Exploiting cooperative behaviors for VoIP communication nodes in a wireless local area network

Guillaume Collin
Boris Chazalet

Project performed at Department of Communication Systems (CoS)

Royal Institute of Technology (KTH)

4th of march 2007

ABSTRACT

The purpose of this project was to implement a new design for VoIP communication in a wireless local area. One of the main goals was to experiment with a new approach that exploits cooperative behaviors. By cooperative we mean that the nodes try to cooperate so as to achieve higher cell capacity.

VoIP communications has such specific data flows that it is interesting to build protocols that exploit these characteristics, for example for speech these characteristics are: regularity and time-constrained delivery. Furthermore the wireless local area network brings another dimension in terms of all of the nodes sharing a wireless cell, as the nodes will share a common bandwidth-limited media. In the current standard for wireless local area networks the computers in the cell have to contend to get access to the shared media which can lead to delays, even if the bandwidth needed is much lower than the bandwidth available. If we focus on the needs of VoIP, one way to improve this is for all nodes to cooperate by organizing themselves to efficiently share the media. If they can agree with each other on what time each node will transmit, then they don't need to contend anymore. Thus the VoIP packets can reach their destination on time and while minimizing the use of resources in this cell.

In this project we have designed and test a proof of concept of this cooperation - in order to prove that it can work and to show what effects and performance it implies. This work can also be seen more generally as context-awareness research - where the context-awareness is used to provide input to the cooperative behavior.

Table of Contents

1 INTRODUCTION

1.1 Context and Delimitation

The study considers in a single IEEE 802.11b/g wireless local area network (WLAN) cell managed by a single access point with multiple VoIP (Voice over IP, i.e voice calls over the Internet) nodes within the cell. The communication sessions are established via the SIP protocol between two SIP user agents. We will assume that one of the parties is in the cell and all other parties in a session are outside the cell. [This assumption means that the media streams all must go through the access point and further to the infrastructure, rather than simply being terminated in the same cell.]

1.2 Introduction to the problem & Initials goals

We want to apply a new approach to VoIP media streams in a WLAN cell. We are interested in studying and exploiting the effects of cooperative behavior by the various nodes on the uplink and the aggregation of VoIP media packets on the downlink.

All the nodes in a wireless cell share the media. Usually, each node competes to access the media according to the IEEE 802.11 standard. The great majority of wireless network nowadays WLANs use one of the IEEE 802.11 media access and control protocols.. At the same time, you may have noticed that the today VoIP communications is incredibly increasing and with the latest generation of smart phones and wireless PDAs incorporating VoIP support. Not suprisingly VoIP communications is becoming one of the most important application for WLANs.

Because the VoIP needs only small and regular amounts of data, one would expect that a large number of VoIP users could be supported in a single WLAN cell whereas wired networks deal generally with applications requiring desynchronized and great amounts of data. Thus we would expect to find more and more WLANs that will be mostly used for transfer voice calls. However, today the number of calls which can concurrently take place in a single WLAN cell is much less than would be expected based upon the bandwidth available versus the bandwidth required by each call, this has motivated the work in this project.

Imagine the ideal case where a WLANs was transfering voice calls and no other types of data. A network that can handle a huge number of nodes sharing the same media so everyone would have to fight with everyone to get the chance to use the media. But as we know that each node only needs to access the media during such a short time every 20 milliseconds - thus if the nodes were organized and could agree each other upon a schedule, everyone's communications could be interleaved in such a way that the traffic would be nearly time multiplexed. Hence the nodes could significantly reduce the overhead caused by contention - by explicitly avoiding contention.

Here is the concept : we want everyone to agree upon a schedule, we avoid the contention for the media and that is what we call a cooperative behavior. How will such cooperative behavior affect the network's performance ? First, as we consider mainly the VoIP application, this will fulfill the most important VoIP need which is we know that each packet will be sent roughly 20 milliseconds after the previous one, thus we would like no one to contest the use of time. Moreover because the nodes will no longer contend for access, this will improve the to use of the media : as all the data packets will follow each other without unnecessary gaps (due to backoff) in between. In

concrete terms that means that we will be able to accept more VoIP calls than with the present WLAN standard into a same cell.

However, there is an issue of fairness. Most of the time a WLAN is setup as a managed architecture; which means that a specific entity, the Access Point (AP) manages the media for all the nodes involved in the WLANS cell. This AP acts as the gateway towards other networks so that every data packet going out has to go through the AP and of course every packet incoming also does. Since everyone, including the Access Point, will get its own time slot in order to send packets but whereas a node needs to send one packet at a time, the Access Point will need to send as many as there are nodes (since we have assumed that each node is communicating with an outside party) within the same allocated slot time. We may correct this by aggregating all the incoming packets into one big broadcast packet that the AP will be send when its turn comes opportunity to access the media - it will send more data with each such opportunity. For analysis of how nodes will share the WLAN media when they have unlimited amounts of traffic see the publications:

Martin Heusse, Franck Rousseau, Gilles Berger-Sabbatel, and Andrzej Duda
"Performance anomaly of 802.11b", In Proceedings of IEEE INFOCOM 2003, San Francisco, USA, March 30-April 3, 2003
http://drakkar.imag.fr/IMG/pdf/perfAnomaly-infocom.pdf

Martin Heusse, Paul Starzetz, Franck Rousseau, Gilles Berger-Sabbatel, and Andrzej Duda,
"Scheduling Time-sensitive Traffic on 802.11 Wireless LANs",
In Proceedings of the 4th COST 263 International Workshop on Quality of Future Internet Services (QoFIS 2003), Stockholm, Sweden, October 1-3, 2003
http://drakkar.imag.fr/IMG/pdf/qofis.pdf

Gilles Berger-Sabbatel, Andrzej Duda, Olivier Gaudoin, Martin Heusse, and Franck Rousseau. Fairness and its Impact on Delay in 802.11 Networks. In Proceedings of IEEE GLOBECOM 2004. Dallas, USA, November 29-December 3, 2004.
http://drakkar.imag.fr/IMG/pdf/globecom2004.pdf

Gilles Berger-Sabbatel, Andrzej Duda, Martin Heusse, and Franck Rousseau. Short-Term Fairness of 802.11 Networks with Several Hosts. In Proceedings of the Sixth IFIP IEEE International Conference on Mobile and Wireless Communication Networks, MWCN 2004. Paris, France, October 25-27, 2004.
http://drakkar.imag.fr/IMG/pdf/mwcn2004.pdf.

1.3 Main goals

We divide up the problem into several part, first the aggregation of traffic on the downlink, the we consider the uplink - first the issue of finding a schedule and secondly the switching between unicast and multicast modes (once a call has started).

We begin by considering the downlink part which needs to aggregate the traffic going from the Access Point to any of the local nodes in the cell. Besides there are a lot of issues that come. If we now aggregate the VoIP packets into a big broadcast packet, we will loose informations about the packet's real destination. Then the broadcast packet will be sent and each of the nodes will check if there is a RTP packet for it inside this multiplexed packet (which we can treat as a {time} multiplexed frame). This part has obviously to be done first since we will used these broadcast packets as a time reference to start the part for the individual nodes within the cell. After this multicast the nodes will be allowed to transmit one after another.

This brings us to the the uplink part. First we must organize the local nodes into some logical order so that they agree upon a shared schedule. While we want them to cooperate, we must also handle new starting calls and the nodes which don't want to (or cannot) cooperate. The task of our uplink processing is mainly to desynchronize the nodes, to avoid them stepping on each other's packets rather than requiring a perfect time synchronization (as in TDMA). Later in the report we will show that we can cause the nodes to cooperate by using the AP to gather information about the nodes. We will also examine how the data can be directed to the correct destinations.

The third issue concerns how to start the mechanism - as a call will start out sending and receiving RTP packets in unicast mode and we would like it to quickly transition to using multicast mode (if and only if it is capable of doing so). In the uplink part, we assume that a node that starts a call will actually cooperate. We must also consider what happens when nodes move in and out of the cell. Note that they can arrive from or depart to another cell - which can even be part of another network (for example GPRS).

1.4 Minisip – For the purposes of this project we have used a locally developed SIP [1] client called Minisip [2]

1.4.1 Introduction to Minisip

Minisip is a user agent using the SIP protocol. Basically, Minisip allows users to make calls over Internet by using the VoIP technology. It offers other useful features as well, such as instant messaging, video conferencing, secure communications. Minisip was partially developed by masters students at the Royal Institute of Technology along with volunteer developers. The source code is available under the GNU Lesser General Public License (LGPL) for the libraries and for applications under the GNU General Public License (GPL). Within our project, we modified the source code of Minisip in order to be able to set up calls and to test our new design. While, Minisip is available for several platforms, such as Linux, Linus for embedded system, Windows, or Pocket PC, we are particularly interested in the Linux and the Pocket PC environments. Indeed, we developed the project under Linux (specifically Ubuntu dapper [3]) and we developed out application assuming the mobility which is offered by the Pocket PC. The primary distinction is that the Pocket PC devices are quite small and thus we can assume that a user might use one in a similar fashion to which they use a cordless phone or a cellular phone.

1.4.2 Minisip on Pocket PC

We used the hp iPAQ Pocket PC h5550 series with Windows CE 2003 [4]. Below we will explain how to install Minisip, but first we will examine how to get all the tools and the needed documentation needed to successfully make Minisip work.

1.4.2.1 Source code of Minisip

The source code for the HP Pocket PC platform can be found in the Minisip svn [5] repository with the following commands. We can use either the last stable version:

 # svn co svn://svn.minisip.org/minisip/trunk

or the version dedicated to Pocket PC:

 # svn co svn://svn.minisip.org/minisip/port_PPC_VS2005

1.4.2.2 Compilation

Minisip is built on four distinct parts (libraries) which are interconnected. It is necessary to build them in the right order in order to satisfy the dependencies. There are two ways to build Minisip. It can be built as a big project with sub-projects as libraries (by using the last version) or each library can be a project as in the case of the Pocket PC version. We used Microsoft Visual Studio 2005 to compile and build Minisip. The documentation to compile, debug, and build Minisip can be found here [5]. It shows step by step what the necessary additional libraries are, how to link the libraries, how to set up and compile a project, and how to fix the compilation errors due to the ARM architecture of the Pocket PC.

