP.pdf');
saveas(gcf,'t-video_upload_UDP.png');
saveas(gcf,'t-video_upload_UDP.fig');
saveas(gcf,'t-video_upload_UDP.eps');
saveas(gcf,'t-video_upload_UDP.jpg');

figure(9);
p=cdfplot(control_download_UDP);
axis([1 100000000 0.94 1.00075]);
set(p,'Color','Blue');
grid off;
set(gca,'XScale','log');
hold on;
p=cdfplot(UDP_down_control);
set(p,'Color','Red');
grid off;
legend('Romania','Spain','Location','NorthWest');
title('UDP control download');
xlabel('Traffic [bytes]');
ylabel('CDF');
saveas(gcf,'t-control_download_UDP.pdf');
saveas(gcf,'t-control_download_UDP.png');
saveas(gcf,'t-control_download_UDP.fig');
saveas(gcf,'t-control_download_UDP.eps');
saveas(gcf,'t-control_download_UDP.jpg');

figure(10);
p=cdfplot(control_upload_UDP);
axis([1 100000000 0.94 1.00075]);
set(p,'Color','Blue');
grid off;
set(gca,'XScale','log');
hold on;
p=cdfplot(UDP_up_control);
set(p,'Color','Red');
grid off;
legend('Romania','Spain','Location','NorthWest');
title('UDP control upload');
xlabel('Traffic [bytes]');
ylabel('CDF');
saveas(gcf,'t-control_upload_UDP.pdf');
saveas(gcf,'t-control_upload_UDP.png');
saveas(gcf,'t-control_upload_UDP.fig');
saveas(gcf,'t-control_upload_UDP.eps');
saveas(gcf,'t-control_upload_UDP.jpg');

figure(11);
p=cdfplot(total_download_UDP);
axis([1 100000000 0.94 1.00075]);
set(p,'Color','Blue');
grid off;
set(gca,'XScale','log');
hold on;
p=cdfplot(total_down_UDP);
set(p,'Color','Red');
grid off;
legend('Romania','Spain','Location','NorthWest');
title('UDP total download');
xlabel('Traffic [bytes]');
ylabel('CDF');
saveas(gcf,'t-total_download_UDP.pdf');
saveas(gcf,'t-total_download_UDP.png');
saveas(gcf,'t-total_download_UDP.fig');
saveas(gcf,'t-total_download_UDP.eps');
saveas(gcf,'t-total_download_UDP.jpg');

figure(12);
p=cdfplot(total_upload_UDP);
axis([1 100000000 0.94 1.00075]);
set(p,'Color','Blue');
grid off;
set(gca,'XScale','log');
hold on;
p=cdfplot(total_up_UDP);
set(p,'Color','Red');
grid off;
legend('Romania','Spain','Location','NorthWest');
title('UDP total upload');
xlabel('Traffic [bytes]');
ylabel('CDF');
saveas(gcf,'t-total_upload_UDP.pdf');
saveas(gcf,'t-total_upload_UDP.png');
saveas(gcf,'t-total_upload_UDP.fig');
saveas(gcf,'t-total_upload_UDP.eps');
saveas(gcf,'t-total_upload_UDP.jpg');
```

2. Download/upload balance

```
%Spain
load TCP_down_control.txt;
load TCP_up_control.txt;
load TCP_down_video.txt;
load TCP_up_video.txt;
load UDP_down_control.txt;
load UDP_up_control.txt;
load UDP_down_video.txt;
load UDP_up_video.txt;
load total_down_TCP.txt;
load total_up_TCP.txt;
load total_down_UDP.txt;
load total_up_UDP.txt;

figure(1);
x =(TCP_up_video);
y = (TCP_down_video);
p= plot(x,y,'.');
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
axis([10000 100000000 10000 100000000]);
title('Spain - TCP video: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
hold on;
j=1:1000:100000000;
p= plot(j,j);
set(p,'Color','Black');
grid off;
saveas(gcf,'Spain-TCP_video.pdf');
saveas(gcf,'Spain-TCP_video.png');
saveas(gcf,'Spain-TCP_video.fig');
saveas(gcf,'Spain-TCP_video.eps');
saveas(gcf,'Spain-TCP_video.jpg');

figure(2);
x =(TCP_up_control);
y = (TCP_down_control);
p= plot(x,y,'.');
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
title('Spain - TCP control: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
hold on;
```

