VoIP Communications over WLANs

Implementation of a new downlink transmission protocol

MEHDI ZEROUALI

KTH Information and Communication Technology

Report for Project in Computer Communications
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:105

VoIP Communications over WLANs

Implementation of a new downlink transmission protocol

Mehdi Zerouali, ✉ mehdi.zerouali@insa-lyon.fr

Stockholm, May 25th, 2010

School of Information and Communication Technology (ICT)
Department of Communication Systems (Cos)

Royal Institute of Technology (KTH)

Supervisor: Professor Gerald Q. Maguire Jr.

Abstract

Voice over IP (VoIP) is becoming more and more popular every day. The number of VoIP service providers is continuously increasing along with the number of customers they serve. Moreover, the latest generation of smartphones and mobile devices now incorporate VoIP support. This enables users within a wireless local area network (WLAN) cell to exchange VoIP traffic with other peers.

This new traffic potentially poses a problem for WLANs, as the WLAN access point could be required to handle a large number of small packets of encoded speech. Since the access to the media can only be made by one node at a time, all of the devices must contend to access it. If there are multiple calls between nodes in the WLAN and nodes in the fixed network, then all of these packets must go to and from the access point. Moreover the access point needs to transmit the downlink traffic for all of these nodes. Because the Access Point has the same probability of getting access to the media as any other node, this can lead to high delays, and limits the maximum number of simultaneous calls to a rather small number, despite the increasing data rates that the WLAN interfaces are capable of.

This project implements and evaluates a new solution that consists of aggregating downlink packets at the access point and transmitting a large multicast packet containing a set of voice frames that need to be sent to nodes within the cell. A demultiplexing process at node extracts the appropriate RTP content from the multicast packet and delivers it locally.

Acknowledgment

I am heartily thankful to my supervisor, Professor Gerald Q. Maguire Jr., whose encouragement, guidance, and support from the initial to the final level enabled me to develop an understanding of the subject. His answers to my questions were very helpful.

I would also like to thank my home university in France, INSA Lyon, which offered this amazing opportunity to go and study for a whole year at the Royal Institute of Technology of Stockholm, discovering a new culture and new teaching methods.

Table of Contents

1 Introduction

1.1 Context and Delimitation

This study has been specially designed to address the problem of media contention for IEEE 802.11b/g/n Wireless Local Area Networks (WLANs) in the particular case of encoded data speech transmission traffic - with a focus on the downlink. In this kind of network, when a node inside the cell (an inner node) wants to exchange traffic with a node in the fixed network, (or more generally attached to any node topologically on the far side of the access point) packets must be forwarded through the Access Point (AP). The AP is usually connected to (or includes in itself) a router and can relay data between the wireless devices (computers, smartphones, printers, …) within the cell and the devices connected to the external network (Internet, other LANs, other WLANs, etc).

The new protocol presented and implemented in this work is specifically intended for the case of VoIP communications involving both inner nodes and outer nodes. This implies that the RTP frames must be forwarded through the AP, in order to be delivered to the correct destination. See Figure 1 for a schematic example of the case that this work will focus on.

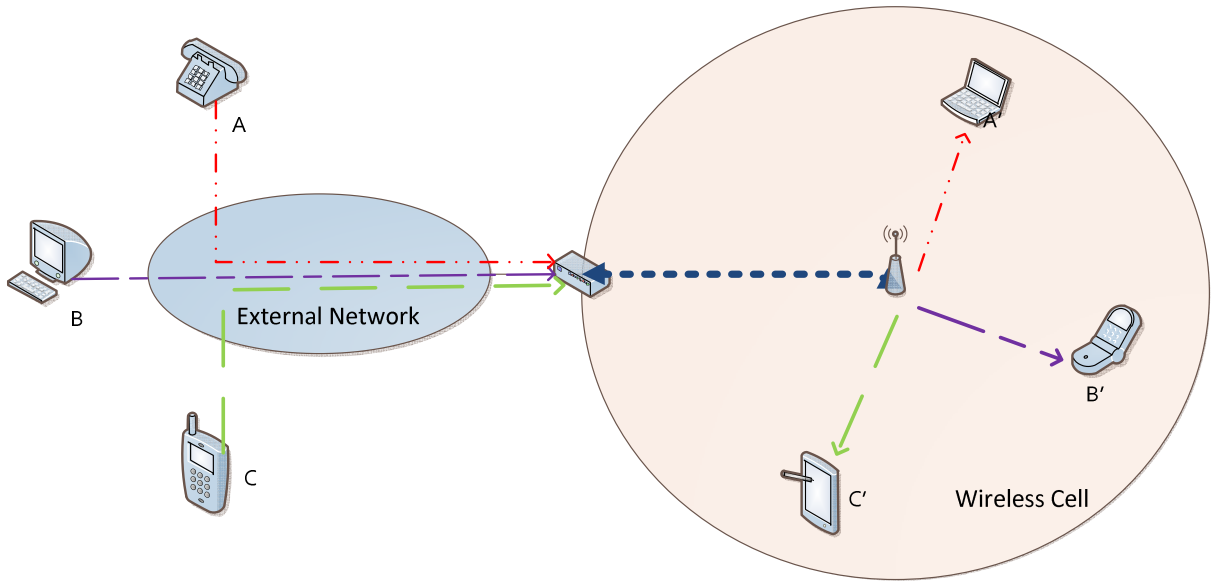


Figure 1: Illustration of the context of this work

1.2 Problems to be addressed by this project

Nowadays, WLANs are capable of providing high data rates, up to 600 Mbit/s for the latest “n” version of the IEEE 802.11 standard [1]. This might lead one to think that a very large number of VoIP users could be accepted within the same WLAN cell, since the amount of data that needs to be exchanged in the case of a standard VoIP session is relatively small (because of the small number of bits that are required to encode speech at sufficient quality for conversations), comparing to the high throughputs that WLANs are capable of.

But as Tony Rybczynski, director of strategic enterprise technology at Nortel Networks Ltd., said during an Internet Telephony Conference & Exposition, “*If you think that Wi-Fi is like a LAN and voice is just another application, you will fail*” [2]. Indeed, VoIP exchanges have many particularities that must be taken into account to provide higher capacity for managing calls and to increase the number simultaneous calls that can be supported.

Assume that many users within the WLAN cell are exchanging VoIP traffic with other users outside the cell. In the case of ITU-T G.711 encoding each one of these inner nodes needs to receive packets from the outside network every 20ms, that means that without our new downlink transmission protocol, the AP will have to contend for the media access each time there is an encoded speech data to be forwarded. Since it has the same probability of accessing the media as the other inner nodes, but has to deliver 50% of all the RTP traffic, the contention that occurs leads to high delays for this traffic.

Our model proposes a solution to reduce this contention. Instead of trying to access the media each time there is a RTP packet to be forwarded by the AP to a node in the cell, we can aggregate all of the RTP packets into one large frame, then multicast this frame to all the inner nodes involved in a VoIP session with the outside network. Therefore, the AP will only have to compete for access to the media every 20ms. If we assume that there is only VoIP traffic to be exchanged, then we see that we have enabled the AP to compete fairly while still getting to deliver the RTP contents that it should. This solution will not interfere with other types of data exchanged with the outer network, since the AP will keep on forwarding non-RTP frames as usual, but aggregating the RTP frames will enable the AP to handle more VoIP users within the cell and will reduce the amount of contention that it generates within the cell (this can even have a beneficial effect for the non-RTP traffic).

1.3 Main goals

The aim of this project is to create a prototype implementation of the new downlink scheme and to evaluate it. Prior work conducted at KTH proposed a system that also included a cooperative behavior for VoIP communication nodes in the WLAN cell (i.e., focusing on avoiding contention on the uplink) [3]. In contrast, in this work we wanted to focus on the downlink protocol, in order to enhance the work done by the former students, by modifying the protocol and proposing a new implementation at a lower level.

We initially targeted our work to use Linux pseudo device drivers and implement a solution as a kernel module. After digging into the world of kernel hacking, and after weeks of research, the contacts that we established with many contributors to the kernel community led us to the conclusion that there was a better way of handling this problem that writing virtual network interfaces as kernel modules. In the following text we will present and explain this method, which relies on ipqueue and Netfilter hooks, in the implementation section of this paper.

The principal requirement of this solution is to respect the time constrained delivery of the RTP packets, which is inherent to maintaining the user's perception of good quality audio. Hence when adding this traffic shaping process at the AP, we must **not** significantly increase the forwarding delays; otherwise we would have reduce the perceived quality of the speech and thus degraded the apparent performance of the AP.

To complete the process of RTP datagram delivery, there must be a demultiplexing mechanism in the inner nodes in order to extract the relevant RTP packet and to deliver it to the application level. In this case as well, high packet delay & delay variation should be avoided.

In the ideal case, this process must be transparent for the outer nodes. Thus any SIP clients or other SIP software must be able to reach inner nodes without needing any modification, as if no downlink process was implemented. This means that we must be able to automatically detect that there is suitable RTP traffic for downlink multiplexing and that the relevant nodes in the cell have the demultiplexing support that is required.

In order to know that the nodes in the cell do have support for downlink demultiplexing there must be a mechanism to register the inner nodes for participating in the multicast process. This process will allow inner nodes that are not (yet) registered to utilize the new downlink protocol to have the opportunity to continue to exchange RTP datagrams with outer nodes, in a transparent way.

Thus inner nodes that do not participate in the downlink multicast will receive by receiving from the AP their RTP frames forwarded individually by the AP in a unicast transmission.

The next section of the report will present the actual protocol, and the structure of the frames exchanged in the proposed new process.

2 Overview of the protocol

2-1 The Model

Assume N inner nodes communicating simultaneously with N outer nodes - with all nodes using G.711 encoding. The volume of the resulting RTP traffic will be 2*N packets every 20ms (assuming that there is no silence suppression). 50% of this traffic is going through the downlink, from the outer nodes to the inner nodes. Therefore, the AP needs to access the media much more often than any other single node. Additionally each of the frames carrying an RTP datagram is quite small, so with the minimum interframe gap and other elements of the WLAN MAC protocol, the available bandwidth is very inefficiently utilized. By aggregating all the downlink RTP frames, we will significantly reduce the AP's need to contend for the media. With this new approach, instead of trying to use the link layer N times every 20ms, it will only transmit bigger frames at a lower rate.

At the AP level, a process waits for incoming packets. If these packets are RTP packets, intended for an inner node registered to use the new protocol, then the packet is queued in a buffer. Actually before being placed in the buffer, the packet's IP and UDP headers are removed (as these are completely predictable). Indeed, there is no need to send the complete IP packet as it arrived at the AP level, but rather we only include the necessary information needed by the demultiplexer to recreate these headers when it extracts the RTP datagrams from this multicast frame. The demultiplexer will recreate and to rebuild the original IP packet and deliver it locally.. So instead of including the 28 bytes of header (20 bytes for the IP header, 8 bytes for the UDP header) in each sub RTP packet of the multiplexed frame, we create a "custom header" that includes the necessary information for the demultiplexer.

This custom header process has been inspired by the Transport Multiplexing Protocol (TMux, RFC 1692) [4], which was created to optimize the frequent transmission of small data packets, for instance in the case of Telnet and Rlogin sessions. It multiplexes several packets into one big frame, and adds a "TMux Mini Header" to each one of them, followed by a "Transport Segment". This model allows the aggregation of different protocols into one frame; for instance, a multiplexed frame could include multiple UDP and TCP sub-packets, thanks to this transport segment.

Since encoded voice data is generally small in terms of the number of bytes needed, RTP packets carrying speech do not generally need to be fragmented and the usual MTU (Maximum transmission Unit) is never reached by a RTP frame, for the voice case. Therefore, all the fields related to fragmentation contained in the IP header need not to be included: hence our custom header does not include the following fields: Identification (16 bits), Flags (3 bits) and Fragment offset (13 bits).

Moreover, other IP header fields can be dropped in the multiplexed frame, because they do not vary from one packet to another and therefore they are completely predictable. For instance, the Version field (4 bits) is always set to 4 for IPv4 data, which is the context of our work. The header length (4 bits) as well will assume that no particular options are used in the transmission of RTP packets.

Because RTP relies on UDP, we will always have the same value (17) in the Protocol field (8 bits). Concerning the TTL (Time to Live) field, the last hop passed by the packet is the AP, so there is no need to include this parameter in our custom header: if the packet succeeded in arriving at the AP, then the datagram should be delivered.