1.4.2.3 Installation

Once Minisip is built, the libraries must be linked according to the documentation, and the files copied to the Pocket PC device. At this point we have to correctly configure Minisip. Minisip is designed to work with a SIP proxy which forwards the SIP requests and Responses. For this purpose we used OpenSER [6].When Minisip starts, it creates two XML format configuration files: *minisip.conf* and *minisip.addr*. In minisip.conf, we have to enter the SIP proxy parameters (the proxy port and address), and specify the correct sound drivers “wave:” in the sound_device section. Changing the XML file minisip.addr is not compulsory but can be useful for testing as you can predefine specific SIP URLs.

Although Minisip was installed and initialization was successfully done, for unknown reasons, it does not transmit any data on its wireless interface when we called another User Agent. Due to our limited time, we chose to not continue to use Minisip on the Pocket PC and decided to return to this problem later(if there was sufficient time). While we were able to execute another SIP User Agent, SJPhone [7], unfortunately, the source code for this SIP UA was not available, hence we were not able to modify it. However; its successful execution convinced us that the Pocket PC device would be usable – if we were able to correct the problem with the Minisip for this platform.

1.4.3 Minisip on Linux

We used the Linux distribution called Ubuntu Dapper [3].

1.4.3.1 Source code and compilation

The source code for the Linux version of Minisip can be found in the svn repository. The lastest stable version can be downloaded with the following command:

 # svn co svn://svn.minisip.org/minisip/trunk

Before compiling Minisip, we needed to install all the packages which the Minisip libraries depend

upon. This can be done by following the steps on the website [8] or with the following two commands:

```
apt-get -y install subversion perl libtool libltdl3-dev automake1.9 g++ make
apt-get -y install pkg-config libssl-dev gawk
```

Both a graphical and text mode user interface are available on the Linux version. In order to enable both these modes, we have to modify the file *~/trunk/build.d/build.conf* and replace the 0 by 1 in the lines which enable the textui or GTK interface.

```
my %minisip_params = (
        %common_params,
        %common_minisip_params,
        gtk => 1,       # enables GTK+ interface
        p2t => 0,       # enables Push To Talk support
        qtui => 0,      # enables Qt interface
        qte => 0,       # enables Qt Embedded interface
        textui => 1,    # enables the text based user interface
);
```

Now, Minisip is ready to be compiled and installed by using the following command:

 # ./build.pl run minisip

This command launches Minisip with the graphic interface. Since we want to use the text user interface as it is easier to use for developing and performing tests, we used the script run.minisip.sh to launch Minisip. One line of the script needs to be modified since the path to the executing file was wrong.

```
MINISIP_PLUGIN_PATH="$minisip_plugin_path_choose" \
      LD_LIBRARY_PATH="$MY_LD_LIB_PATH" \
            install/x86-pc-linux-gnu/usr/bin/minisip_textui
```

1.4.4 Modifications of Minisip

In order to test our implementation, we needed to find a way to redirect the RTP packets generated by a User Agent to another User Agent. Several solutions were feasible. We can intervene at different levels in the OSI stack. The lowest possible level is the network (IP)layer. Or comes at the transport layer (using UDP), or at the application layer using the SIP protocol. Within the framework of our project, we chose to operate at the level of the transport layer, i.e., operating upon the UDP packets.

Since every communication session in the cell has two streams (one on the downlink and one on the uplink), we also have modified Minisip in two ways. One for the downlink case and the other for the uplink case. For both modifications, we changed the remote IP address and the remote port. Minisip manages each call as a session (Session.cxx). Each session contains a media stream receiver and a media stream sender (MediaStream.cxx). We focused on the media stream sender where we can find all the parameters of the VoIP session such as the remote IP address, the remote RTP port, and the codec used to encode the voice. By changing the remote IP address and the remote port, we

can easily redirect the RTP packet to the desired destination.

The main problem with this modification is that the real remote address and the remote RTP port are missing in the RTP packet what is a problem since we have to issue the RTP packet to the right recipient. In order to remedy this problem, we add the remote IP address and the remote port at the end of the RTP packet before sending it on the network.

1.4.4.1 Modification in detail

The two modified files are *MediaStream.cxx* (Appendice 9.1) located in *trunk/libminisip/source/ mediahandler/* and *MediaStream.h* located in *trunk/libminisip/include/libminisip/mediahandler/*. First of all we create, in the header file, a new buffer *dataIpPort* which will contain the voice data, the remote address and the remote port. We create as well a new object *IPAddress* which will contain the new remote IP address. Then, in *MediaStream.cxx*, the remote IP address and the remote RTP port are concatenated with the following syntax: *xxx.xxx.xxx.xxx:xxxxx.*

```
memcpy(dataIpPort,data,length);
memcpy(dataIpPort+length,remoteAddress->getString().c_str(),
remoteAddress->getString().size());
memcpy(dataIpPort+length+remoteAddress->getString().size(),":",1);
memcpy(dataIpPort+length+remoteAddress->getString().size()+1,
&remotePort,sizeof(remotePort));
uint32_t                  lengthIpPort=length+remoteAddress->getString().size()
+sizeof(remotePort)+1;
packet = new SRtpPacket( dataIpPort, lengthIpPort, seqNo++, lastTs, ssrc );
```

Finally, we create a new remote address and we change the remote port before sending the new RTP packet.

```
proxyAddress = new IP4Address("127.0.0.1");
packet->sendTo( **senderSock, **proxyAddress, 1206);
```

The differences between the uplink and the downlink are the remote address and the remote RTP port. We will see later in detail how the RTP packets are redirected.

1.4.5 Use of Minisip

As we saw before, we use Minisip with the text mode by running the file *run.minisip.sh*. The text mode uses a configuration file *.minisip.conf* located the user home directory. In this file the different parameters such as the the account name, the proxy address or the proxy port have to be set up. You can refer to the example of *.minisip.conf* file in appendice 9.3. As we specified before, Minisip must use a SIP proxy to work correctly. However, it is possible to avoid the installation of a proxy by using the remote user agent address and port as the proxy address and the proxy port in the configuration file. It is clear that this solution is not acceptable and it might be an improvement that to allow Minisip to work without any proxy.

The command to call a remote user agent is *call sip:name@domain.com*. We noticed that we were not able to call a Pocket PC using SJPhone without any account name specified in the configuration file.

Later in this report, we will refer either to Minisip or SIP agent/software to designate it. We

always work with Minisip as modified user agent but we could have chosen any open SIP agent and modified them a bit in order to achieve our purpose.

2 DOWNLINK

We start our project with the implementation of the downlink protocol. We will see that the implementation of this protocol was needed in order to make the uplink protocol work.

2.1 Overview

As we stated earlier, a VoIP session generates a periodic stream of constant size packets. If we take consider of Minisip when using the G711 codec, the packetization interval is 20 ms and the RTP packet size is 172 bytes. We must note that the packetization interval depends on the CODEC which is used and can vary somewhat [9].

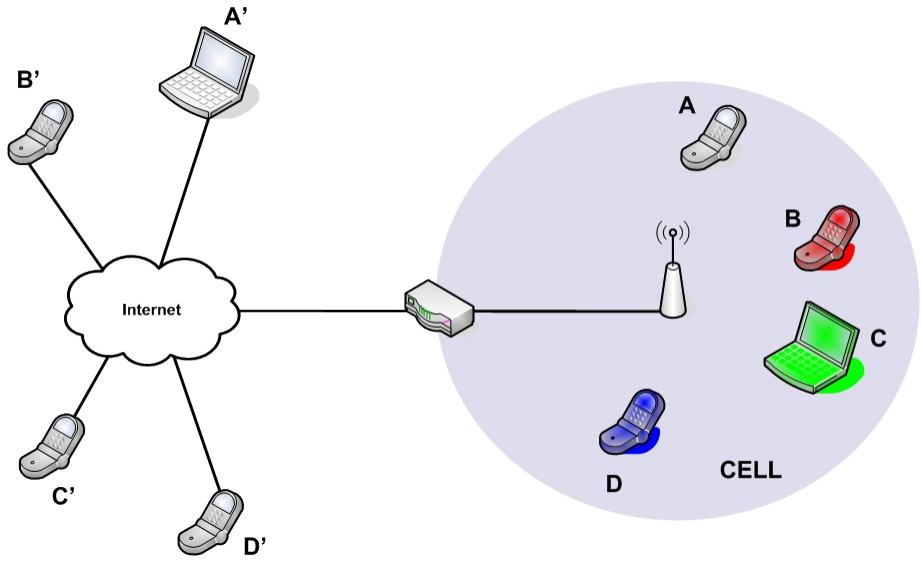

Figure 1: Four VoIP sessions established between four outer nodes and four inner nodes

If we consider four VoIP communications between four nodes (A', B', C', and D') and four nodes (A, B, C, and D) in a cell, we will obtain the following incoming traffic arriving at the access point.

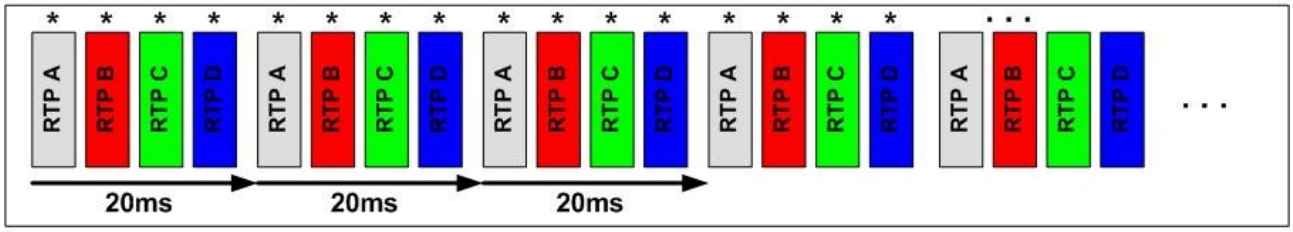

Figure 2 : RTP traffic incoming and outgoing from the access point (note that the order of the packets in each interval may or may not be in the order shown)

The access point would forward each incoming packets to the destination within the cell. Each node within the cell (including the AP) has an equal chance to get an access to the medium

what implies that the access point will have 1/N of the bandwidth (with N-1 nodes and the AP within the cell). However, the access point has N-1 packets to forward every 20 ms, this means that it has to contend N-1 times for access to the media every 20ms. From another point of view, with N-1 nodes within the cell and each node having an ongoing communication with a node attached to the Internet beyond the AP, we will have $2*(N-1)$ packets in 20ms (N-1 for the downlink and N-1 for the uplink). The access point is responsible for sending N-1 of these packets, hence it needs significantly more bandwidth than any other single node. We notice that it could be interesting to group together all the packets arriving within the same of 20ms, thus reducing the need to contend for the media within the cell. Indeed, instead of trying to access to the media N-1 times, the access point will only have to forward one bigger packet every 20ms.