```
j=1:1000:100000000;
p= plot(j,j);
set(p,'Color','Black');
grid off;
saveas(gcf,'Spain-TCP_control.pdf');
saveas(gcf,'Spain-TCP_control.png');
saveas(gcf,'Spain-TCP_control.fig');
saveas(gcf,'Spain-TCP_control.eps');
saveas(gcf,'Spain-TCP_control.jpg');

figure(3);
x =(total_up_TCP);
y = (total_down_TCP);
p= plot(x,y,'.');
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
title('Spain - TCP total: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
hold on;
j=1:1000:100000000;
p= plot(j,j);
set(p,'Color','Black');
grid off;
saveas(gcf,'Spain-TCP_total.pdf');
saveas(gcf,'Spain-TCP_total.png');
saveas(gcf,'Spain-TCP_total.fig');
saveas(gcf,'Spain-TCP_total.eps');
saveas(gcf,'Spain-TCP_total.jpg');

figure(4);
x =(UDP_up_video);
y = (UDP_down_video);
p= plot(x,y,'.');
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
axis([10000 100000000 10000 100000000]);
title('Spain - UDP video: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
hold on;
j=1:1000:100000000;
p= plot(j,j);
axis([0 50000000 0 50000000]);
set(p,'Color','Black');
grid off;
saveas(gcf,'Spain-UDP_video.pdf');
saveas(gcf,'Spain-UDP_video.png');
saveas(gcf,'Spain-UDP_video.fig');
saveas(gcf,'Spain-UDP_video.eps');
saveas(gcf,'Spain-UDP_video.jpg');
```

```
figure(5);
x =(UDP_up_control);
y = (UDP_down_control);
p= plot(x,y,'.');
axis([0 50000000 0 50000000]);
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
title('Spain - UDP control: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
hold on;
j=1:1000:100000000;
p= plot(j,j);
axis([0 50000000 0 50000000]);
set(p,'Color','Black');
grid off;
saveas(gcf,'Spain-UDP_control.pdf');
saveas(gcf,'Spain-UDP_control.png');
saveas(gcf,'Spain-UDP_control.fig');
saveas(gcf,'Spain-UDP_control.eps');
saveas(gcf,'Spain-UDP_control.jpg');

figure(6);
x =(total_up_UDP);
y = (total_down_UDP);
p= plot(x,y,'.');
axis([0 50000000 0 50000000]);
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
title('Spain - UDP total: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
hold on;
j=1:1000:100000000;
p= plot(j,j);
set(p,'Color','Black');
grid off;
saveas(gcf,'Spain-UDP_total.pdf');
saveas(gcf,'Spain-UDP_total.png');
saveas(gcf,'Spain-UDP_total.fig');
saveas(gcf,'Spain-UDP_total.eps');
saveas(gcf,'Spain-UDP_total.jpg');