The Differentiated Services field (8 bits) is dedicated to the control and the provision of QoS (Quality of Service) [5]. It allows network devices to handle in a particular way specific types of packets, by prioritizing their processing depending on the value of this field. This is done relative to an administrative policy. Our protocol does not handle this prioritization system, since it is intended to be used with only RTP packets and no differentiation is needed between the RTP frames. Therefore, this value is not included in our custom header.

Finally, the Header checksum field (16 bits) can be computed at the inner node after receiving the RTP frame and putting together all the information included in the custom header to re-generate the IP header.

On the other hand, the IP destination and source addresses (32 bits for each) must be included so that the demultiplexer will be able to extract the appropriate RTP sub-packet from the large multicast frame (by checking the IP destination address) and so that the receiver can create an IP packet with the appropriate source address (using the IP source field).

Concerning the UDP header, it is necessary to keep the ports (source and destination, 16 bits each), since the demultiplexer has to know on which port the application is waiting for the RTP frames. The checksum field (16 bits) can be computed later, hence there is no need to include it in the custom header. Note that the overall multicast frame is protected by a link layer checksum.

The length fields of both the UDP header and IP header carry the same information: the payload size transported, together with the header lengths. In our reduced header, we will only include the length of the payload (RTP header size + RTP payload size), thus the demultiplexer can compute the two length fields from this information.

2-2 Packets Format

The custom header has the following structure:

IP address Outer node source 4 octets	IP address Inner node destination 4 octets	Length RTP length 2 octets	UDP Port Source 2 octets	UDP Port Destination 2 octets

Figure 2: Custom header structure

As we can see, the total length of this header is 14 octets. It is half of the IP and UDP headers' size, thus we are saving 14 bytes for each RTP frame included in the multicast frame. In our source code, this structure is encoded as follows:

```
struct Custom_Header_struct
{
      u_int32_t ip_src;
      u_int32_t ip_dest;
      u_int16_t length;
      u_int16_t udp_port_src;
      u_int16_t udp_port_dest;
};
```

The types u_int32_t and u_int16_t (unsigned 32 bit integer and unsigned 16 bit integer respectively) allow us to specify the exact size that we want the field values to take. It is very important to keep a consistent size and byte order (in our case network byte order), since at the demultiplexer level we will have to fetch the five fields according to their position in the frame.

Figure 3 illustrates the aggregation mechanism

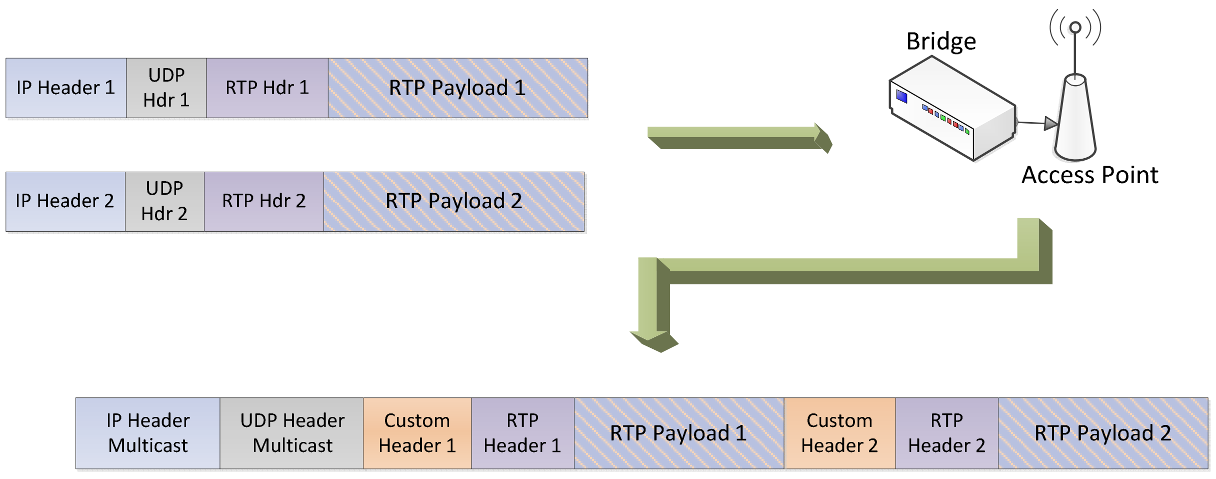


Figure 3: Illustration of the aggregation mechanism

2-3 Registration process

One of our requirements is to let inner nodes that do not want to participate in this particular downlink protocol to exchange RTP traffic as usual with the external network. This means that the AP will not aggregate their incoming RTP packets into the multicast frames, but instead forward them as usual as unicast frames, these transmissions will contend for access to the media as usual.

To allow this transparency, we must register the users that want to participate, in order to enqueue their RTP packets into a queue where they can be processed. To do so, we created two programs: one at the inner node level, and another at the AP. Note that when referring to the "AP level", we actually mean the aggregator machine, in our implementation this is connected between the wired network and the AP. This aggregator looks like by an Ethernet bridge to all of the non-aggregated traffic. Details of this aggregator will be give later, in the implementation section.

At the inner node level, before setting up SIP session for a VoIP session, the users must run a program to indicate that they want to use this new downlink transmission protocol. This program will send to the aggregator (via a TCP socket) a message containing the IP address of the node and the range of ports that will be used for its RTP packets. This information is needed by the aggregator in order to queue the relevant traffic, hence the aggregator must know both the IP address of the participant and the ports that will be used for the RTP packets carrying encoded speech traffic. This is the reason why we created a specific registration message. Still to be addressed is the question of how the software running on this inner node knows the address to send this message to - this will be described in section 3.4.

Rather than introducing a new message we could have used the reception of an IGMP join request to the downlink aggregation multicast group, then add a specific entry into the aggregator with the IP address of the sender of this request. However, since the port range information is not included in this message, and since we do not want to queue inappropriate traffic to the aggregator, a dedicated message was necessary. Although for future work, we should consider if learning the IP address of nodes that want to participate because their demultiplexer joins the multicast group, that we could simply watch for RTP traffic to this node and apply the downlink multiplexing to it.

Consider that the inner node sends to the AP the following message before initiating a SIP session:

```
Register 192.168.0.5:6600:6700
```

After receiving this message, the aggregator will start forwarding to the queue all of the UDP packets destined to 192.168.0.5 and that have their UDP destination port in the range between 6600 and 6700.

If the same inner node wants to unregister, hence start receiving packets in the classical unicast way, it sends an unregister message, i.e., a message:

```
Unregister 192.168.0.5
```

Since we used a TCP connection exchanging these messages, we can also simple unregister the inner node when the TCP socket, and therefore the TCP connection is closed

2-4 Period of multicast frame emission

Naturally, a question comes to mind when designing the different programs implementing our protocol: When should the aggregator issue the aggregated multicast frame? This leads to the question: How long should the period be during which we buffer incoming RTP frames, i.e., when exactly should it stop aggregating and transmit the frame?

Two parameters must be taken into consideration in order to provide a consistent answer to this question. The first one is the maximum frame size for WLAN link. Indeed, if the buffer reaches this limit, the frame should be sent. For an Ethernet wired network the maximum frame size (without fragmentation) is 1500 bytes [6]. For an 802.11 network, the theoretical MTU value is 2336 octets including the full MAC layer header. This number is a maximum theoretical value, and in most cases, the actual maximum frame length is less than 2300 octets. So we will set this value to

2300, leaving enough space to the 802.11 header, of a minimum size of 34 bytes [7]. Note that for our testing phase, which will initially use a wired network replacing the WLAN, we will set the maximum frame size to 1450.

The other parameter that is important for transmitting the multicast frame is the delay between two incoming frames for the same destination at the AP level. Since we want to avoid introducing any significant delay, then knowing that the sources outside the cell will transmit RTP packets every 20 ms, the AP must forward its aggregated frame at least at this same frequency, in order for the inner node to receive RTP packets every 20ms, so that it can send them to the application with the expected interarrival delay distribution. This means that the aggregator will increase the delay of RTP packets by at most roughly 20 ms (on average 10ms). It also means that the interarrival delays experienced by the RTP traffic arriving at the demultiplexer should be integer multiples of 20 ms.

Therefore the aggregator will emit the aggregated frame when either the buffer size reaches its maximum value, or it has been 20 ms since the last aggregated frame was transmitted. After transmitting the multicast message, we clear the buffer and rest our 20 ms timer.

We will see, in the Test and Measurement section, if modifying this 20ms period by a smaller amount enhances the perceived quality of the communication.

3 Implementation

3-1 Choice of the tools used

In order to construct the most efficient system, we needed to operate at the lowest possible level of the computer model. With this assumption, we began working on the conception of a pseudo-device driver, which would act as an aggregator. In the Linux environment, most network interfaces are related to a real physical which is responsible for the packet transport. However, some network interfaces can be exclusively software based, thus it is possible to create virtual network interfaces that uses other network interfaces for actually emitting and transmitting network packets. When it comes to writing device drivers, it is important to distinguish between kernel space and the user space [8].

The user space constitutes the memory reserved for end-users programs. For instance, the UNIX shell and all the Graphical User Interface (GUI) programs execute in user space , but may make calls to the operating system that result in a context switch to process this call into and out of kernel space. Obviously, programs in user space need to communicate with the underlying hardware, for example to output character onto the user's screen. However, these programs generally do not interact with the hardware directly. The relationship between these two spaces and the underling hardware is shown in Figure 4.

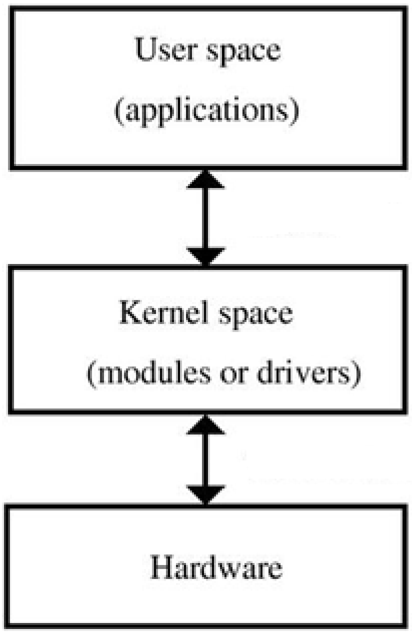


Figure 4: Linux Operating System model

We first started by thinking about the aggregator as a pseudo-device driver, to which we route incoming RTP traffic, do the aggregation, and then emit a multicast frame. After many hours of research to become more familiar with module writing, we found out that there was an easier and possibly more efficient way to perform these tasks. We contacted one of the authors of the reference book O'Reilly Linux Device Drivers [9], who kindly helped us find this alternative method: exploiting the Linux bridging functionality.

3-1-1 Linux Bridge

In our model, we have two sub networks: the wired network (or the outside network) another sub network: the wireless cell. We need to have a mechanism to forward frames or packets between these two independent networks. This could be done by bridging code, routing code, or an application level gateway. The Linux bridge code, fully integrated to most Linux distributions since the 2.4 kernel series, provides a way to connect two network segments (such as two Ethernets) together in a protocol independent way. The bridge operates at level 2 of the OSI model, forwarding is based on the Media Access and Control (MAC) addresses rather than being routed based upon IP addresses.

A bridge actually acts as a switch. Assume that the bridge links two networks via network interfaces eth0 and eth1. The first network (supporting the IP subnet 192.168.1.0/24) is connected to the bridge via the interface eth0, while the second network (supporting the IP subnet 192.168.2.0/24) is connected via interface eth1. When a frame arrives at the bridge, for example an IP packet from the source address 192.168.1.3 would arrive on the eth0 interface, the bridge examines the destination MAC address and if it is not one of the MAC addresses know to be connected to the network attached to interface eth0, then it will forward the frame to the second interface eth1. Note that unlike a router, the bridge will not use the IP addresses to decide where to forward the frame, but rather it only uses the MAC addresses instead [10].