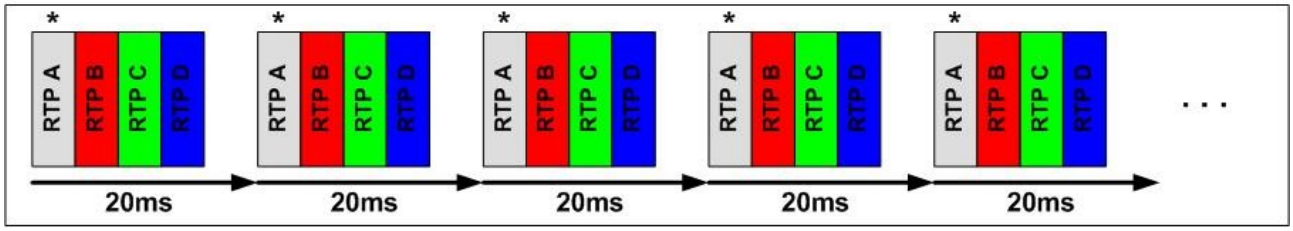

Figure 3 : The access point contends for the media and sends all RTP packets in a single larger multicast packet

This means that we will multiplex and demultiplex the the RTP packets, the proxy/access point will need to buffer the incoming RTP packets, aggregate them within a new packet, then broadcast or multicast this newly formed packet to all the nodes in the cell. As each of these nodes is expecting a single RTP packet, we need to demultiplex this packet into the correct incoming packet for this node. Then it will be delivered internally to the user agent that should have received it.

Two different components have to be distinguished in the downlink protocol. The first one performs the buffering, multiplexing, and broadcasting, and the second one demultiplexing and local forwarding of the RTP packets. The first part takes place at the access point (or a proxy between it and the Internet) and the second part takes place on the receiving node itself.

2.2 Location of the processes

In our project, we implemented the first part as a proxy since we were not able to modify the firmware of the access point. The second part is implemented as a local program running on the receiving nodes. The proxy acts as a server which waits for the incoming RTP packets on an UDP socket. It uses the IP address 192.168.0.3 and the port 32666. So that the outer nodes are able to send their packets to the proxy instead of to one of the inner node. We use the modified version of Minisip which we saw previously in the part 2.5. The local program acts as a server which listens to the broadcast packets on the port 5066. The local problem demultiplexes the packet and deliverers the packet locally to the correct port.

2.3 Packet structure

2.3.1 Structure of the packets issued by Minisip

As we explained previously, we use a modified Minisip. This Minisip stores the remote IP address and the port at the end of the RTP packets which contain the voice data. The remote IP address and the remote port are separated by ":" what will allow us recognizing the end of the

remote IP address and the beginning of the remote port. The packets have the following structure:

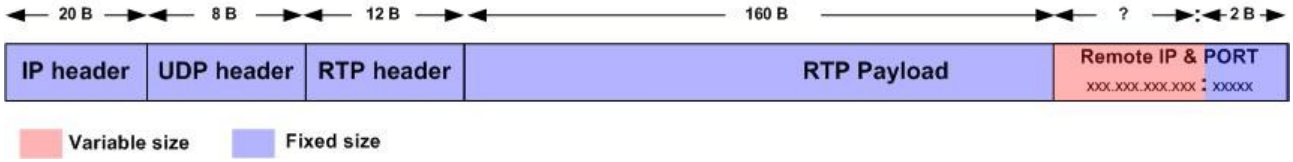

Figure 4 : Structure of a modified RTP packet

The header and the payload of the RTP packet are known and do not vary. The size of the header is 12 Byte and the size of the payload is 160 Bytes (with the use of the G.711 codec). Then we know that the remote IP address will start at the 173rd Bytes. Since the IP address has a fixed size (32 bits), we know where the remote port is (the port number is encoded a *uint_16* which has the size of 2 Bytes).

2.3.2 Structure of the packet created by the proxy

The multiplexed packet created by the proxy has to provide the information necessary for each of the clients to extract the data which is relevant to them. A broadcast packet is used for the downlink protocol. The uplink protocol will be explained later in this document but we use the downlink packet to provide information about the order of the nodes to send uplink packets. This order is stored as a list at the start of the broadcast packet. It begins with the size of the list (an integer of four bytes) followed by the list, each element of this specifies the IP address and the port of each node. After this, the downlink RTP packets are multiplexed in their order of arrival at the proxy. The structure of the packet is shown in the figure bellow:

Figure 5 : Structure of a broadcast packet

The RTP packets are simply aggregated one after the other. We do not modify the structure of the packet. Thus each packet already contains the information about which IP address and port it is.

2.4 The broadcast packet as a time reference

The broadcast packets play an important role in the new protocol, as well for the uplink as for the downlink. Indeed, the downlink protocol needs to regularly deliver a broadcast packet so that the inner nodes which receive the packet have the time to demultiplex it and forward it on the local interface. Since these tasks are time consuming and since the time interval between two normal RTP packets is not exactly 20 ms, the broadcast packets are issued every 15ms. These parameters on the downlink are not the only ones to take into account. We have to consider the uplink as well. In a few words, the uplink protocol aims to synchronize the nodes within the cell. To do so, we need to have a time reference that all the nodes will be able to use. The best reference which we can have is the broadcast packet. It is regularly issued and all the nodes can receive it since it is a broadcast. Then the nodes wait for this broadcast packet before to be able to issue their own RTP packet. In the same way than for the downlink, the tasks of the uplink protocol are time consuming, so we have to adjust the time interval between two broadcast packets so that the remote note is able to receive the packets on time.

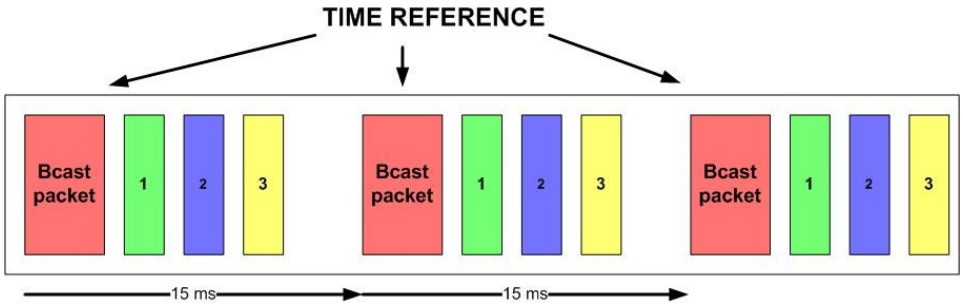

Figure 6 : Broadcast packet as a time reference

2.5 The downlink protocol On the Access Point

The implementation of the proxy was a hard task since the design had to consider the real-time constraints.

2.5.1 Constraints

An important constraint was the real time constraint. Indeed, time is extremely important in VoIP communications since the audio quality, the comprehension of a message depends on the time arrival of the RTP packets. The loss or the excessive delay of a RTP packet reduces the quality of the perceived sound. The proxy needs to be able to forward the RTP packets quickly and it must support the maximum number of nodes which might be in the cell.

The second constraints is because we want to use the downlink broadcast/multicast for time synchronization we would like it to be transmitted periodically. The variance in the transmission time will be reflects in the variance which the nodes in the cell must deal with.

2.5.2 Environment of programming

We explored several solutions to take into account these major constraints. Firstly, we had to choose the programming language. Due to the low level of programming, C seemed to be the most appropriate language. Then, we chose our programming environment consisting of Linux and Vi. We created a Makefile in order to compile the source code and satisfy all the dependencies.

2.5.3 Socket programming

In order to intercept the RTP packet from the outer node, we set up certain number of UDP sockets. The First pair of socket takes place between the outer node which uses a modified Minisip and the proxy within the cell. The parameters of the socket are the following:

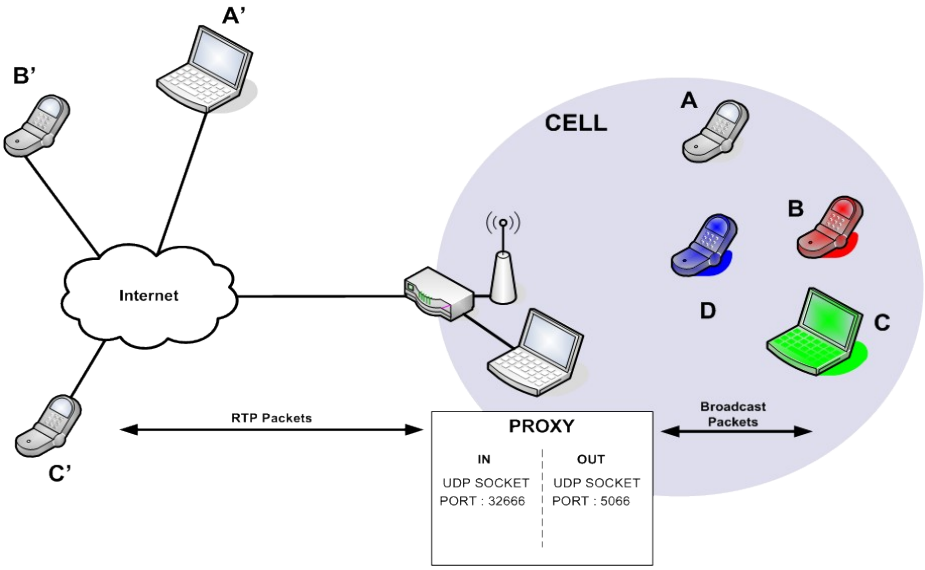

Figure 7 : Creation of UDP Socket

The outer nodes need to send their RTP packets which are contained in UDP packets to the proxy. The proxy listens to the incoming UDP packet on the port 32666 which is thus known by all the outer nodes. The sending socket on the outer nodes uses a random port number between 1024 and 65000.