%Romania
load TCP_down_control.txt;
load TCP_up_control.txt;
load TCP_down_video.txt;
load TCP_up_video.txt;
load UDP_down_control.txt;
load UDP_up_control.txt;
load UDP_down_video.txt;
load UDP_up_video.txt;
load total_down_TCP.txt;
load total_up_TCP.txt;
load total_down_UDP.txt;
load total_up_UDP.txt;

figure(1);
x =(TCP_up_video);
y = (TCP_down_video);
p= plot(x,y,'.');
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
axis([10000 100000000 10000 100000000]);
title('Romania - TCP video: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
hold on;
j=1:1000:100000000;
p= plot(j,j);
set(p,'Color','Black');
grid off;
saveas(gcf,'Romania-TCP_video.pdf');
saveas(gcf,'Romania-TCP_video.png');
saveas(gcf,'Romania-TCP_video.fig');
saveas(gcf,'Romania-TCP_video.eps');
saveas(gcf,'Romania-TCP_video.jpg');

figure(2);
x =(TCP_up_control);
y = (TCP_down_control);
p= plot(x,y,'.');
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
title('Romania - TCP control: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
hold on;
j=1:1000:100000000;
p= plot(j,j);
set(p,'Color','Black');
grid off;
saveas(gcf,'Romania-TCP_control.pdf');
saveas(gcf,'Romania-TCP_control.png');
saveas(gcf,'Romania-TCP_control.fig');
saveas(gcf,'Romania-TCP_control.eps');
saveas(gcf,'Romania-TCP_control.jpg');

figure(3);
x =(total_up_TCP);
```

```
y = (total_down_TCP);
p= plot(x,y,'.');
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
title('Romania - TCP total: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
hold on;
j=1:1000:100000000;
p= plot(j,j);
set(p,'Color','Black');
grid off;
saveas(gcf,'Romania-TCP_total.pdf');
saveas(gcf,'Romania-TCP_total.png');
saveas(gcf,'Romania-TCP_total.fig');
saveas(gcf,'Romania-TCP_total.eps');
saveas(gcf,'Romania-TCP_total.jpg');

figure(4);
x =(UDP_up_video);
y = (UDP_down_video);
p= plot(x,y,'.');
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
axis([10000 100000000 10000 100000000]);
title('Romania - UDP video: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
hold on;
j=1:1000:100000000;
p= plot(j,j);
set(p,'Color','Black');
grid off;
saveas(gcf,'Romania-UDP_video.pdf');
saveas(gcf,'Romania-UDP_video.png');
saveas(gcf,'Romania-UDP_video.fig');
saveas(gcf,'Romania-UDP_video.eps');
saveas(gcf,'Romania-UDP_video.jpg');

figure(5);
x =(UDP_up_control);
y = (UDP_down_control);
p= plot(x,y,'.');
axis([0 50000000 0 50000000]);
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
title('Romania - UDP control: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
```

```
hold on;
j=1:1000:100000000;
p= plot(j,j);
axis([0 50000000 0 50000000]);
set(p,'Color','Black');
grid off;
saveas(gcf,'Romania-UDP_control.pdf');
saveas(gcf,'Romania-UDP_control.png');
saveas(gcf,'Romania-UDP_control.fig');
saveas(gcf,'Romania-UDP_control.eps');
saveas(gcf,'Romania-UDP_control.jpg');

figure(6);
x =(total_up_UDP);
y = (total_down_UDP);
p= plot(x,y,'.');
axis([0 50000000 0 50000000]);
grid off;
set(gca,'XScale','log');
set(gca,'YScale','log');
title('Romania - UDP total: download/upload balance');
xlabel('sent bytes (upload)');
ylabel('received bytes (download)');
set(p,'Color','Blue');
hold on;
j=1:1000:100000000;
p= plot(j,j);
set(p,'Color','Black');
grid off;
saveas(gcf,'Romania-UDP_total.pdf');
saveas(gcf,'Romania-UDP_total.png');
saveas(gcf,'Romania-UDP_total.fig');
saveas(gcf,'Romania-UDP_total.eps');
saveas(gcf,'Romania-UDP_total.jpg');
```

3. Intra-domain and inter-domain traffic division