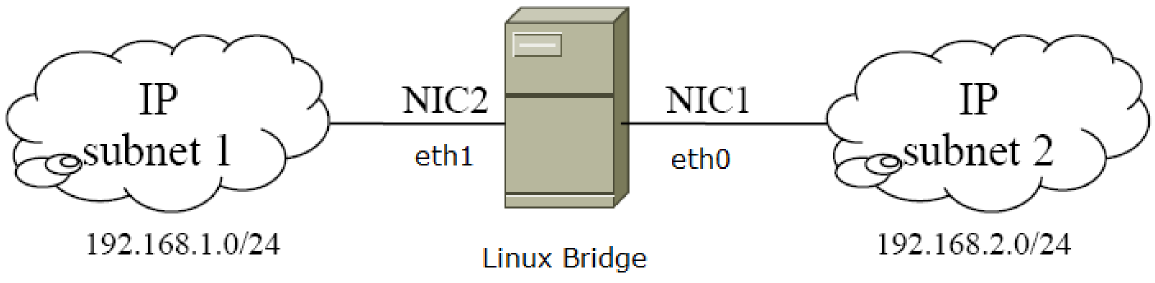


Figure 5: Linux Bridge illustration

In order to set up the bridge between two interfaces, the commands shown in Figure 6 must be typed into a shell, with the root privileges. The first of these commands "brctl" invokes the bridge control program to create a new bridge named "br0". The seconds and third commands add the interfaces eth0 and eth1 to this bridge. The fourth and fifth commands reset the network interface to remove the IP addresses that were associated with these two interfaces. The result is that the two network segments are now connected via bridge "br0" and routing is no longer done between these two IP subnets.

```
## Create the Bridge
brctl add br0

## Add physical interfaces to the bridge
brctl addif br0 eth0
brctl addif br0 eth1

## Reset IP interface
ifconfig eth0 0.0.0.0 up
ifconfig eth1 0.0.0.0 up
```

Figure 6: Bridge configuration

This bridge will now forward packets from the outer nodes to the inner nodes (those connected to the IP subnet 192.168.1.0/24).

This mechanism is actually not fully used in our implementation, since the traffic rules rely on the IP level of the OSI model.

The next subsection will present the tool used to actually aggregate the RTP traffic coming to the downlink.

3-1-2 Netfilter and iptables

Netfilter is a framework that provides hooks to intercept and process network packets. Merged into Linux kernel version 2.3 in March 2000, it is a powerful tool used for packet filtering. This framework is the basis for a number of other network tools, such as the user space tool “iptables” that we use in this project.

Iptables replaced Ipchains within the Linux kernel versions 2.4 and beyond. Iptables is used for IPv4 frames, whereas other tools exist for other types of frames: ip6tables applies to IPv6, arptables to ARP, and ebtables for Ethernet layer 2 frames.

Iptables enables users (with root privileges) to specify specific traffic rules, based on the header of the packets transmitted or received. For example, iptables can perform firewalling tasks, by examining if the IP packets meet the rules of a particular policy (i.e., the policy is implemented by specifying a set of rules which iptables then uses to process the traffic). These rules are organized as chains, and every network packet arriving at or leaving from the computer traverses at least one chain.

There are five predefined chains, although one can create as many chains as desired.

“INPUT” chain	Used for traffic entering our system, the packet will be delivered to an IP address present on our local address.
“OUTPUT” chain	Used for packet emitted by our station.
“PREROUTING” chain	Before any routing rule is applied, the packets will enter this chain.
“FORWARD” chain	Used for packets that have been routed and that were not intended to an IP address of our station.
“POSTROUTING” chain	After routing rules were applied, the packets will enter this chain

For the purpose of this work, we will be using the INPUT chain, to catch traffic entering our system, since the Forward chain is exclusively utilized for routing processes.

After entering a chain, the packet is examined according to the rules of the chain. Each rule contains a target, or verdict, stating the action that is to be performed for packets matching this rule. There are four targets:

“ACCEPT” target	Allow the packet to continue its path, as if no firewalling action happened. The packet is not modified
“DROP” target	Blocks the packet, it will not be delivered and the station will act as if it has never been received or created
“REJECT” target	Also blocks the packet and deny access to traffic, but sends back the appropriate ICMP error message to the originator of the packet
“QUEUE” target	Transfer and queue packet to user-land programs and applications. It allows the processing of the packet to be done in the user space

For our project, it is the last target, “QUEUE” target, which we will be using. We will add the appropriate rule to extract the RTP frames and send them to user space where aggregation will take place. To process the extracted frames, we will use the libnetfilter_queue user space library as it provides an API for handling the packets that have been enqueued by the kernel packet filter [11, 12].

3-1-3 Libnetfilter_queue

The Libnetfilter_queue API has deprecated the former libipq development library.

This library requires other libraries to be present on the system , specifically the following:

- The iptables development package. This can be installed (on systems that have apt-get installed) with the following command :
  ```
  sudo apt-get install iptables-dev
  ```

- The low level library for Netfilter related kernel space/user space communication. The necessary files for the installation can be downloaded from: `http://ftp.netfilter.org/pub/libnfnetlink/`

After successfully adding these two libraries, the libnetfilter_queue API can be installe. The necessary files can be found at: http://ftp.netfilter.org/pub/libnetfilter_queue/ . Note that it is important to follow the above order when installing the libraries (due to each of the libraries dependencies). We assumed that the kernel to be used is at least at version 2.4, so that the kernel already incorporates Netfilter and ipqueue [13].

3-1-4 IP aliases

For the purpose of our work, we need to either have a number of computers or simulate multiple nodes within one system. Each of these physical or virtual nodes will be identified by different IP addresses. One simple way of having multiple IP addresses bound to a single network interface is to use aliases. Thanks to this mechanism, one physical node can have several logical IP addresses. For instance, the following command shows how to create the alias eth0:0 that will be associated with the physical interface eth0:

```
ifconfig eth0:0 192.168.0.10 netmask 255.255.255.0 up
```

3-2 The Aggregator:

Once the packets are enqueued into the user space process, the aggregator program processes them in the way we want, generating the multiplexed frame. The same program sends the resulting frame onto the multicast address, to which the inner nodes are listening. Two independent threads are needed to perform these actions. One of these threads will be used for receiving and processing intercepted frames and the other will emit the resulting multicast frame. We will refer to these threads as the traffic shaper and the emitter (respectively)

3-2-1 Traffic shaper

The `traffic_shaper extracts` packets from the queue, one by one, and calls `build_multiplex().` This function will extract from the packet the RTP payload, and generate from the IP and UDP headers the custom header specific to each sub packet.

The function `ipq_read`, provided by the libnetfilter_queue API, reads one packet at a time from a queue and copies it to a buffer. The `ipq_get_packet` function, called with this buffer as the only parameter, returns a packet whose structure is as defined as shown in figure 7:

```
typedef struct ipq_packet_msg
{
    unsigned long packet_id;         /* ID of queued packet */
    unsigned long mark;              /* Netfilter mark value */
    long timestamp_sec;              /* Packet arrival time (seconds) */
    long timestamp_usec;             /* Packet arrival time (+useconds) */
    unsigned int hook;               /* Netfilter hook we rode in on */
    char indev_name[IFNAMSIZ];       /* Name of incoming interface */
    char outdev_name[IFNAMSIZ];      /* Name of outgoing interface */
    unsigned short hw_protocol;      /* Hardware protocol (network order) */
    unsigned short hw_type;          /* Hardware type */
    unsigned char hw_addrlen;        /* Hardware address length */
    unsigned char hw_addr[8];        /* Hardware address */
    size_t data_len;                 /* Length of packet data */
    unsigned char payload[0];        /* Optional packet data */

} ipq_packet_msg_t;
```

Figure 7: IP packet message structure

The actual packet is accessible with the following syntax: `msg->payload`. The other attributes of the structure are not used in this project. Although in future work these other attributes might be used.

After fetching the message, we call the `build_multiplex()` function with the following parameters:

- The message, `msg`, obtained with the functions above.
- Another buffer, `multiplex`, which is the multiplexed frame to be updated by this function.
- An offset parameter, to know the size of the data already contained in the `multiplex` buffer. This offset will be modified by the function after adding the new RTP sub frame.

Since the size of the `multiplex` frame is always changing, and increasing with each call to this function, we use a `realloc` call that allows us to increase the size of the buffer. When an RTP packet is processed, we simply drop the incoming frame, so that it will not be delivered by AP.

Note the realloc call is not necessary. Since we know the upper bound on the size of the multiplexed frame, we can simply create a buffer of this maximum size initially.

3-2-2 Sender

Before beginning an infinite loop, the sender thread creates a UDP socket for multicast. The destination IP multicast address chosen is 225.0.0.10, and the port used is 8888 (these values can be different, as long as they match those used by the demultiplexer). The addresses in the range 224.0.0.0 – 224.0.0.255 are reserved for routing and maintenance protocol.

Within an infinite loop, the `sender` thread is responsible for sending the multiplexed frame to the destination IP multicast address. If the offset variable, which is common to the two threads (i.e., the offset in the `multiplex` buffer) reaches the maximum value of 1440, then the thread sends the packet directly. Otherwise, the thread will wait 20ms and then send the multicast frame. In both cases, before sending the frame, it will compute the number of sub packets contained in the `multiplex` buffer, in order to make the demultiplexing process at the inner node level easier and faster. This value is increased each time the `build_multiplex` function is called in the `traffic_shaper` thread.

After sending the packet, we must reset all the shared variables. The thread will free the `multiplex buffer`, reinitialize the `offset` and the number of packet values, and allocate space for the buffer `multiplex`, by using a call to the `malloc` function.

3-2-3 Mutual Exclusion

As we saw in the previous section, the two threads share three variables: the offset, the number of packets, and of course the multiplexed frame. It is extremely important to provide a mechanism to protect these variables from simultaneous and concurrent access. For instance, it must be forbidden for the sender thread to emit the multiplexed frame while the traffic shaper thread is still updating it.

The pthread library provides a way to prohibit this concurrent access that could lead to inconsistent results. By using mutexes in an appropriate way, we can lock some parts of the code and therefore prevent data inconsistencies due to race conditions. Mutexes are declared and initialized with the following commands:

```
//Declaration of the mutex
pthread_mutex_t mut;

//Initialization of the mutex
pthread_mutex_init(&mut, NULL);
```

Each time before a global resource is accessed, the thread must try to lock the mutex, by calling the function: `pthread_mutex_lock(&mut)`. The thread is therefore blocked, waiting for the mutex to be released by the other thread. When the mutex is free, the thread can enter the "critical section" and then modify the shared variables. When the process is finished, the function pthread_mutex_unlock(&mut) is called in order to unlock the mutex, allowing the other thread to get the access to it and to enter its "critical section".

Note in contrast to semaphores, mutex can only be used for threads belonging to the same process [14].

In order to avoid the concurrent access problem, we also could have implemented the aggregator as one single thread with an infinite loop and an event timer. The arrival of an RTP packet would play the role of the trigger, and the program will check for how long the oldest packet contained in the buffer has been queued. If this time reaches 20ms, the multicast frame is sent. It is also sent if the limit size of the buffer (the MTU) is reached.

3-3 The Demultiplexer

Upon receiving the multiplexed frames, the inner nodes need to extract the appropriate content (i.e., the sub-frames that are intended for their IP address). To do so, here again, two threads are needed: a receiver and an extractor.

.

3-3-1 Receiver

The receiver thread simply listens to the multicast socket to get the multiplexed frames emitted by the aggregator. An IGMP join request structure is created and an ADD Membership message is sent, before starting an infinite loop. Sending this IGMP join message is done by using the socket API, particularly the `setsockopt` () method that enables us to specify particular settings for a socket. Another socket is created during this phase, a Raw socket, used for sending the actual RTP packets extracted from the multiplexed frame. Within the infinite loop, after each packet is received from the multiplex frame, the frame content is copied to another buffer, shared with the second thread, in which extraction of the RTP frame occurs. Note that for this process, we also use mutexes to ensure that no concurrent access will occur and to prevent inconsistent results.

3-3-2 Extractor

The extractor thread must run after each multicast packet to port 8888 is received. In addition to the mutexes, we are using a Boolean as a flag to know if a new packet has arrived and if additional processing needs to be done. The thread must not run twice on the same multiplexed frame; as this would result into duplicated RTP packets. It would have no effect on the conversation quality since the SIP user agent would put the contents in the same place for playout - hence the content will only be played once (this can be done because of the sequence number and timestamp in the RTP packet header).