The second pair of sockets is between the proxy and all the inner nodes (node within the cell). This socket is different than the previous one since the packets are broadcasted. Then we need to use the c function *setsockopt()* to set up the broadcast option. The remote address becomes 255.255.255.255 (*INADDR_BROADCAST*) and the remote port is 5066.

```
struct sockaddr_in sock;
int sk,n_sent,optval;
if( (sk=socket(PF_INET,SOCK_DGRAM,0)) < 0 ) {
        printf("socket creation error!\n");
        return ;
}
optval=1;
setsockopt(sk,SOL_SOCKET,SO_BROADCAST,&optval,sizeof(optval));
sock.sin_family = AF_INET;
sock.sin_addr.s_addr=htonl(INADDR_BROADCAST);
sock.sin_port = htons(REMOTEPORT);
```

All the inner nodes listen to broadcast packets on the port 5066. We will talk about the last pair of socket which is in charge to forward the RTP packets on the local interface later (in section 4.6).

2.5.4 Implementation of the multiplexing program

The multiplexing program needs to be as simple as possible in order to allow him to absorb a huge number of RTP session. It has to do the minimum number of actions on the incoming RTP packets and performs multiplexing only. That's why we have only three tasks in the program. The first one is in charge to read the incoming data in the socket and copy them into a buffer.

```
n_read = recv(sk,buf,MAXBUF,0);
```

The second task consists in aggregating the incoming RTP packet contained in the buffer

after the packets which are already arrived (already in the broadcast packet). We use an offset which indicates the place where to begin to copy of the incoming packet.

```
memcpy(bcastPacket+lengthBcastPacket,buf,n_read);
```

The third task actualizes the offset by adding the size of the incoming packet. Thus, the offset always indicates the beginning of the copy in the broadcast packet.

```
lengthBcastPacket=lengthBcastPacket+n_read;
```

The buffer and the broadcast packet are both tables of characters. The maximum size of the buffer is RTP packet size + remote IP address size + remote port size, namely 172+16+2=190Bytes. The maximum size of the broadcast packet is arbitrary. We utilized a large limit (10000). This will be large enough to store 52 RTP packets along with the uplink list.

These three tasks are in an infinite loop. The function recv() which copy the incoming data in a buffer is blocking since we don't need to do the tasks two and three without of incoming RTP packets. In the case of simultaneous communications, if we receive a RTP packet before the tasks two and three are done, the operating system automatically store the incoming packet in a buffer and deliver it to socket later. Thus we do not lose any incoming packets. The simplicity and the reduced number of tasks allow the multiplexing part to absorb a heavy load of RTP traffic. Now that the RTP packets are multiplexed in the broadcast packet, we need to send it within the cell what is the purpose of the following paragraph.

2.5.5 Sending the broadcast packet and time reference

As we said previously in the constraints, the broadcast packet is a time reference, for the uplink protocol. We prefer that this should be quite regular – so that all the nodes will get a chance to send their outgoing packets within a reasonable time. In order to respect this constraint we needed this part to work independently from the multiplexing work. That why we chose to work with independent threads. The first thread multiplexes the RTP packets and the second one sends the broadcast packet within the cell. Since the second thread works independently, a heavy load of RTP packets will not affect the performance of the second thread and will not add an extra delay the interval between to broadcast packets. However, we need now to find a way to precisely set up the time between two broadcast packets. Or in the c language we did not find a library which allowed us to have such an accurate timer. Indeed we need to set up a time in milliseconds. The solution was to find a derived function from *sleep()* which is *usleep(useconds_t useconds)* in the library unistd.h. We created an infinite loop which starts with *usleep(PERIOD)*. Thus, every peace of code within the loop are executed every PERIOD us. This loop contains the function which sends the broadcast packet and re-initializes it. The re-initialization is time consuming, so we have to take into account this time by reducing the PERIOD time between two packets.

The two threads need to share the buffer of the broadcast packet since the first one aggregates the RTP packet inside and the second one sends and re-initializes it. That's why the broadcast packet is a global parameter. To avoid conflicts between the two threads we need to use semaphores. We were not able to implement a semaphore yet. See appendix 9.4 for more details of the code.

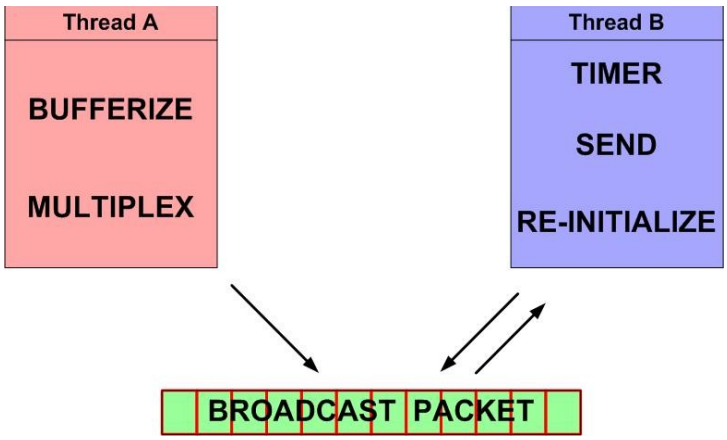

Figure 8 : Independent tasks of the threads accessing to the same buffer

2.6 The downlink protocol on the local nodes

The program on the local node has a simple task - listen for a broadcast packet to port 5066, demultiplex the incoming packet, and forward the right RTP packets to the local port of Minisip. In our implementation, the local port number is specified in the extension of the RTP frame, hence we only need to extract the frame (based upon matching the "remote" IP address against our address and then forwarding the packet locally.

2.6.1 Listening the broadcast packets

The local program is divided into three independent threads. The part of the program which concerns the downlink is run by the first thread. The second and third threads are part of the uplink protocol which we will describe later. The first thread is pretty simple. We create a normal socket listening on the port 5066 to the broadcast packets. Each time there is an incoming broadcast packet, the data are copied into a buffer and then this buffer is analysed with the help of a library to be explained bellow.

2.6.2 Analysing, demultiplexing and forwarding the RTP packets

We created a library to make easier the analysis and demultiplexing of the incoming broadcast packets. Here are the functions of the library:

```
//Create an empty message (malloc)
buffered_msg* msg_create(unsigned short int rtp_packet_size);
//Delete a message (free)
void  msg_delete(buffered_msg* msg);

/*SET/GET functions*/
void  msg_set_rtp_packet(buffered_msg *msg, char *rtp_packet, unsigned short int length);
int   msg_set_ip_dest_str(buffered_msg *msg,char *ip_string);
int msg_set_udp_port(buffered_msg *msg, unsigned short int udp_port);
unsigned short int msg_get_udp_port(buffered_msg *msg);
int msg_set_length(buffered_msg *msg, unsigned short int length);
unsigned short int msg_get_length(buffered_msg *msg);
char* msg_get_ip_dest(buffered_msg *msg);
void msg_print_it(buffered_msg *msg);

/*Read & Write buffers and socket*/
```

```
int msg_read_socket(char* buffer, int socket);
int msg_read_a_msg(buffered_msg *msg, char* buffer, int offset);
void msg_into_big_buffer(buffered_msg *msg, char* buffer, int *length);
void msg_send_it(buffered_msg *msg, int socket);
int msg_udp_socket_connect(buffered_msg *msg);
int msg_create_socket(int type,int *ptr_port, struct sockaddr_in *ptr_adresse);
int msg_get_ip_from_bcastpacket(char* buffer, char* tmp);
unsigned short int msg_get_port_from_bcastpacket(char* buffer, int *offset);
```

When a broadcast packet is received, the first task is to extract the list from the beginning of the packet. Then the first RTP packet is extracted with the function *int msg_read_a_msg(buffered_msg *msg, char* buffer, int offset)* . All the informations such as the RTP payload, the remote IP address or the remote port are stored in the structure buffered_msg. Now we are able to compare the parameters of the newly extracted RTP packet with those of Minisip. Thus we compare the remote IP address with the one of the node. If the remote IP address match, the UDP payload (RTP header + RTP payload) is sent on the local interface with the function *void msg_send_it(buffered_msg *msg, int socket).* We have to notice that the remote IP address and the remote port are removed from the end of the RTP payload and thus didn't affect Minisip. Once the RTP packet is sent on the local interface, the structure in which the parameters were stored is deleted. This chain of processing is repeated as long as there is one RTP packet left in the broadcast packet thanks to an infinite loop.

We tried with the part to explain what the principles of functioning of the downlink protocol are and how we tried to implement it. The source code is not perfect and needs to be improved in order to be more readable and more efficient. We will provide the source code if you like to give a look at it.

3 UPLINK

3.1 Overview

This part is about how the nodes will organize themselves to send their uplink messages (from the nodes to outside via the Access Point). All of them will follow a determined schedule and for this reason we will try to build up a protocol that doesn't need to exchange a lot of messages in order to avoid to disturb this schedule by contending with the certain node allowed to speak.

Before we started, several solutions came to us. We could have worked into different layers : IP, UDP or RTP layer. We needed to decide which level was the best, more efficient, more appropriate and not too complex to implement. First of all, sharing the media as we want to do must be a task for the MAC layer. Basically, it is what does the PCF mode of the wireless MAC layer (see background section). But we would like our process to run in a widely-deployed environment (so is the DCF mode) and to be able to take place with other applications that don't need (and certainly don't want) to take part of it. The ideal case would be to deploy the protocol in every Access Point and that it can adapt itself with the data traffic going through each given wireless area. Working in a given layer means that we are able to get the whole layer packets including header and payload and that we will certainly modify them and redirect them keeping the same basic structure.