```
load SIeI_TCP_up_video.txt;
load RIeI_TCP_up_video.txt;

figure(1);
p=cdfplot(SIeI_TCP_up_video(:,1));
axis([1 100000000 0 1.05]);
grid off;

set(gca,'XScale','log');
set(p,'Color','Blue');
hold on;
p=cdfplot(SIeI_TCP_up_video(:,2));
set(p,'Color','Red');
grid off;
hold on;
p=cdfplot(RIeI_TCP_up_video(:,1));
set(p,'Color','Yellow');
grid off;
hold on;
p=cdfplot(RIeI_TCP_up_video(:,2));
set(p,'Color','Green');
grid off;

xlabel('Traffic [bytes]');
ylabel('CDF');
title('TCP video upload');
legend('Spain intra domain traffic CDF','Spain inter domain
traffic CDF','Romania intra domain traffic CDF','Romania inter
domain traffic CDF','Location','NorthWest');
saveas(gcf,'tot_IeI_TCP_up_video.pdf');
saveas(gcf,'tot_IeI_TCP_up_video.png');
saveas(gcf,'tot_IeI_TCP_up_video.fig');
saveas(gcf,'tot_IeI_TCP_up_video.eps');
saveas(gcf,'tot_IeI_TCP_up_video.jpg');
```

4. Ports' utilization

```
load romania_tcp_destport_count.txt;
load romania_tcp_sourceport_count.txt;
load spain_tcp_destport_count.txt;
load spain_tcp_sourceport_count.txt;
```

```
%TCP destination ports
figure(1);
x =(romania_tcp_destport_count(:,1));
y =(romania_tcp_destport_count(:,2));
p = plot(x,y,'.');
grid off;
set(gca,'YScale','log');
title('TCP destination ports utilization');
xlabel('port number');
ylabel('utilization [times]');
set(p,'Color','Blue');
hold on; %it allows the plotting of several lines on the same
graph
x =(spain_tcp_destport_count(:,1));
y =(spain_tcp_destport_count(:,2));
p = plot(x,y,'.');
set(p,'Color','Red');
grid off;

legend('Romania','Spain','Location','NorthEast');
saveas(gcf,'TCP_dest_ports.pdf');
saveas(gcf,'TCP_dest_ports.png');
saveas(gcf,'TCP_dest_ports.fig');
saveas(gcf,'TCP_dest_ports.eps');
saveas(gcf,'TCP_dest_ports.jpg');

%TCP source ports
figure(2);
x =(romania_tcp_sourceport_count(:,1));
y =(romania_tcp_sourceport_count(:,2));
p = plot(x,y,'.');
grid off;
set(gca,'YScale','log');
title('TCP source ports utilization');
xlabel('port number');
ylabel('utilization [times]');
set(p,'Color','Blue');
hold on;
x =(spain_tcp_sourceport_count(:,1));
y =(spain_tcp_sourceport_count(:,2));
p = plot(x,y,'.');
set(p,'Color','Red');
grid off;

legend('Romania','Spain','Location','NorthEast');
saveas(gcf,'TCP_source_ports.pdf');
saveas(gcf,'TCP_source_ports.png');
saveas(gcf,'TCP_source_ports.fig');
saveas(gcf,'TCP_source_ports.eps');
saveas(gcf,'TCP_source_ports.jpg');


load romania_udp_destport_count.txt;
load romania_udp_sourceport_count.txt;
load spain_udp_destport_count.txt;
load spain_udp_sourceport_count.txt;

%UDP destination port
figure(3);
x =(romania_udp_destport_count(:,1));
y =(romania_udp_destport_count(:,2));
p = plot(x,y,'.');
grid off;
set(gca,'YScale','log');
title('UDP destination ports utilization');
xlabel('port number');
ylabel('utilization [times]');
set(p,'Color','Blue');
hold on;
x =(spain_udp_destport_count(:,1));
y =(spain_udp_destport_count(:,2));
p = plot(x,y,'.');
set(p,'Color','Red');
grid off;

legend('Romania','Spain','Location','NorthEast');
saveas(gcf,'UDP_dest_ports.pdf');
saveas(gcf,'UDP_dest_ports.png');
saveas(gcf,'UDP_dest_ports.fig');
saveas(gcf,'UDP_dest_ports.eps');
saveas(gcf,'UDP_dest_ports.jpg');