When the extractor thread enters its “critical section”, by locking the mutex, it begins by extracting the first two bytes of the packet, which represents the number of sub RTP packet contained in the large packet. Thanks to the custom header of each of these sub packets, the extractor can determine their length by checking the length field. Therefore, the extractor knows where an RTP frames ends and where the next one starts.

The thread invokes the function `addresse_ip_check()` for each one of the sub packets. As its name indicates, this function looks at the IP destination field of the customer header, and returns 1 if this destination IP address matches the IP address used by the process.

If the value one is returned, this means that the particular sub packet treated is intended for this particular inner node. Hence we need to re-generate the original IP packet, i.e., we will produce an IP packet that is identical to the one emitted by the outer node. To perform this action, the extractor thread calls the `raw_packet` function, with the sub_packet and the descriptor of the Raw socket as parameters.

In this function, the IP and UDP headers will be recreated exactly the same as the original, thanks to the custom header. Raw sockets, supported by the Berkley socket API, provide a way to bypass the encapsulation processes done by the network stack of the operating system. This allows the program to forge a packet with a different IP source address than the running program. Note that Raw sockets can be used in a malicious way, to impersonate other computers of a network by spoofing their IP addresses in order to perform intrusion attacks, such as for a SYN DoS attack [15].

To perform this packet forging successfully, both the IP header and the UDP checksums must be recalculated before sending the packet to the wired network. Moreover, we must tell the kernel not to add headers to these packets, since the thread already created these headers

.

```
{

    int one = 1;
    const int *val = &one;
    setsockopt (s, IPPROTO_IP, IP_HDRINCL, val, sizeof (one)

}
```

The demultiplexer program must be run with the root privileges, since this is a requirement for performing Raw socket operations.

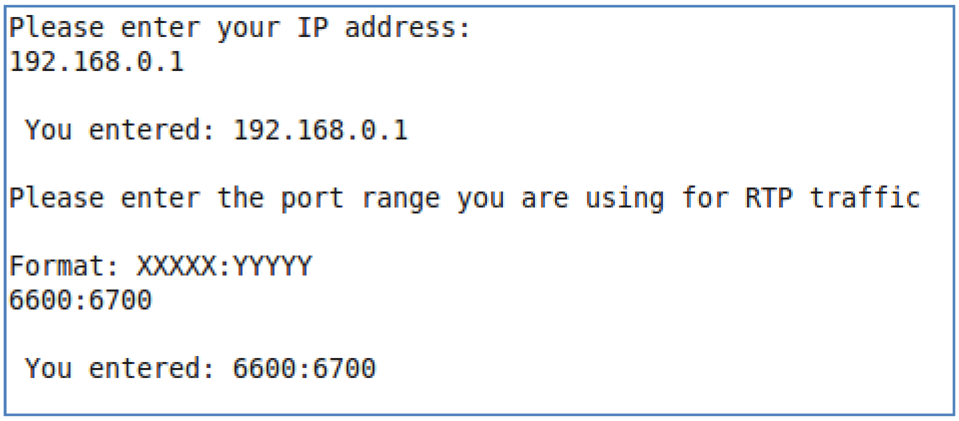
3-4 Registration / Unregistration Mechanism:

In order to forward the UDP packets destined to IP address X with a destination port between Y and Z, the following command must be entered at the aggregator:

```
Iptables –A INPUT –p UDP –d X/32 --dport Y:Z –j QUEUE
```

The above code cause iptables to add a new entry to the INPUT chain that will cause packets destined to exactly the IP address X (indicate by saying that all 32 bits of the IP address must match X) and that the destination port is within the range Y to Z. If a packet matches these requirements, it is enqueued for further processing.

When the demultiplexer is launched, the program asks the user to enter the IP address to which it will listen for packets, and the range of ports that will be used for exchanging encoded speech data with the wired network



```
Please enter your IP address:
192.168.0.1

 You entered: 192.168.0.1

Please enter the port range you are using for RTP traffic

Format: XXXXX:YYYYY
6600:6700

 You entered: 6600:6700
```

Figure 8: Demultiplexer screenshot

The program creates a socket for communication with the aggregator and then transmits this information. After receiving this message, the aggregator calls a function that takes the message as a parameter, and generates the appropriate iptables command. Using a call to the "system()" function, the command is executed which adds the iptables rule to the INPUT chain.

To enable unregistration, we implemented a means to dynamically stop the forwarding of the UDP packets to the queue. When the user wants to stop using our downlink protocol, he or she simply stops the demultiplexing program by pressing the combination of key Ctrl+C. By doing so, an interruption signal or SIG INT is sent to the process [16]. For most C programs without a specific interrupt handler, this will simply stop the programs execution. If this were to happen, the queuing process at the aggregator would continue, hence the RTP frames intended for the node would still be aggregated into the large multiplexed frame and sent to the multicast group. Therefore, because the demultiplexing program has terminated, a VoIP application waiting for these packets will not be able to receive them.

In our program, we implemented an interrupt handler function that is called when a SIG INT is received. By calling signal(SIGINT, unregistration) we catch the signal and call our own unregistration function, which sends a message to the aggregator through the socket. When the aggregator receives this message, the iptables_rule function will delete the rule matching this particular inner node (based upon its particular IP address) and the specified port range. Therefore, as soon as the Ctrl+C is pressed by the user, the system will switch to unicast transmission of RTP frames to this destination IP address.

There is a potential security problem here someone could generate packets to the TCP port indicating that someone else's use of the aggregator should be terminated. Note that the only effect that this will have is to prevent RTP packet aggregation and all of the RTP packets will be delivered via the usual unicast mechanism. Rather than being a pure denial of service attack this would simply be a degradation of service attack, by preventing aggregation.

To solve this problem, we can encrypt the registration and unregistration message with a secret key shared by the aggregator and a unique inner node. To do so, we must have a Key exchange protocol, such as a Diffie–Hellman key exchange.

Note that most users do not know their IP address, they just connect the network and use their machine to accessing the data they want. Hence, they would not be able to specify the IP address when running the demultiplexer program. To solve this problem, we could have implemented a sniffer that sends the registration message when it detects RTP streams exchanged by the machine. In this way, the registration will be transparent for the user.

Another problem is how do the inner nodes know the address of the aggregator machine? It can be configured manually in the software, which means that it should be modified each time the aggregator changes its IP address. Another solution is actually to send the registration and unregistration messages in a broadcast packet, which can be interpreted only by the aggregator. Note this solution brings other potential security issues.

4 Tests and Performance Measurement

In order to validate our model, we will perform a number of tests to show that all the different components of our system work correctly and produce the desired results. We will also analyze parameters such as the delay, the number of packets lost and the amount of jitter that is induced. In general, when a sequence of RTP packets is sent from machine 1 to machine 2, these packets will take a different time to reach their destination. Assuming that a source emits RTP frames every 20 ms, then the receiving party will probably receive these packets with different time intervals. This variance in the periodic arrivals is referred to as jitter. This jitter can be cause by queuing in routers, contention for links, etc.

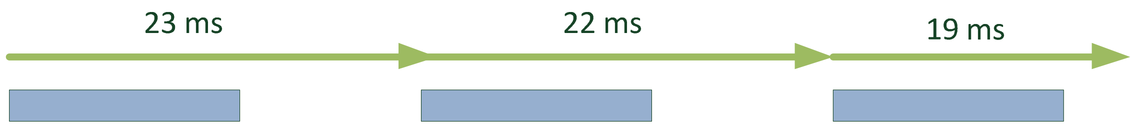


Figure 9: Illustration of the difference in arriving time between packets

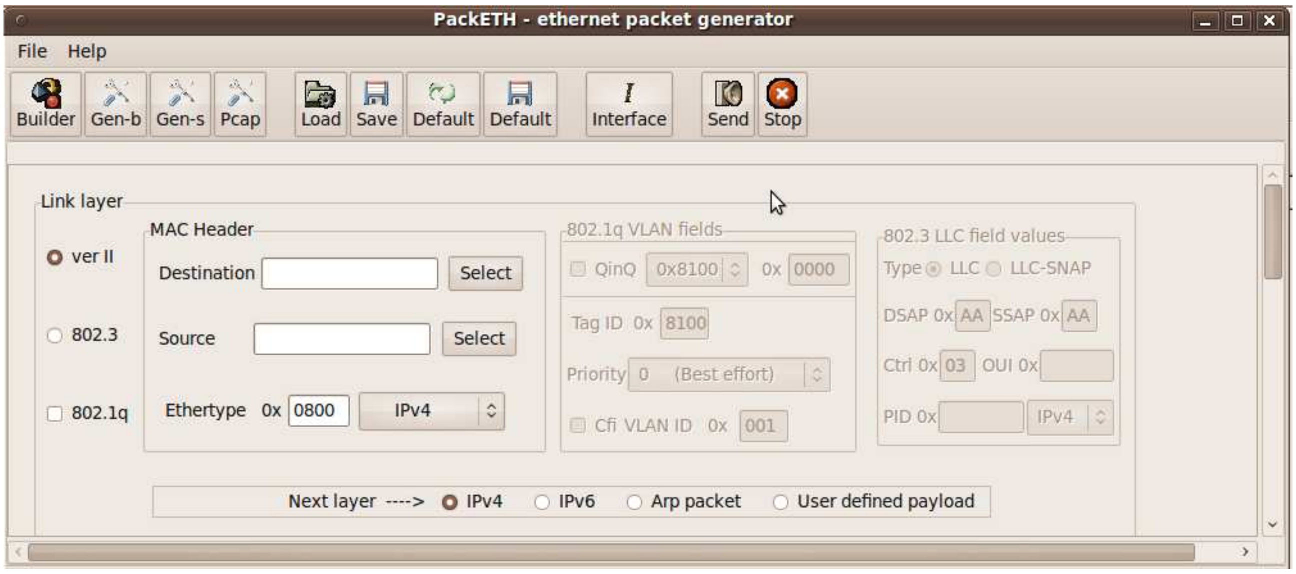
4-1 Tools used

To perform the tests, we used two different programs to emit RTP packets from an outer node. The first one is a C program that we wrote, which sends RTP packets to different IP addresses, using RAW sockets to forge the UDP and IP header fields. Used within an infinite loop, in conjunction with a sleep(20000) call, it sends packets every 20 ms packets to the IP addresses of inner nodes. Each of these packets has a different payload. We have implemented a function that randomizes the payload before sending the packet, so that no two packets will ever be identical. The length of the payload is also a random parameter, that is generated each time a new packet is to be created. This represents a more demanding stream of packets that would be expected from encoded audio, but might be more representative of encoded video RTP packets.

Concerning the RTP header, except for some fields (version = 2, extension and padding = 0, and payload type = RTP_PAYLOADTYPE_G711), that are constants for all the sequences; we increment the sequence number and increase the timestamp by 20ms each time a packet is sent.

To deliver a RTP stream which can be interpreted and analyzed to extract useful parameters, we used a Linux GUI packet generator tool, specifically the packETH software [17]. This tool allows users to create and send many different kind of frames: Ipv4 (UDP, TCP, IGMP, ICMP...), ARP, and custom network layer frame.

The user interface to this program is shown in Figure 8.