Working in the IP layer could have been a great solution. As all the informations would have stayed in the original packets including the IP, UDP and RTP layer we would have had no problem to redirect them, no needs to store data anywhere else in every packet. But that implies a lot of work since you need to handle by yourself the IP packets bypassing all the IP layer process that the OS does usually. It is obviously feasible by using the RAW mode that the socket layer offers. But you need to analyze, build up, check up each packet in several entities depending on how many intermediate entities you have between the two user agents. That means a large amount of work only to handle the structure of the packet which is not our final purpose. However this can be a good solution for a next implementation that would like to focus on efficiency.

As we want to focus on a given application, the VoIP, it seems logical to get interested in working in the application layer. As we said previously the VoIP application comes with two application protocol, SIP and RTP. The first one establish the multimedia session between two entities and the second is in charge to transfer the voice over the network.

The UDP layer doesn't bring much help and it even complicates the task since we need to bypass the standard socket layer and get access to the network data into a lower state of the OS. However working in a higher level than UDP costs us to lose informations or rather to let the OS take care of them. That means that we need to open the right socket on the right computer every time that we want to get a packet (since we don't bypass the UDP or IP layer).

Though it seems to be the easiest way to implement it, that implies obvious difficulties. Let's take an example from the downlink as it is easier to understand. The Access Point aggregates all the incoming RTP packets and for this reason it needs to receive them first. So the packets must be sent to him and it needs to open a socket to get them. But if the IP header doesn't store the informations about the final destination anymore, we need then to store them somehow in the packet. Here's one

of the main issue : the socket layer make us change the IP header from the original one generated by minisip and it is now our job to take care of those informations.

A perfect case would have been to be able to intercept the packets on a computer that would have had neither the right IP destination nor the right UDP socket (mainly on the Access Point). One solution is to use the promiscuous mode of the network card as do all the packet traffic analyzer. But the implementation would have taken too much time since that is not an objective of our work and we do have more important goals to complete. That is why we decided not to use the promiscuous mode and then to manage the socket informations by our own. It is mostly a question of fitting into our initial goals and plans but it is needless to say that it would have been a better solution (see the 8th chapter).

3.2 Centralized list

3.2.1 The purpose of the list

In order to speak on time, every node only needs to know two things : which node is its previous neighbor and which node is speaking at the present time. The list will help the nodes to answer to the first question.

The list will store all the entities that take part of the shared speaking process at the present time in a given order. The list is duplicate the same way on all the nodes from the original one that the Access Point is in charge to keep up to date. And every twenty seconds the list is broadcast in the broadcast packet from the Access Point, the same one that includes all the incoming RTP packets.

As we said, we don't want to exchange new dedicated messages but as long as we can we would like to exploit the informations featuring in the packets that will be exchanged anyway. And in the list the entities are stored with unique keys that can identify a node with certainty. So we may use informations from the RTP packets to create the unique keys we need. And as we saw earlier, we modified minisip in order to add both the final IP address and UDP port to the RTP payload. And we know that this couple is inevitably unique since it refers to a single socket (UDP port) in a single computer (IP address). This couple seems to match exactly what we need.

The Access Point will need to manage the list as every node will do. So we wrote a library featuring about ten functions that will be deployed on every node and the Access Point. The functions will provide functions to deal with the list both on a node and on the Access Point. Indeed they don't have the same needs since the Access Point must keep the list up to date by pushing and popping new and old elements into and from the list whereas a node only needs to get the list from the broadcast packet and to locate his own position in it.

The purpose of the library is to provide high level functions that never change thus it doesn't matter how they are actually implemented.

Here is the *list.h* library file providing access to the list.

#list.h

```
#Use only on the AP
void list_push(list **start, int port, char *ip_address);
void list_pop(list **start, int port, char *ip_address);
```

```
int list_is_present(list *start, int port, char *ip_address);
char* list_to_string(list *start);
int list_count(list *start);

#Use only on the nodes
int list_is_first(list *start);
int list_is_prev_neigh(list *me, int port, char *ip_address);
void list_from_string(list **start, char *string);
list *list_find(list *start, int port, char *ip_address);

#Use on both
void list_create(list **start);
void list_delete(list **start);
```

The function names are for the most part very explicit and will see further throughout this chapter where and how the functions are called. We can already notice that as we said, the entities stored within the list are identified by both a port and an IP address (which has to be an unique couple).

We chose to implement it as a double linked list in order to find easily a element (since we can go through the list either way) and more than anything it is easy to add and remove an element at the end of the list. In a double linked list each element is linked with its previous and its next neighbor as each element keeps two pointers in its own structure.

Once again, how we programmed it doesn't matter as long as it matches with the function's signatures given in the *list.h*.

3.3 Multicast communication

Though the list will answer to the first question, we still need to deal with the second one : which node is speaking at the present time. When our previous neighbor starts to speak then we know that at the end of his speech time, which is actually the time it will take him to send the whole packet, that is going to be our turn to speak. A node that is speaking is in concrete term a node that is sending a RTP packet to its call partner. Normally such a RTP packet is sent from the first user agent in a unicast IP packet and will reach the second user agent without have been read by any other nodes since its destination is a single computer.

In a wireless network every packet can be seen by everyone since the media is shared like in a wired network that would have been plugged with a hub. But the networking process doesn't bring up to the higher layers all the packets that don't match with its own IP address. By enabling the promiscuous mode of the network card you can get access to all the packets that are listened over the media and thus get more packets than you should. And from this assumption you will be able to listen to the nodes that take part of the speaking process and so to know which one is talking at a

given time.

Unfortunately that involves some difficulties to develop it that don't worth it. And there is actually another solution to achieve what we want to do. We can also use the multicast communication mode. Indeed a good solution may be to modify the packets from the source. Instead of sending our unicast RTP packets that go directly from an user agent to another, we will send the RTP packets to everyone within the wireless cell. That way any node will listen to every RTP packet from any other node in the wireless area and will be able to figure out which one is speaking.

So every node that wants to take part of the process needs to open a multicast socket listening to the multicast packets sent over the wireless media. We use so the special multicast mode that limits the broadcast of the packets within the local network (i.e the wireless cell here). Each node set this mode by adding its membership to the multicast group represented by the local address 224.0.0.1. Sending a packet is even easier since we only need to send the packet to the IP address 224.0.0.1 and the predetermined port 1205 (which could have been anything else).

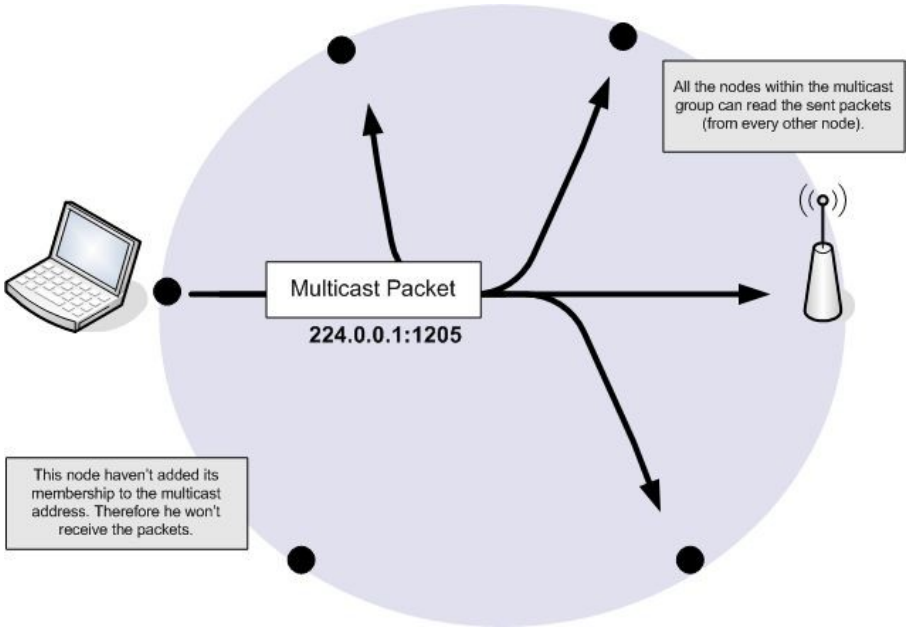

Figure 9 : multicast sending

That alternative is efficient as well but as we change the true destination of the RTP packet in order to replace it by the multicast destination, we need somehow to send it back towards its real destination. As minisip running on the inner nodes that will send the multicast packet we are then still able to modify the packet as we did before. What is more is that we use the same principle which consists in forcing minisip to add the real IP address and the real UDP port in the end of the RTP payload and then send the packet to the local proxy running on the same computer. The local proxy will simply change the IP header by sending the packet towards the multicast address in order to reach all the nodes that have added their membership to the group. At this time, the multicast packet over the media has got all the informations needed to redirect it to its real destination. Once again the Access Point will be in charge of this task.

As any other node in the wireless cell, the Access Point will subscribe to the multicast

group. Both because it needs to redirect the RTP multicast packets and because it will also need to listen to every multicast packet to keep the list up to date (which could have been done by two different entities). Then for every RTP packet it will receive on the multicast socket, it will build up a new IP header featuring the data it will find at the end of the RTP payload.

3.4 On the AP

As every node in the wireless cell, the Access Point creates a socket that is going to listen to the multicast packet over the shared media.

3.4.1 List management

From the Access Point's point of view every multicast packet incoming needs first to be analyzed. For each of them the Access Point calls the *list_is_present* function that returns 0 or 1 whether the nodes is already in the list or not. If the node not is not in the list then this means that it was its first RTP packet sent through the multicast socket and the Access Point will add it into the list. It will be put into the list by the *list_push* function of the list library. For the two functions the present node must be identify by the remote port and IP address is communicating with. These informations will be found as usual in the end of the RTP payload since they had been added by the modified minisip in the inner node.