%UDP source port
figure(4);
x =(romania_udp_sourceport_count(:,1));
y =(romania_udp_sourceport_count(:,2));
p = plot(x,y,'.');
grid off;
set(gca,'YScale','log');

title('UDP source ports utilization');
xlabel('port number');
ylabel('utilization [times]');
set(p,'Color','Blue');
hold on;
x =(spain_udp_sourceport_count(:,1));
y =(spain_udp_sourceport_count(:,2));
p = plot(x,y,'.');
set(p,'Color','Red');
grid off;

legend('Romania','Spain','Location','NorthEast');
saveas(gcf,'UDP_source_ports.pdf');
saveas(gcf,'UDP_source_ports.png');
saveas(gcf,'UDP_source_ports.fig');
saveas(gcf,'UDP_source_ports.eps');
saveas(gcf,'UDP_source_ports.jpg');
```

5. Network population for every minute

```
load romania_minutes.txt;
load spain_minutes.txt;

figure(1);
x =(romania_minutes(:,1));
y = (romania_minutes(:,2));
p= plot(x,y);
grid off;

title('Network population per minute');
xlabel('Time [minutes]');
ylabel('Number of peers');
set(p,'Color','Blue');
hold on;
x = (spain_minutes(:,1));
y = (spain_minutes(:,2));
p= plot(x,y);
set(p,'Color','Red');
grid off;

legend('Romania','Spain','Location','West');
saveas(gcf,'Minute.pdf');
saveas(gcf,'Minute.png');
saveas(gcf,'Minute.fig');
saveas(gcf,'Minute.eps');
saveas(gcf,'Minute.jpg');
```

6. T representation

```
load interarrivo.txt;