Link layer: [✓ ver II] 802.3 802.1q

MAC Header
Destination: Select
Source: Select
Ethertype 0x 0800 IPv4

802.1q VLAN fields
QinQ 0x8100 0x 0000
Tag ID 0x 8100
Priority 0 (Best effort)
Cfi VLAN ID 0x 001

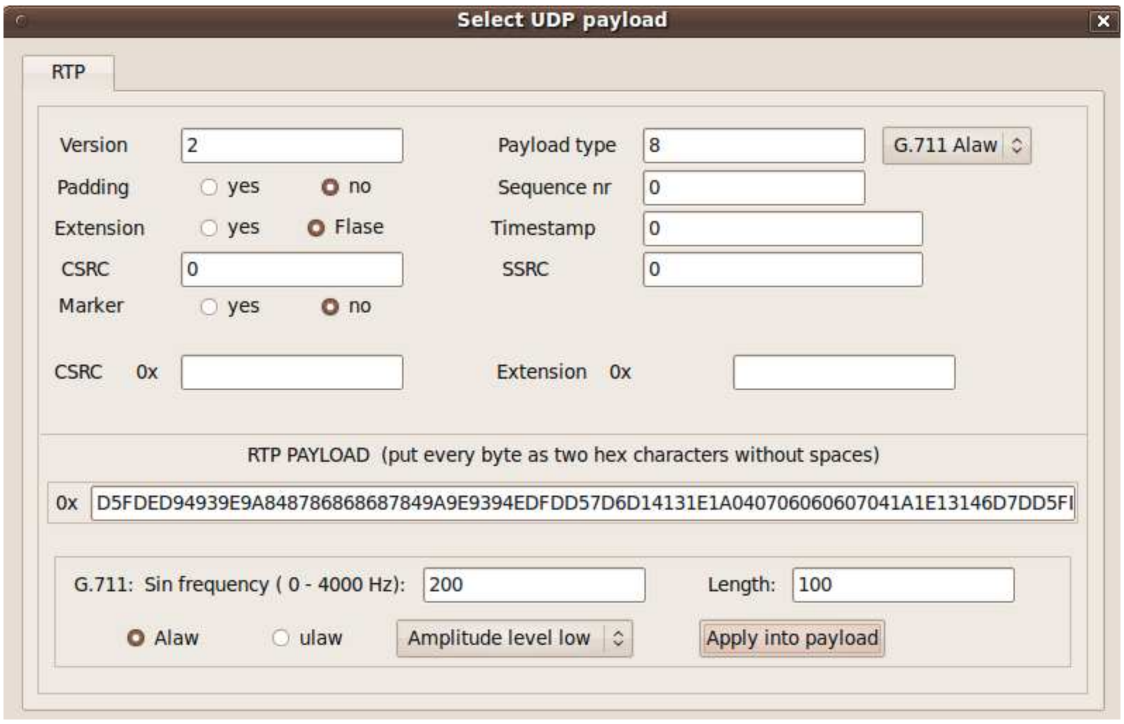
802.3 LLC field values
Type [✓ LLC] LLC-SNAP
DSAP 0x AA SSAP 0x AA
Ctrl 0x 03 OUI 0x
PID 0x IPv4

Next layer ----> [✓ IPv4] IPv6 Arp packet User defined payload

Figure 10: Screenshot from the software packETH: Main interface

Using the tool we can specify the destination and source IP addresses or the source and destination MAC addresses for layer 2 headers. Moreover, the “Interface” button permits user to select which interface the packet is sent through.

The features of this software that interests us are the possibility to send a continuous RTP data flow. For example, we can encode a sinusoidal wave of any frequency between zero and 4000 Hz with the G 711 CODEC. The GUI for generating RTP flows is shown in Figure 11.


Select UDP payload

RTP

Version: 2
Padding: yes [✓ no]
Extension: yes [✓ Flase]
CSRC: 0
Marker: yes [✓ no]
CSRC 0x:

Payload type: 8 — G.711 Alaw
Sequence nr: 0
Timestamp: 0
SSRC: 0
Extension 0x:

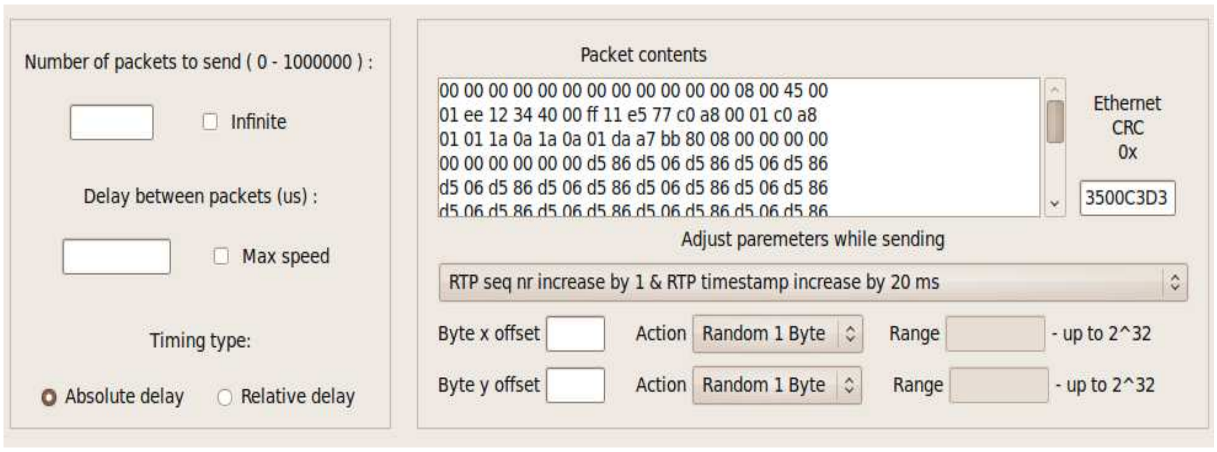
RTP PAYLOAD (put every byte as two hex characters without spaces)

0x D5FDED94939E9A848786868687849A9E9394EDFDD57D6D14131E1A040706060607041A1E13146D7DD5FI

G.711: Sin frequency (0 - 4000 Hz): 200
Length: 100
[✓ Alaw] ulaw
Amplitude level low

Figure 11: Screenshot from the software packETH: RTP Interface

The other essential feature of this tool that is necessary for our testing is the ability to modify of some fields during the sending process. For example, the software has an option to transmit sequence of packets leaving the choice of the delay between the frames to the user. One can specify changes to be made on some parameters of the packets while sending. One of these options allows us to increment the sequence number of the RTP header, and to increase by 20ms the timestamp field. (See Figure 12)


Number of packets to send (0 - 1000000) :
Infinite

Delay between packets (us) :
Max speed

Timing type:
[✓ Absolute delay] Relative delay

Packet contents

```
00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00
01 ee 12 34 40 00 ff 11 e5 77 c0 a8 00 01 c0 a8
01 01 1a 0a 1a 0a 01 da a7 bb 80 08 00 00 00 00
00 00 00 00 00 00 d5 86 d5 06 d5 86 d5 06 d5 86
d5 06 d5 86 d5 06 d5 86 d5 06 d5 86 d5 06 d5 86
d5 06 d5 86 d5 06 d5 86 d5 06 d5 86 d5 06 d5 86
```

Ethernet CRC 0x: 3500C3D3

Adjust paremeters while sending: RTP seq nr increase by 1 & RTP timestamp increase by 20 ms

Byte x offset: Action: Random 1 Byte Range: - up to 2^32
Byte y offset: Action: Random 1 Byte Range: - up to 2^32

Figure 12: Screenshot from the software packETH: modification of parameters

4-2 Different Test Scenarios

4.2.1. Testing the correct reception of three packets: payload comparison.

In order to validate the correct reception of the frames at the inner node level, we performed the following test:

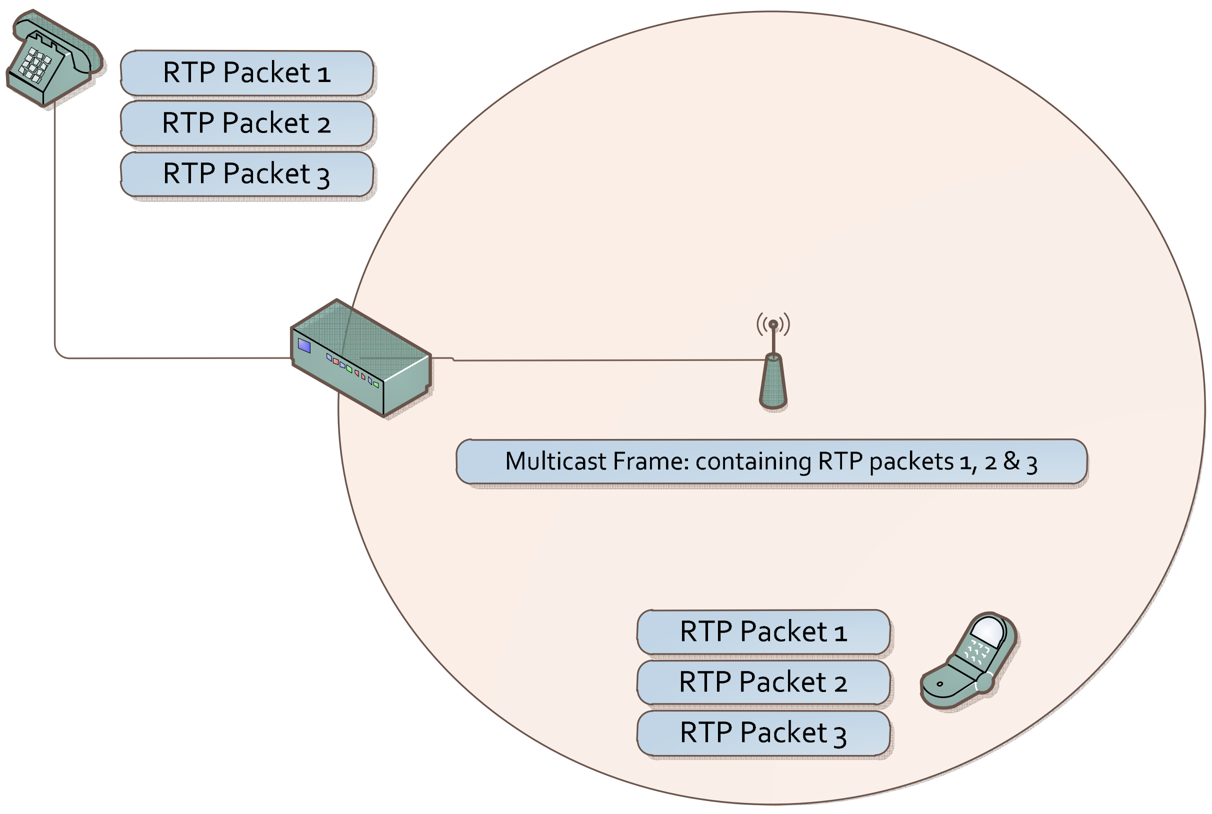
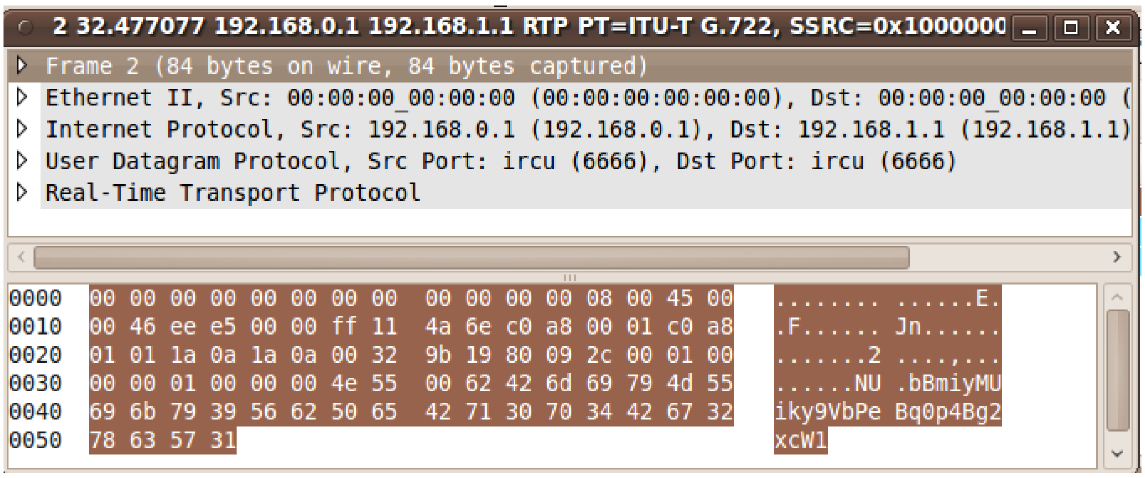


Figure 13: Illustration of the test 1: three RTP packets exchanged

Using our sender program, we sent three RTP frames to an inner node. The aggregator multiplexed them into one large multicast frame and transmitted this frame to the multicast group, to which the demultiplexer program of the inner node is listening. The demultiplexer processes the frame and re-creates the initial three frames and sends them via the loopback interface. Comparing the payload of the initial frames emitted by the sender program (shown in figure 14) with the final three frames extracted from the multiplexed one show that the aggregator and demultiplexer work as expected The packet that is received by one of the inner nodes is shown in Figure 15.