If the node is already in the list, there is nothing to do on the list : it is already up-to-date. In both cases the packet will go through the rest of the program to be sent towards its final direction.

Then the list will be broadcast over the media into the same broadcast packet that carries the RTP packets coming into the wireless area (see the downlink part). The list is put into the packet as a string representing its content by calling the *list_to_string* function which returns a buffer filled with the string.

As we need to join the mechanism when we start a VoIP call, we will need to quit it when the voice call will end. And since we don't want to add new dedicated packets it must be another alternative. The most obvious thing that we can see when a voice call ends is that there is no RTP packet going out from the node to outside. So the Access Point can check whether the nodes use their speech turn or not. If one doesn't, it will be deleted from the list because that certainly means that it isn't going to use it anymore (see also the *error detection* section). If we are wrong, i.e if that was a single mistake of the nodes or whatever lost packet, the node can go back into the list by sending the next RTP packet he has got as a new node would do (and it is actually a *new* node since it is not in the list anymore).

This method is not implemented in our platform but that doesn't bother our test since we don't need it to experiment both downlink or uplink mechanism.

3.4.2 Multicast to unicast

The packet has been analyzed by the list management methods. So we already know what are the remote port and IP address of the right destination we need to send the packet to. Then it is as easy as opening a socket and sending the unicast packet to the final user agent. The Access Points acts as a gateway both between the multicast and unicast modes and between the local network and the outer networks.

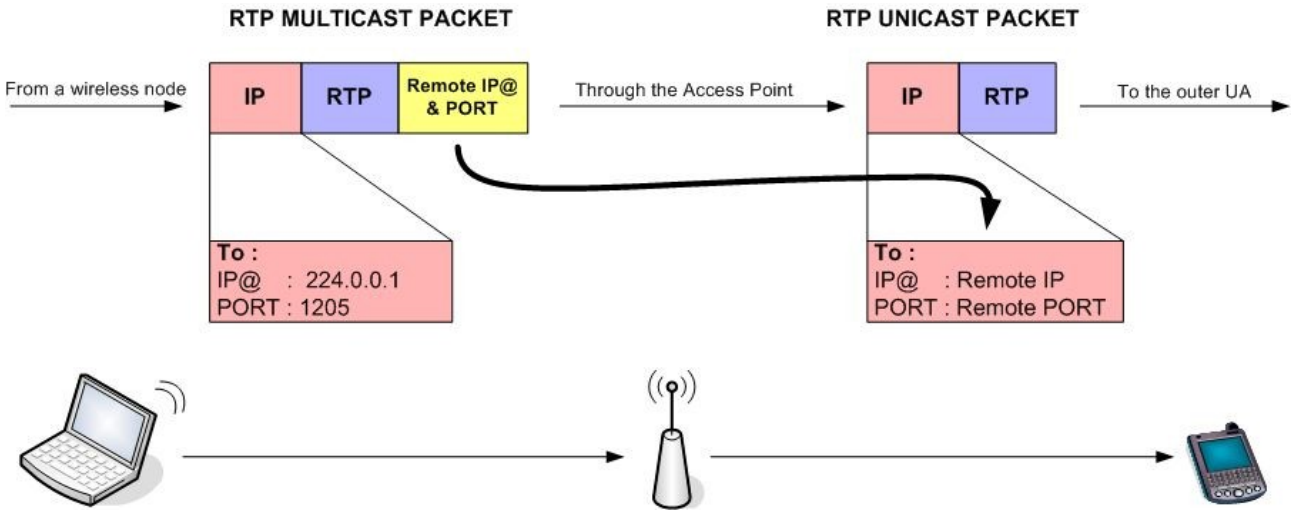

Figure 10 : multicast to unicast

3.5 On the inner nodes

As the main purpose is more about desynchronizing the nodes with each other, we don't need a very accurate process to manage the sending time on each node. The system we use is basically a boolean variable *my_turn* that will store either 0 or 1 whether it is our turn to speak or not. As several threads run in the same time, this variable is shared between the threads that need to update it (the ones in charge of the multicast and broadcast packets reception) and the one that need to check it in order to know whether the node is allowed to speak or not (the thread in charge to redirect the incoming RTP packet over the multicast group).

At the very beginning of the local proxy (that is how we call the program running on the inner node that is in charge to drive the node into a cooperative behavior), a local list is created by calling the *list_create* function featuring in the list library. This list will stay until the death of the program and will be updating regularly by different threads.

3.5.1 Reception

3.5.1.1 List management

You may start to know that about every twenty seconds a new broadcast packet will be sent from the Access Point over the wireless media. The packet structure is described in the 4.4.3 section (in the downlink chapter). The RTP packets featuring in that packet are not interesting since they are only part of the downlink process but the broadcast packet does bring something we are interested in. It brings us the last updated version of the list.

After have read first how big is the list's size, we are able to get from the packet a special string representing the list in a temporary buffer. Then we will make the list fit into the adapted structure already created by calling the *list_from_string* function. Thereafter the list can be handled by any function of the list library.

As soon as a node receive the list from the broadcast packet, it must check if it is the first one in it. Indeed that a special case since usually a node knows that it is allowed to speak if it hears its

previous neighbor. But if the node is the first one of the list then it doesn't have any previous neighbor. So it checks whether it is the first element of the list by calling the *list_is_first* function. This function returns a integer that tells if the node is the first one in the list and takes the element representing the node itself in parameter. In order to find that element the node calls the *list_find* function giving the remote port and the remote IP address used in the outer node that the call is established with. The inner node knows necessarily these informations since it is exchanging RTP packets with its call partner and so is able to reach the right outer user agent. To sum up, the call of *list_is_first(list_find())* with the right parameters will allow the node to know whether is the first node allowed to speak after the broadcast packet or not.

If the node is indeed the first to get the right to speak then the thread will set the *my_turn* variable to 0 and so any waiting RTP packets will be sent over.

3.5.1.2 Is it my turn ?

Even though the broadcast packet can change the value of my_turn that is not its main purpose and it is limited to one special case i.e if the node is just aware that it is the first one in the new broadcast list. Keeping the my_turn variable up-to-date is actually the purpose of another thread. Except for the first element of the list that we already handle, every node must wait for its previous neighbor to get a chance to access the shared media. This process creates a socket multicast to listen to all the outgoing RTP packets.

For every multicast packet it will receive through this socket it must check whether the sender is its previous neighbor. It can use the *list_is_prev_neigh* function that returns 0 if the sender is actually its previous neighbor or 0 if it is not. The node needs to give to the function the couple remote port and IP address of the multicast that it can get easily with others functions we implemented and its own position inside the list (to see where the present sender is located in relation to the node). The node can use the *list_find* function to get its own position and then give it to *list_is_prev_neigh* function we already saw.

Depending on what returns the *list_is_prev_neigh* function the variable may be updated. If it wasn't our speaking turn and our previous neighbor so it is now our turn to speak and we set *my_turn* to 0. If it wasn't our speaking turn and the multicast packet was not sent from our previous neighbor *my_turn* stay unchanged and it keeps its 1 value. If it was indeed our speaking turn and a multicast packet is coming it must be a mistake : either the sender can be wrong or we can be wrong. Anyway we set back my_turn to 1 to avoid contending to send a new packet whereas someone else would like to as well. We may have already sent our packet(s) before the mistake occurs and we will wait for our next turn (i.e in the next round) to speak again.

Thereafter the received multicast packet can be delete as soon as *my_turn* is updated since we are not interested in the data the contain. Let's notice that everyone could read every RTP within the wireless area without opening a new socket ! We will discuss this point in the 8th chapter.

3.5.2 Sending

3.5.2.1 Principle

As the background chapter explains it the contention is a wireless media is ruled by the CMSA/CA. There is no longer such things as collisions with the mechanism of RTS/CTS. But as there is more than enough space for everyone to talk we would like to avoid contention that can lead

to some lost packets. We won't bypass the RTS/CTS mechanism at all but we will organize the nodes in order to be sure as much as possible that the CTS will not be denied after the RTS request. That makes our mechanism both truly flexible and weak since it can be disturbed by any other node that wouldn't like to cooperate. Throughout this following chapter, you should always remember that a cooperate node must be allowed to talk and then will anyway request access to the wireless media with the RTS/CTS mechanism (MAC layer).

3.5.2.2 Unicast to multicast

We saw how the two first threads manage to keep the *my_turn* variable up-to-date. *My_turn* actually rules when the packets are sent from the sending nodes. But it unfortunately doesn't rule when the packets are generated.

Everything starts with the SIP user agent, minisip, in the inner node. It is in charge to record the voice from the computer microphone and encoding with the help of the chosen codec. Thus it can build a RTP packet by adding header around the voice encoded data. Basically we modified minisip to make it send the RTP packets to the local proxy, i.e through the localhost IP address (127.0.0.1) and a predetermined port which here is 1206. Furthermore after building the RTP packet and before sending it towards the local proxy, we modified minisip in order to add the real UDP port and IP address at the end of the RTP payload to be able to find later where to redirect the RTP packet.

Then the packet is received by the local proxy that is continuously listening to the socket waiting for every RTP packet from minisip. Its role is to send it over the shared media as a multicast packet but only when it is allowed to. The new multicast packet is so build and put into a buffer waiting for *my_turn* to be set to 0. As long as the value of *my_turn* is 1 the node will wait few micro seconds and then check again the value. As soon as the value of *my_turn* is set to 0, the multicast packet that was waiting in a buffer is sent over the shared media. Once the packet sent *my_turn* will be set back to 1 in order to avoid the node to keep speaking over the media while the other nodes are receiving the packet and so one of them will figure out that its turn just comes.

What implies this scheme is that only one RTP packet can be sent at a time and that is exactly our purpose. If there are more packets coming from minisip, they won't be lost but they will be handled by the socket layer and queued until the socket will be read again. And as the socket can't be read as long as the previous packet isn't sent, we are sure that the packets will be sent one at a time.

From the wireless network's point of view, there is no longer contention to access to the shared media. Every packet seems to be right on time, one after another. Each round will bring its new list (that doesn't have necessarily to change by the way) and then will restart all the process.