figure(1);
p=cdfplot(Spain_interarrivo);
set(gca,'XScale','log');
grid off;
set(p,'Color','Blue');
xlabel('interarrival time intervals');
ylabel('interarrival time CDF');
title('interarrival time trend');
saveas(gcf,'Spain_interarrivo.pdf');
saveas(gcf,'Spain_interarrivo.png');
saveas(gcf,'Spain_interarrivo.fig');
saveas(gcf,'Spain_interarrivo.eps');
saveas(gcf,'Spain_interarrivo.jpg');
```

Bibliography

[1] http://www.crunchbase.com/company/youtube
last consultation on February 23th 2010

[2] Thomas Silverston, Olivier Fourmaux, Alessio Botta, Alberto Dainotti, Antonio Pescapé, Giorgio Ventre, Kavé Salamatian.
"*Traffic analysis of peer-to-peer IPTV communities*".
2008 Elsevier B.V. 8 November 2008

[3] Dave Kosiur.
"IP Multicasting - *The complete guide to interactive corporate networks"*.
Wiley Computer Publishing, John Wiley & Sons, Inc.
New York, Chichester, Weinheim, Brisbane, Singapore, Toronto 1998

[4] Pantaleo Mastrapasqua.
"Algoritmi di bilanciamento del carico per reti Peer-to-Peer".
Politecnico di Bari, Bari, Italy. March 19, 2007.

[5] http://support.microsoft.com/kb/291786/en-us?fr=1
last consultation on January 16th 2010

[6] Vaneet Aggarwal, Robert Calderbank, Vijay Gopalakrishnan, Rittwik Jana, K.K. Ramakrishnan, Fang Yu.
"The Effectiveness of Intelligent Scheduling for Multicast".
MM'09, Beijing, China. October 19-24, 2009.
Copyright 2009 ACM 978-1-60558-608-3/09/10

[7] http://www.eventhelix.com/realtimemantra/networking/ip_routing.htm
last consultation on January 21th 2010

[8] Dan Komosny, Vit Novotny, Miroslav Balik.
“Bandwidth Redistribution Algorithm for Single Source Multicast”.
Department of Telecommunications, Brno University of Technology,
Brno, Czech Republic.
Proceedings of the International Conference on Networking, International Conference on Systems and International Conference on Mobile.
Communications and Learning Technologies (ICNICONSMCL’06).
© 2006 IEEE

[9] Cisco Systems.
“Rendezvous Point Engineering”.
Cisco Public Information. 2009.

[10] Loránd Jakab, Albert Cabellos-Aparicio, Thomas Silverston, Jordi Domingo-Pascual.
“*CoreCast: Efficient Inter-Domain Live Streaming in the Core/Edge Separated Internet”*.
Under submission

[11] Markus Oliver Junginger, Yugyung Lee.
“A Self-Organizing Publish/Subscribe Middleware for Dynamic Peer-to-Peer Networks”.
IEEE Network, pp 38 – 43. January/February 2004

[12] https://jxta.dev.java.net/
last consultation on February 9th 2010

[13] Hary Om.
“Enhanced Unified Architecture for Video-on-demand Services”.
1-4244-0216-6/06/$20.00 ©2006 IEEE

[14] Tongqing Qiu, Zihui Ge, Seungjoon Lee, Jia Wang, Jun (Jim) Xu, Qi Zhao.
“*Modeling User Activities in a Large IPTV System”.*
IMC’09, Chicago, Illinois, USA. Copyright 2009 ACM 978-1-60558-770-7/09/11. November 4–6, 2009.

[15] Hao Yin, Xuening Liu, Feng Qiu, Ning Xia, Chuang Lin, Hui Zhang, Vyas Sekar, Geyong Min.
“*Inside the Bird’s Nest: Measurements of Large-Scale Live VoD from the 2008 Olympics”.*
IMC’09, Chicago, Illinois, USA. November 4–6, 2009.
Copyright 2009 ACM 978-1-60558-770-7/09/11

[16] Cisco Systems.
“Resource Reservation Protocol”
Internetworking Technology Overview, chapter 43. 2000

[17] Luigi Alfredo Grieco.
“MPEG-4”.
DEE- Telematics Labs, Politecnico di Bari, Bari, Italy. October 2007.

[18] Hyunseok Chang, Sugih Jamin, Wenjie Wang.
“*Live Streaming Performance of the Zattoo Network”.*
IMC’09, Chicago, Illinois, USA. Copyright 2009 ACM 978-1-60558-770-7/09/11. November 4–6, 2009.

[19] Xiaojun Heiy, Chao Liangz, Jian Liangy, Yong Liuz, Keith W. Rossy.
"A Measurement Study of a Large-Scale P2P IPTV System".
Polytechnic University, Brooklyn, NY, USA 11201

[20] Chuan Wu, Baochun Li, Shuqiao Zhao.
"Diagnosing Network-wide P2P Live Streaming Inefficiencies".
ECE, University of Toronto, Tech. Rep.. January 2009.

[21] Olivier Bonaventure.
"Scaling the Internet with LISP".
Department of Computing Science and Engineering Université catholique de Louvain (UCL), Place Sainte-Barbe, 2, B-1348, Louvain-la-Neuve - Belgium. 2009

[22] David Meyer.
"The Locator Identifier Separation Protocol (LISP)".
The Internet Protocol Journal, vol. 11, no. 1, pp. 23-36. March 2008.

[23] http://www.wireshark.org/
last consultation on February 18th 2010

[24] Ulf Lamping.
"*Wireshark Developer's Guide 30065 for Wireshark 1.2.0*".
Copyright © 2004-2008 Ulf Lamping.

[25] http://perldoc.perl.org/
last consultation on December 3rd 2009

[26] http://www.mathworks.com/products/matlab/
last consultation on December 19th 2009