It is clear that this matches the fields in the packet shown in Figure 14 except for the destination IP address which in the locally generated packet which shows the loop-back interface's IP address rather than the node's network interface's IP address. While this does enable the packet to be delivered to the waiting RTP listener, it might not be acceptable from a security point of view - because even though the checksums have been re-computed with the IP address that is used as the destination - the application might not be accept the change in the destination IP address.

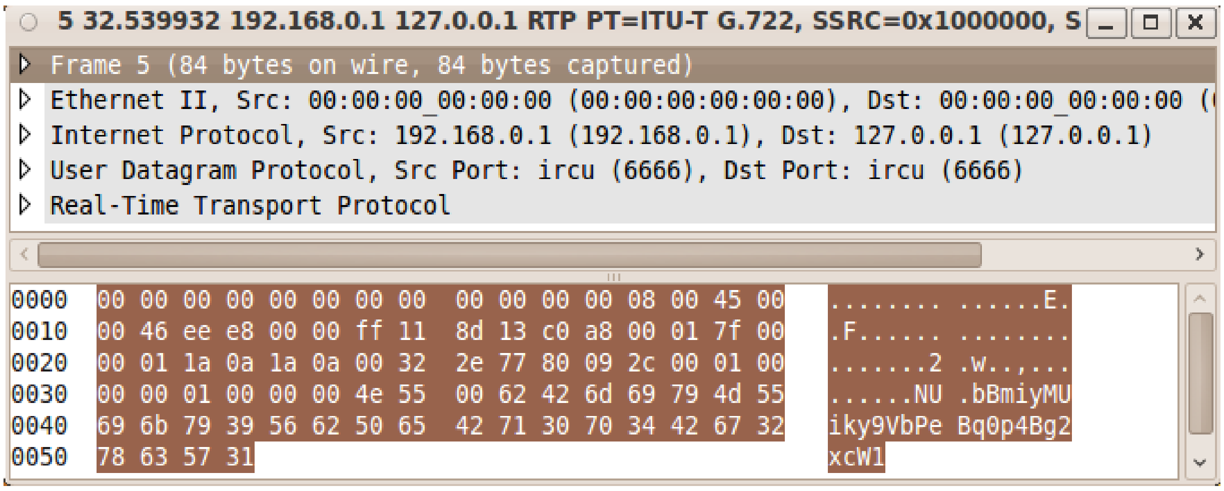
We capture the packets exchanged using Wireshark. Wireshark can display the payload and can decode the RTP packets. This makes it easier for us to compare the original RTP datagrams with the final RTP datagrams.



```
2 32.477077 192.168.0.1 192.168.1.1 RTP PT=ITU-T G.722, SSRC=0x1000000
▷ Frame 2 (84 bytes on wire, 84 bytes captured)
▷ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (
▷ Internet Protocol, Src: 192.168.0.1 (192.168.0.1), Dst: 192.168.1.1 (192.168.1.1)
▷ User Datagram Protocol, Src Port: ircu (6666), Dst Port: ircu (6666)
▷ Real-Time Transport Protocol

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 46 ee e5 00 00 ff 11  4a 6e c0 a8 00 01 c0 a8   .F...... Jn......
0020  01 01 1a 0a 1a 0a 00 32  9b 19 80 09 2c 00 01 00   .......2 ....,...
0030  00 00 01 00 00 00 4e 55  00 62 42 6d 69 79 4d 55   ......NU .bBmiyMU
0040  69 6b 79 39 56 62 50 65  42 71 30 70 34 42 67 32   iky9VbPe Bq0p4Bg2
0050  78 63 57 31                                        xcW1
```

Figure 14: Packet emitted by the outer node using the sender program



```
5 32.539932 192.168.0.1 127.0.0.1 RTP PT=ITU-T G.722, SSRC=0x1000000, S
▷ Frame 5 (84 bytes on wire, 84 bytes captured)
▷ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (
▷ Internet Protocol, Src: 192.168.0.1 (192.168.0.1), Dst: 127.0.0.1 (127.0.0.1)
▷ User Datagram Protocol, Src Port: ircu (6666), Dst Port: ircu (6666)
▷ Real-Time Transport Protocol

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 46 ee e8 00 00 ff 11  8d 13 c0 a8 00 01 7f 00   .F...... ........
0020  00 01 1a 0a 1a 0a 00 32  2e 77 80 09 2c 00 01 00   .......2 .w..,...
0030  00 00 01 00 00 00 4e 55  00 62 42 6d 69 79 4d 55   ......NU .bBmiyMU
0040  69 6b 79 39 56 62 50 65  42 71 30 70 34 42 67 32   iky9VbPe Bq0p4Bg2
0050  78 63 57 31                                        xcW1
```

Figure 15: Packet recreated by the inner node demultiplexer after receiving the multicast frame

As we can see, the payloads are quite similar, which means that the demultiplexer largely succeeded in regenerating the packet that was issued by the outer node. The result was the same for the two other frames. Since no time delay between the packets was specified in the sender for this test, the three packets were sent sequentially resulting in all of them being multiplexed into a single multiplexed packet emitted by the aggregator.

4.2.2. Transmission of a flow of RTP packets from a single outer node to a single inner node:

The main goal of this test is to verify the consistency of the delays and jitters as experienced at the receiving side, by comparing them to the delay and jitter at the emission side. In this test, there is only one outer node sending encoded voice data multiplexed by the aggregator. The results of this test will allow us to know if the presence of our multiplexing system introduces traffic latency, or if the RTP packets will not be affected by the new protocol.

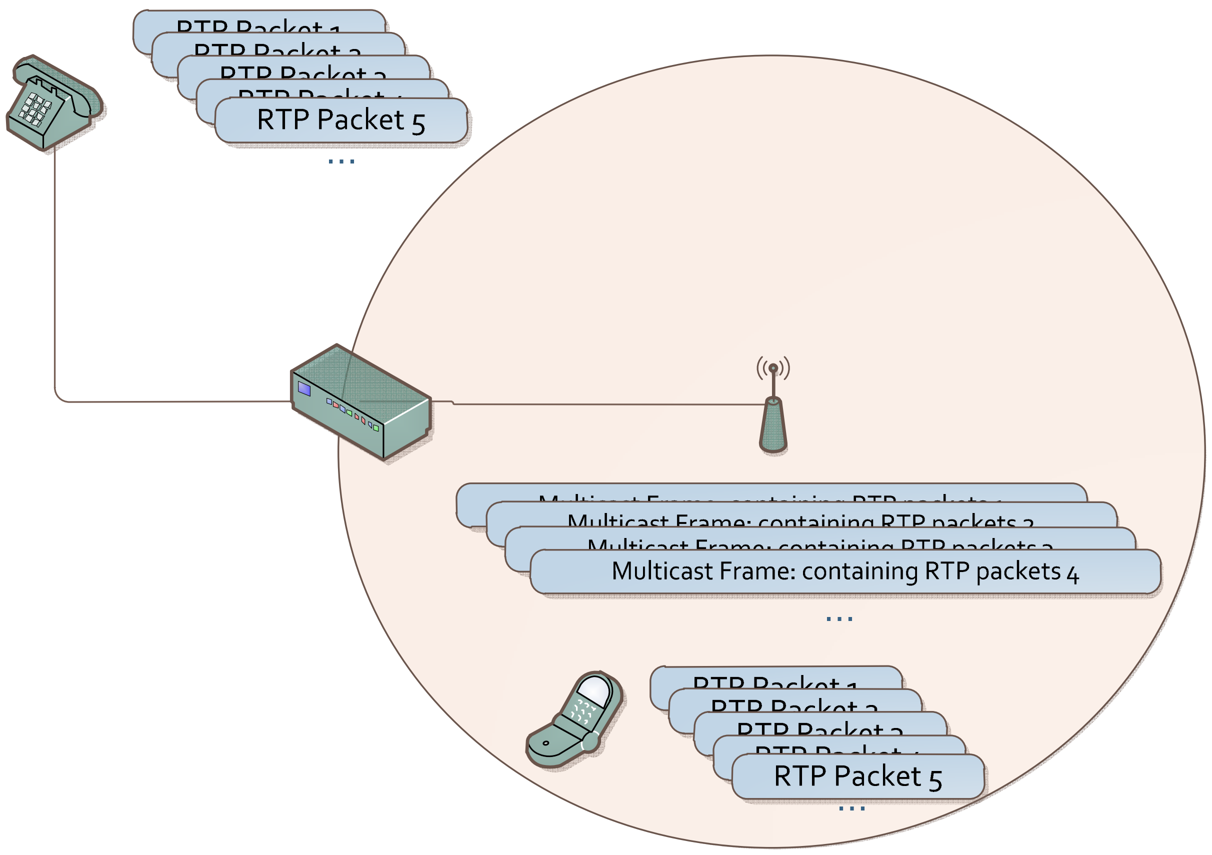


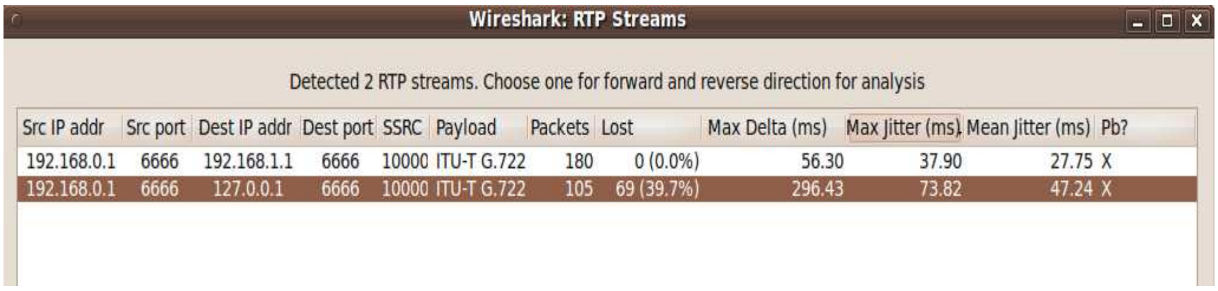
Figure 14: Illustration of the test 2: one single RTP stream

Every 20ms, the sender transmits a packet to the inner node. At the bridge level, the aggregator processes the incoming frames and puts them into a multicast packet and sends them to the multicast group every 20ms. In this test case, the maximum size of 1460 octets is never reached, since only one RTP stream is simulated.

We recorded the traffic with Wireshark on both the network where the traffic generator was running and on the inner node., and thanks to the stream analysis function of this software we can display the number of packets lost, the mean and maximum jitter, and the maximum delay.

Note that we purposely put short pauses in both the demultiplexer and the aggregator threads. This was because we found that in order to ensure the correct synchronization between the threads, a mutex is not always sufficient. We added a call to the usleep function within the infinite loop of both threads. We modified the duration of these pauses to see how they affected the performance of the protocol.

a) Without any pauses in the threads (i.e., usleep is not called at the end of the threads):



Detected 2 RTP streams. Choose one for forward and reverse direction for analysis

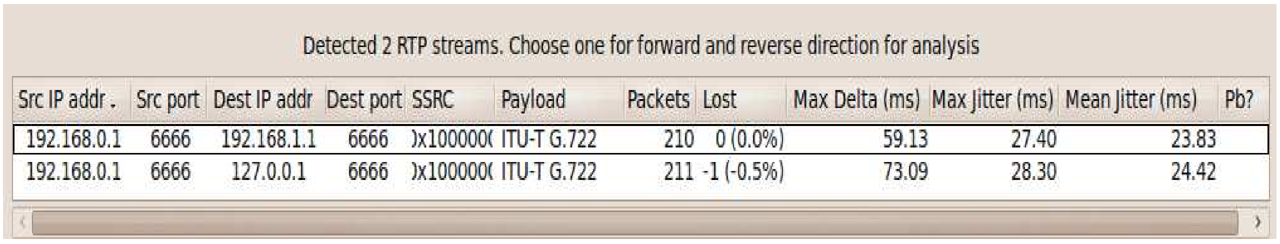
Src IP addr	Src port	Dest IP addr	Dest port	SSRC	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)	Pb?
192.168.0.1	6666	192.168.1.1	6666	10000	ITU-T G.722	180	0 (0.0%)	56.30	37.90	27.75	X
192.168.0.1	6666	127.0.0.1	6666	10000	ITU-T G.722	105	69 (39.7%)	296.43	73.82	47.24	X

Figure 17: Wireshark Stream Analysis N°1

The first line of Figure 17 reports statistics for the packets leaving the outer node’s interface, whereas the second lines represent the packets sent by the demultiplexer to itself, after fetching the data from the multiplexed frame. As we can see, the results are not satisfying at all, since almost 40% of the packets that arrived at the node were lost. Moreover, the jitter and the delay are way too high comparing to their values in the initial stream. Note that this packet loss occurs inside the software and not on the network.

b) With a short time out (0.5 ms) added inside the threads:

By adding a small delay in the loops we obtained better results (see figure 18). No packets were lost. Actually one packet was received twice, but this will probably not affect the quality of the communication at all. The mean jitter and the maximum jitter values are very close to the initial ones. The maximum delay is more important though, but this maximum value occurred only on one packet.