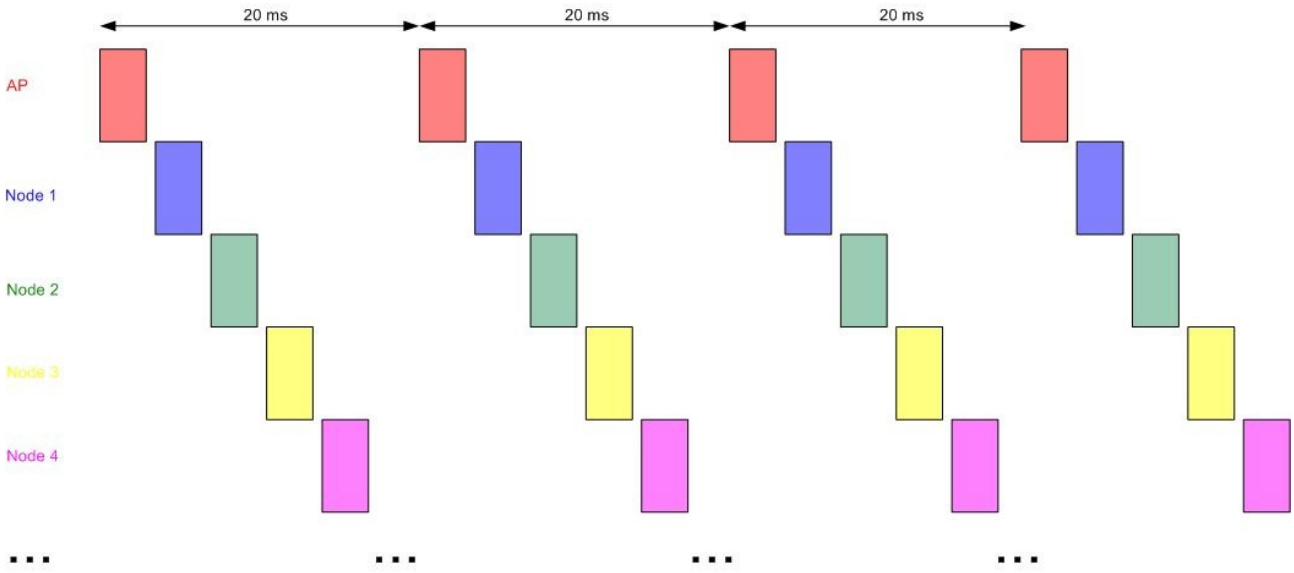

Figure 11 : pseudo synchronization

3.5.2.3 How everything starts

We mostly saw how all is going on when everything has already been started. But we will see now how the first nodes will get into the system. At the very beginning of the process there is no traffic over the share media but the Access Point that broadcast its regular packets. In all the following schematics the RTP part of the broadcast packet will be ignore since we are talking here about the uplink methods.

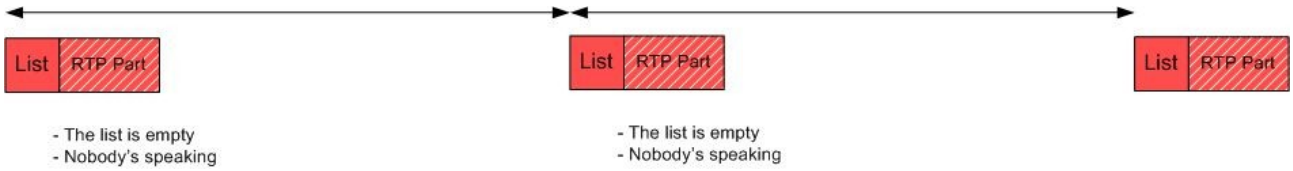

Figure 12 : first starting step

By now a node wants to send its RTP packets as it is beginning a new VoIP communication. As the local proxy has just been launched its *my_turn* variable is set to 0 and as soon as it is going t get a RTP packet coming from minisip it will send it and set back *my_turn* to 1. You should remind that thanks to the CMSA/CA, there can't be any collisions, only contention. So the new packet we want to send won't bother the broadcast one but at worst it will delay it a little bit. After sending its first RTP packet the first node will be automatically updated in the list managed by the Access Point and so in the next broadcast list. In the next round the node will be the first and the only node of the list so its turn will come just after the broadcast packet.

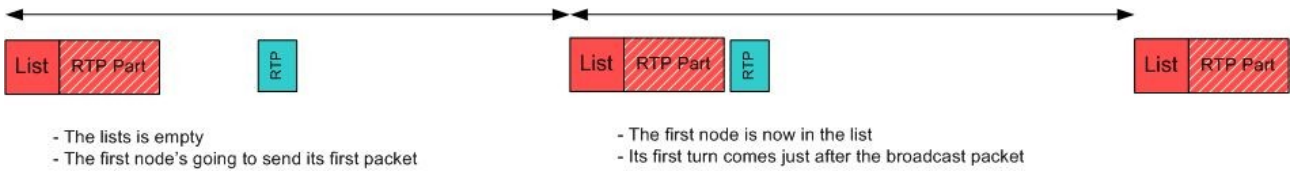

Figure 13 : second starting step

Then a second node wants to join the platform. Its my_turn variable is initially set to 0 and will take its turn as soon as the local proxy will get a RTP packet coming from minisip. Thus this new packet will slip in between the broadcast packet and the the RTP packet from the first node. Whether the packet will come before or after the packet from the first node doesn't matter since the first node knows that it is the first element in the list and it is ready to send. It is just waiting to get his RTS/CTS clear to send the packet over the media.

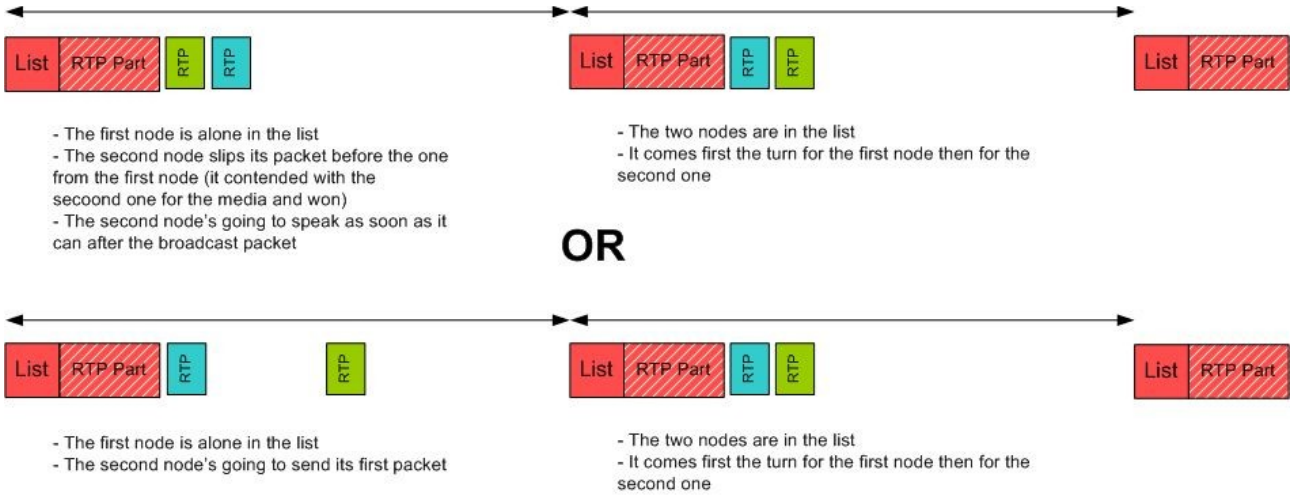

Figure 14 : third starting step

The new node has been added to the list and will occur in the next broadcast packet. Eventually the two nodes take part of the mechanism and it will be exactly the same for any other node that would like to join them.

We finally chose the solution using the RTS/CTS mechanism (or any contention mechanism used in the wireless cell) to add a node to the platform but here is another solution that it would be interesting to implement. There's not much work to do and that will improve the system since each new node will wait for being sure that no one want to talk anymore.

Here the first node waits for a first broadcast packet which gives it all the informations needed and it will start with a *my_turn* set to 1 (not allowed to talk). As the list from the broadcast packet is empty the result of the *list_is_first* function will be 0 which means that the node is actually the first element of the list. Needless to say that the node sends its RTP packet just after the broadcast one. It will thus be added into the list.

Then a second node wants to come into the platform. It first waits for a first broadcast packet and see that the list is not empty. As we want to cooperate even before sending its first RTP packet it would like not to contend the media with any node that has already taken part in the mechanism. So it will wait for the last element of the list to speak and, as it was its previous neighbor, will speak after it in an unallocated time that ends with the next broadcast packet. By sending its packet in this unallocated time, it will automatically be added in the list and take part normally in the round of the mechanism. Besides if there were several nodes trying to join the mechanism, the would contend with each other and would finally all join the cooperative mechanism.

3.5.2.4 Error detection

But what about a node that suddenly stops sending its RTP packets when its turn comes ? How will the system react ? In fact, that will crash all the system until the next broadcast packet. What is worst is if the first element in the list crashes then all the nodes will lost one packet of their voice call. The solution is to look further than our previous neighbor and check more generally if the nodes before had taken their turn.

Let's say that we know which node is our previous neighbor and we know also which node is the previous neighbor of our previous neighbor. We know also the size of the RTP packets since they have got all the same amount of data (172 octets) so we can estimate how long it will take at most to send it. Now the node that is located two elements before us is speaking and we start the a timer. If our previous neighbor does not speak before the end of the timer, we will consider that it isn't going to speak this round and we will set our *my_turn* variable to 0 in order to sent our waiting RTP packet.

As soon as any node speaks again the system will restart on its own (if of course there is no more nodes that skip their turn). And even if we lost time to restart it, because we need time to be aware of a mistake, it doesn't matter : that will not disturb the rest of process. Nevertheless that will protect the system against single mistakes that don't follow each other in a same round. If two following nodes are mistaken then the error detection method we just saw won't be enough to handle it. One alternative will be to expand the number of nodes that we check but anyway two consecutive errors in a same round will be very rare. As a matter of fact we haven't implement yet the error detection and that would be a great improvement that don't need a lot of work since we can mostly use functions from the list library to implement it.