Detected 2 RTP streams. Choose one for forward and reverse direction for analysis

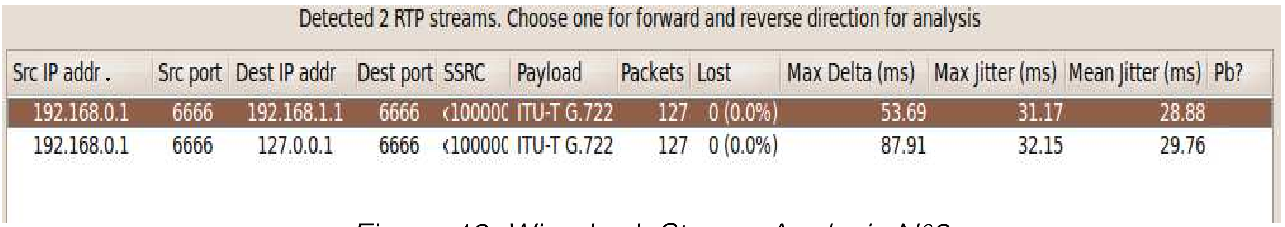
Src IP addr .	Src port	Dest IP addr	Dest port	SSRC	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)	Pb?
192.168.0.1	6666	192.168.1.1	6666	)x100000(	ITU-T G.722	210	0 (0.0%)	59.13	27.40	23.83	
192.168.0.1	6666	127.0.0.1	6666	)x100000(	ITU-T G.722	211	-1 (-0.5%)	73.09	28.30	24.42	

Figure 18: *Wireshark Stream Analysis N°2*

With these results we bring to the fore the necessity of adding a sleep call to the infinite loops of the threads running at the bridge level and at the inner node level. In the next test we will see if increasing the duration of the pause leads to better results.

b) With a longer time out (1ms) added inside the threats:

The results of the use of a 1 ms pause are shown in Figure 17. As we can see they are similar to the previous ones. In fact, the mean jitter at the reception is really close to the initial stream's jitter (less than 1ms longer). However, the maximum delay is 63% bigger than the initial delay, and in contrast to the previous test, many packets had a high delay.



Detected 2 RTP streams. Choose one for forward and reverse direction for analysis

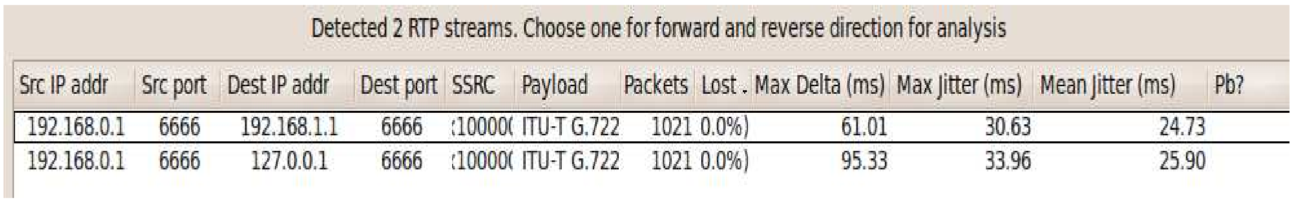
Src IP addr .	Src port	Dest IP addr	Dest port	SSRC	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)	Pb?
192.168.0.1	6666	192.168.1.1	6666	‹10000(	ITU-T G.722	127	0 (0.0%)	53.69	31.17	28.88	
192.168.0.1	6666	127.0.0.1	6666	‹10000(	ITU-T G.722	127	0 (0.0%)	87.91	32.15	29.76	

Figure 19: *Wireshark Stream Analysis N°3*

In the following tests we will use the short pause value (0.5 ms).

c) Longer streams

In the previous tests, we only transmitted a “few” packets (less than 210 packets). This test of our system is much longer, hence processing many more packets than before. During a test lasting 30 seconds, the outer node sends RTP frames with a 20ms period. The results of this test are shown in Figure 18.



Detected 2 RTP streams. Choose one for forward and reverse direction for analysis

Src IP addr	Src port	Dest IP addr	Dest port	SSRC	Payload	Packets	Lost .	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)	Pb?
192.168.0.1	6666	192.168.1.1	6666	:10000(	ITU-T G.722	1021	0.0%)	61.01	30.63	24.73	
192.168.0.1	6666	127.0.0.1	6666	:10000(	ITU-T G.722	1021	0.0%)	95.33	33.96	25.90	

Figure 20: *Wireshark Stream Analysis N°3*

The results that we get from this test are similar to the previous ones, so no effect is noticed in real VoIP exchange conditions.

4.2.3. Multiple simultaneous calls

Until now we have measured the performance of the system in terms of delay and jitter for only one call. Now we will consider a case where the new downlink transmission protocol can actually be useful, specifically when multiple users are inside the cell and they are streaming RTP traffic to and from other users outside the cell.

In order to simulate such a scheme, we modify the sender program to make it send RTP content to 10 different nodes. We only run one demultiplexer node, but we also add an iptables rule for the others to the aggregator, so that the multiplexed frame can also contain their traffic.

```
Iptables –A INPUT -p udp -d 192.168.1.0/24 --dport 6600:6700 -j
QUEUE.
```

The above iptables rule will forward all the UDP packets, with a destination port between 6600 and 6700 and destined to any node of the 192.168.1.0/24 sub network.

Because there are now ten different simultaneous RTP streams, the multicast frame is therefore much bigger, with an average length of 1400 octets. On the destination node we must also create as much IP aliases as necessary. The logical configuration for this test is shown in Figure 21.

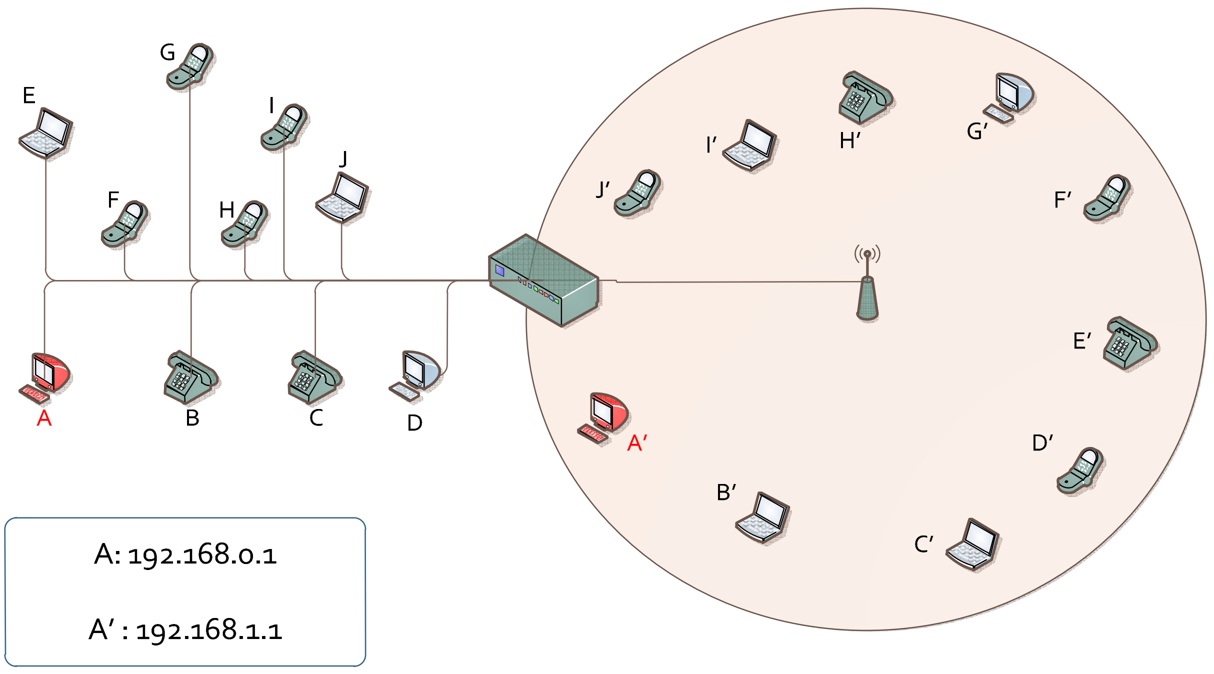
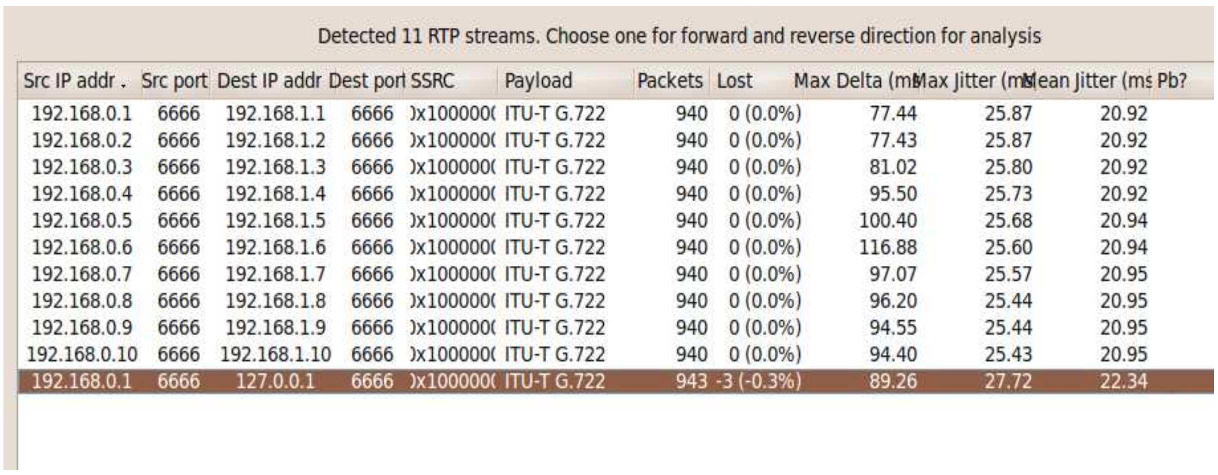


Figure 21: *10 outer nodes (A -> J) communicating with 10 inner nodes (A' -> J')*

We obtain the following results from this test. Note that since we used a single machine for simulating the outer nodes, the inner nodes and the aggregator, we are able to analyze all the streams in one single Wireshark session, allowing us to have a time synchronization.



Detected 11 RTP streams. Choose one for forward and reverse direction for analysis

Src IP addr .	Src port	Dest IP addr	Dest port	SSRC	Payload	Packets	Lost	Max Delta (m	Max Jitter (m	Mean Jitter (ms	Pb?
192.168.0.1	6666	192.168.1.1	6666	)x100000(	ITU-T G.722	940	0 (0.0%)	77.44	25.87	20.92	
192.168.0.2	6666	192.168.1.2	6666	)x100000(	ITU-T G.722	940	0 (0.0%)	77.43	25.87	20.92	
192.168.0.3	6666	192.168.1.3	6666	)x100000(	ITU-T G.722	940	0 (0.0%)	81.02	25.80	20.92	
192.168.0.4	6666	192.168.1.4	6666	)x100000(	ITU-T G.722	940	0 (0.0%)	95.50	25.73	20.92	
192.168.0.5	6666	192.168.1.5	6666	)x100000(	ITU-T G.722	940	0 (0.0%)	100.40	25.68	20.94	
192.168.0.6	6666	192.168.1.6	6666	)x100000(	ITU-T G.722	940	0 (0.0%)	116.88	25.60	20.94	
192.168.0.7	6666	192.168.1.7	6666	)x100000(	ITU-T G.722	940	0 (0.0%)	97.07	25.57	20.95	
192.168.0.8	6666	192.168.1.8	6666	)x100000(	ITU-T G.722	940	0 (0.0%)	96.20	25.44	20.95	
192.168.0.9	6666	192.168.1.9	6666	)x100000(	ITU-T G.722	940	0 (0.0%)	94.55	25.44	20.95	
192.168.0.10	6666	192.168.1.10	6666	)x100000(	ITU-T G.722	940	0 (0.0%)	94.40	25.43	20.95	
192.168.0.1	6666	127.0.0.1	6666	)x100000(	ITU-T G.722	943	-3 (-0.3%)	89.26	27.72	22.34	

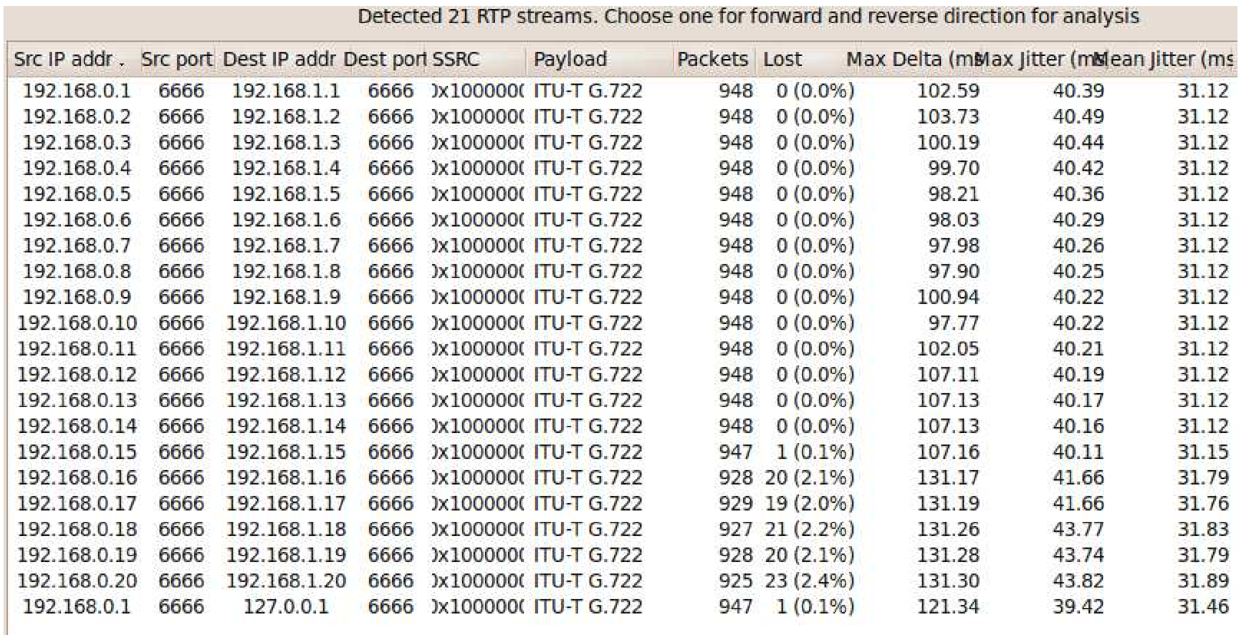
Figure 22: Wireshark Stream Analysis N°4

To get a jitter closer to 20ms, we changed the usleep value of the sender program, from 20000 (i.e., 20 ms) to 16000 (16 ms). This needed to be done because of the time it takes for the program to actually send the packets.

Comparing the first line to the last; we can see that the jitter greater when more RTP sub packet are contained within the multiplexed frame, even though the jitter is still relatively close to that of the original stream.

The maximum delay is also longer, but in comparison with the previous tests it is closer to the original stream (e.g., it is only 15% larger than the original maximum delay).

We increased the number of simultaneous communication simulated, up to 20, using the packETH software, and we obtain similar results. (See Figure 23.)



Detected 21 RTP streams. Choose one for forward and reverse direction for analysis

Src IP addr	Src port	Dest IP addr	Dest por	SSRC	Payload	Packets	Lost	Max Delta (m	Max Jitter (m	Mean Jitter (ms
192.168.0.1	6666	192.168.1.1	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	102.59	40.39	31.12
192.168.0.2	6666	192.168.1.2	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	103.73	40.49	31.12
192.168.0.3	6666	192.168.1.3	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	100.19	40.44	31.12
192.168.0.4	6666	192.168.1.4	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	99.70	40.42	31.12
192.168.0.5	6666	192.168.1.5	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	98.21	40.36	31.12
192.168.0.6	6666	192.168.1.6	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	98.03	40.29	31.12
192.168.0.7	6666	192.168.1.7	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	97.98	40.26	31.12
192.168.0.8	6666	192.168.1.8	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	97.90	40.25	31.12
192.168.0.9	6666	192.168.1.9	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	100.94	40.22	31.12
192.168.0.10	6666	192.168.1.10	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	97.77	40.22	31.12
192.168.0.11	6666	192.168.1.11	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	102.05	40.21	31.12
192.168.0.12	6666	192.168.1.12	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	107.11	40.19	31.12
192.168.0.13	6666	192.168.1.13	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	107.13	40.17	31.12
192.168.0.14	6666	192.168.1.14	6666	)x100000(	ITU-T G.722	948	0 (0.0%)	107.13	40.16	31.12
192.168.0.15	6666	192.168.1.15	6666	)x100000(	ITU-T G.722	947	1 (0.1%)	107.16	40.11	31.15
192.168.0.16	6666	192.168.1.16	6666	)x100000(	ITU-T G.722	928	20 (2.1%)	131.17	41.66	31.79
192.168.0.17	6666	192.168.1.17	6666	)x100000(	ITU-T G.722	929	19 (2.0%)	131.19	41.66	31.76
192.168.0.18	6666	192.168.1.18	6666	)x100000(	ITU-T G.722	927	21 (2.2%)	131.26	43.77	31.83
192.168.0.19	6666	192.168.1.19	6666	)x100000(	ITU-T G.722	928	20 (2.1%)	131.28	43.74	31.79
192.168.0.20	6666	192.168.1.20	6666	)x100000(	ITU-T G.722	925	23 (2.4%)	131.30	43.82	31.89
192.168.0.1	6666	127.0.0.1	6666	)x100000(	ITU-T G.722	947	1 (0.1%)	121.34	39.42	31.46

Figure 23: Wireshark Stream Analysis N°4

Here again, the last line should be compared to the first one. The jitter and delay values are higher than in the previous tests, because of the additional processing time required by needing to handle 10 more RTP packets (that have to be created and transmitted). Nevertheless, at the receiver, the delay and jitter values are quite similar to the parameters observed at the outer node level.

We also notice that some packets are lost in the original streams sent by 192.168.0.14-15-16-17-18-19-20. We cannot provide a consistent explanation for this problem, although we suppose the problem comes from the packETH software which might experience some problems increasing correctly the sequence number at the emission.

4-3 Results analysis

This set of tests, conducted with one single computer, allows us to validate the model proposed in this project. Indeed, even though experiments under real conditions have not been made, these results are encouraging. The next step is to measure the performance with an actual WLAN cell and several computers playing the roles of the inner nodes and the outer nodes. . However, from these earlier tests we observed that the aggregation processing and the demultiplexing tasks do not add too much delay, and do not lead to higher jitter values. Packet lost was minimal, so we expect that the quality of the RTP Stream is not affected by this new downlink protocol.

Due to a limited time for this project, we were not able to conduct all the tests we wanted to. It would have been interesting to analyze the behavior of the system with a real WLAN cell. Moreover, the use of a real SIP user agent would have allowed us to send and receive real encoded speech, and would also let use test the perceived quality of the sessions. Unfortunately, it is rather difficult to use such a software for both the callee and the caller on a single machine, that is the reason why we utilized to our sender program and the packETH software.

5 Conclusion

This project was a great opportunity to dig deeper into networking research. Conceiving a system, gaining knowledge of some new technologies, confronting the practical problems, and being able to modify and adapt the initial goals, were among the lessons that we learned. Moreover, our understanding of the protocols and technology improved thanks to this work.

As mentioned previously, the tests and experiments conducted to validate our model, but show that it can be improved. Thus the testing needs to be extended to real VoIP sessions under realistic conditions. Nevertheless, the tests showed that while there was some increase in the delay and jitter these increases seem to be quite small. This means that our system should not impair the perceived quality of the communication session.

Even though we did not manage to test the system with a large number of inner and outer users, we assume that it will support more concurrent VoIP sessions than the same cell could without this downlink multiplexing protocol at the same time, since the contention for the medium is significantly reduced. However, this should be verified and quantitated.

This new downlink transmission protocol is not intended exclusively for VoIP sessions, and could be adapted and extended for other similar kinds of traffic. Indeed, every protocol that requires exchanging small packets at a high rate and in a periodic fashion can benefit from our downlink model. For instance, we can imagine a Video on Demand service operating over a WLAN link. The frames carrying the video content can be aggregated at the AP level, and sent as a multicast to the users, who will simply demultiplex the packet to extract the appropriate content. Note that this can even be used to simply multiplex the audio and video RTP datagrams into a single flow of datagrams.

References

[1]: Gokul Rajagopalan, “802.11n Client Throughput Performance”, Aruba Networks White Paper http://www.arubanetworks.com/pdf/technology/TB_11NPERF.pdf

[2]: Jim Rendon, “WLAN and VoIP: A match made in heaven?”, Networking News, 12 Oct 2004 (Accessed on the 15th May 2010) http://searchnetworking.techtarget.com/news/article/0,289142,sid7_gci1015068,00.html

[3]: Guillaume Collin and Boris Chazalet, “Exploiting cooperative behaviors for VoIP communication nodes in a wireless local area network”, Project in Computer Communications, Department of Communication Systems, Royal Institute of Technology (KTH), 4th March 2007 http://web.it.kth.se/~maguire/Boris-Chazalet_and_Guillaume-cooperative-behaviors-final-report-20070816.pdf

[4]: P. Cameron, D. Crocker, D. Cohen, J. Postel, “Transport Multiplexing Protocol (TMux) ”, IETF, RFC 1692, August 1994. http://www.rfc-editor.org/rfc/rfc1692.txt

[5]: K. Nichols, S. Blake, F. Baker, D. Black, “Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers”, IETF, RFC 2474, December 1998. http://www.ietf.org/rfc/rfc2474.txt

[6]: Wikipedia Article, “Maximum transmission unit”, (Accessed on the 14th May 2010) http://en.wikipedia.org/wiki/Maximum_transmission_unit

[7]: Pablo Brenner, “A technical tutorial on the IEEE 802.11 Protocol”, BreezeCom White paper, 1997 http://www.sss-mag.com/pdf/802_11tut.pdf

[8]: Alessandro Rubini, “Virtual Network Interfaces”, (Accessed on the 10th May 2010) http://www.linux.it/~rubini/docs/vinter/vinter.html

[9]: J. Corbet, A. Rubini, G. Kroah-Harman, “Linux Device Drivers” 3rd Edition, O’Reilly Collection, February 2005

[10]: Bridge Linux Foundation, Bridge webpage, (Accessed on the 15th May 2010) http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge

[11]: Rusty Russel, « Linux 2.4 Packet Filtering HOWTO », January 2002, (Accessed on the 15th May 2010) http://netfilter.org/documentation/HOWTO//packet-filtering-HOWTO.html

[12]: Rusty Russell and Harald Welte, “Linux netfilter Hacking HOWTO”, July 2002, (Accessed on the 15th May 2010) http://netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html

[13]: Chris, Christy and Frank, “The quick intro to libipq”, Accessed on the 15th May 2010. http://www.imchris.org/projects/libipq.html

[14]: Greg Ippolito, YoLinux team, “POSIX thread (pthread) libraries”, Accessed on the 20th May 2010 http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

[15]: Mixter, BlackCode Magazine, “A brief programming tutorial in C for raw sockets”, Accessed on the 20th May 2010 http://mixter.void.ru/rawip.html

[16]: Dave Marshall, “IPC:Interrupts and Signals”, May 1999, (Accessed on the 20th May 2010) http://www.cs.cf.ac.uk/Dave/C/node24.html

[17]: PackETH source forge web page , (Accessed on the 20th May 2010) http://packeth.sourceforge.net/

TRITA-ICT-EX-2010:105