4 HOW TO HANDLE THE NON-COOPERATIVE NODES

4.1 Designed to run smoothly with existing traffic

Here is one of the most powerful advantage of the solution we implemented. The process can take place in any wireless cell without interfering with the local and already running data traffic. We have seen how the protocol runs when there are only cooperative nodes within the cell. But everything runs still smoothly if there are nodes that have a non-VoIP traffic to send. Because the order set in the protocol is based upon who the previous neighbor is and because it still relies on the RTS/CTS mechanism, any nodes can slip an outgoing packet anywhere in between the scheduled VoIP packets. It is exactly the same mechanism as we use to insert the first packet of a new node that wants to cooperate.

That constitutes a great difference with the PCF mode in which the Access Point polls the stations and everyone within the wireless cell has to follow this, without being given a choice. But here we merely add a protocol over the existing DCF and as it has been designed for, both time-constrained traffic scheduled by our protocol and normal IP traffic can run in the same time over the wireless media. The wireless contention mechanism will rule what packet is sent over the media at a given time but this will not bother the order of the scheduled packet.

The risk is that if there is too much non-cooperating IP traffic over the wireless media, the scheduled packets might be delayed too much and thus miss the turn in the current round, which might mean loosing these packets in the voice conversation.

4.2 Multicast as a group management

One of the original issues was to know who is taking part of the mechanism e.g what nodes are cooperating together. Indeed in the downlink way we must aggregate only the RTP packets coming for a cooperating nodes. The multicast mode that we introduced for the uplink part in order to keep everyone up may be used as a group management. Listening at the multicast socket requires the station to register first and thus with the last releases of the multicast protocol we may be able to interrogate the multicast group in order to know for example how many nodes are sharing a given multicast group at a time.

That could improve both uplink and downlink protocol especially in the way we handle the error detection. Because we can know who is cooperating at any given time, we can handle easily a node that wants to stop participating. It needs to unsubscribe from the multicast group and the Access Point will know that the packets coming to and outgoing from it will no longer be handled as a part of the scheduled mechanism.

5 TESTS & PERFORMANCES

The tests which we performed here have two main goals. The first goal is to check that the modifications we added don't impair the VoIP communications by adding delays or increasing the jitter. The second goal is to compare the performance of WiFi in the case of a normal VoIP communications with the case of a modified VoIP communication.

We focused our tests on the downlink communications which are a one way communication from a User Agent outside of the cell to a user agent within the cell. The first three experiments aim to show that two nodes can communicate with their modified program. Then, the next ones underline the differences between a normal and a modified communication.

All the users agent belong to the same network. In order to represent a communication between an outer node and an inner node, the outer node uses an Ethernet and the inner node uses WLAN. Thus, communications within the cell , go through the access point.

The RTP parameters, such as the listening UDP port, are negotiated by Minisip and the SIP protocol. The programs which are running on the UAs do not set up the RTP communication.

Measurements are performed with Wireshark [10] which is a powerful network sniffer. It allows to measure parameters such as the delay or the jitter.

5.1 Dowlink

5.1.1 Scenario

In order to verify that our implementation of the downlink protocol works correctly, we set up a communication session between one outer node (User Agent A') and one inner node (User Agent A).

The UA-A' uses a modified Minisip. The destination address of the RTP packets is changed to go through the proxy. The remote port as well as the remote IP address are concatenated at the end of the RTP packets.

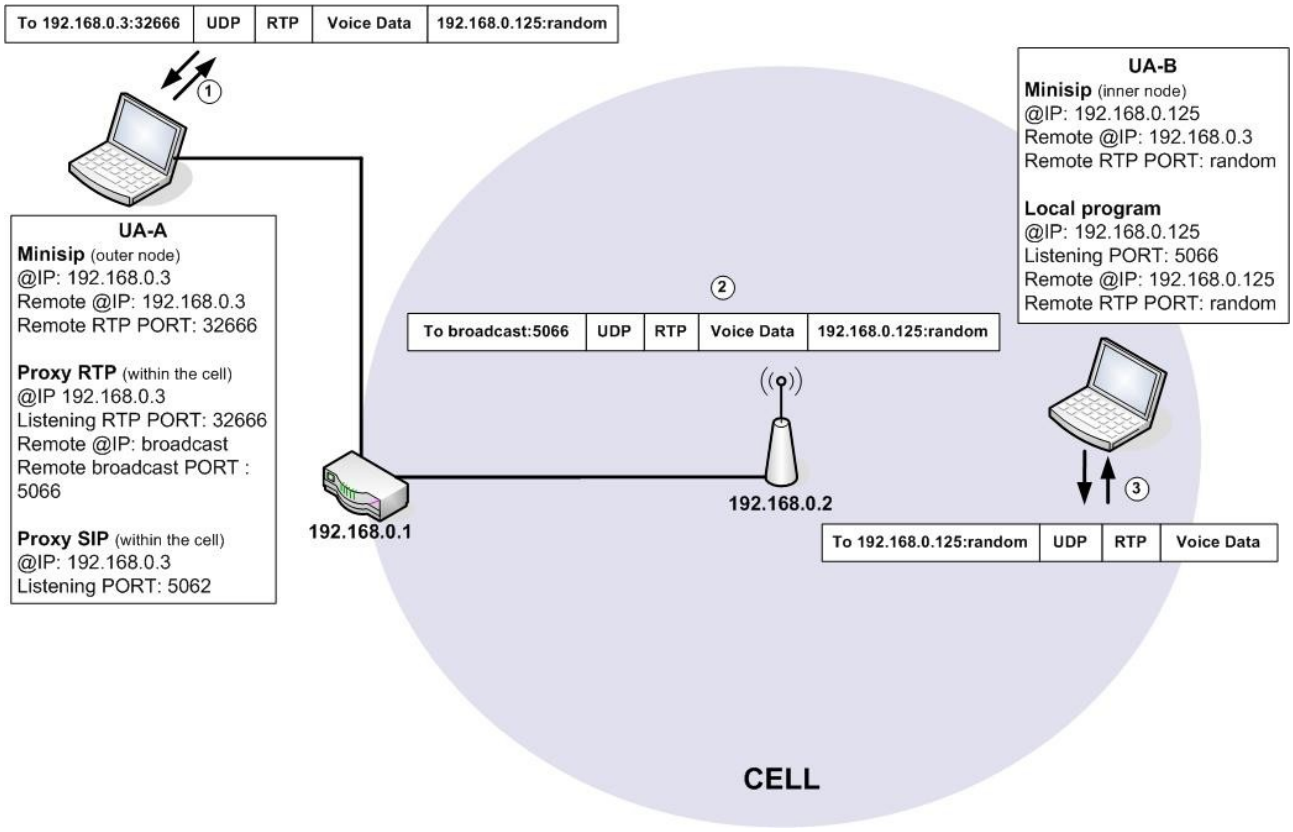

Figure 15 : Single downlink communication

The UA-A uses a normal Minisip listening on its WiFi interface and runs the program in charge of extracting the right VoIP packet (the packet which was intended for it).

The UA-A sends its packets to the proxy which aggregates them **(1)** and then sends a broadcast packet within the cell **(2)**. The UA-B receives this broadcast packet and extracts the RTP packet before forwarding it to Minisip on a local socket **(3)**. The UA-B sends its own packets to the UA-A (without of going through the proxy).

5.1.2 Results

In order to check that our implementation does not disturb a VoIP communication on the downlink, it is interesting to find out the differences between the RTP stream sent by the UA-A and the one received by the UA-B. We will focus on parameters such as jitter, delay, or the quality of the audio communication to validate the implementation.

One packet captured on the UA-A local interface has the following structure. We can see that we find the remote IP address and the remote port at the end of the packets.

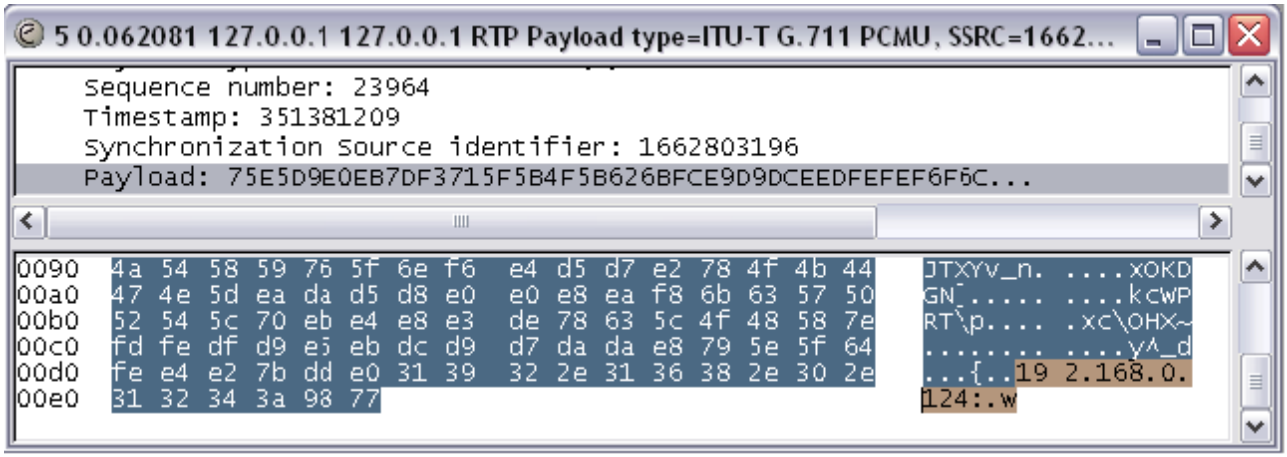

Figure 16 : Issued modified packet

We have the following RTP stream coming out of the UA-A on its local interface, thus it is not modified yet.

Src IP@	Src Port	Dest IP@	Dest Port	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter
127.0.0.1	32228	127.0.0.1	32666	G.711	568	0%	35.94	6.42	3.31

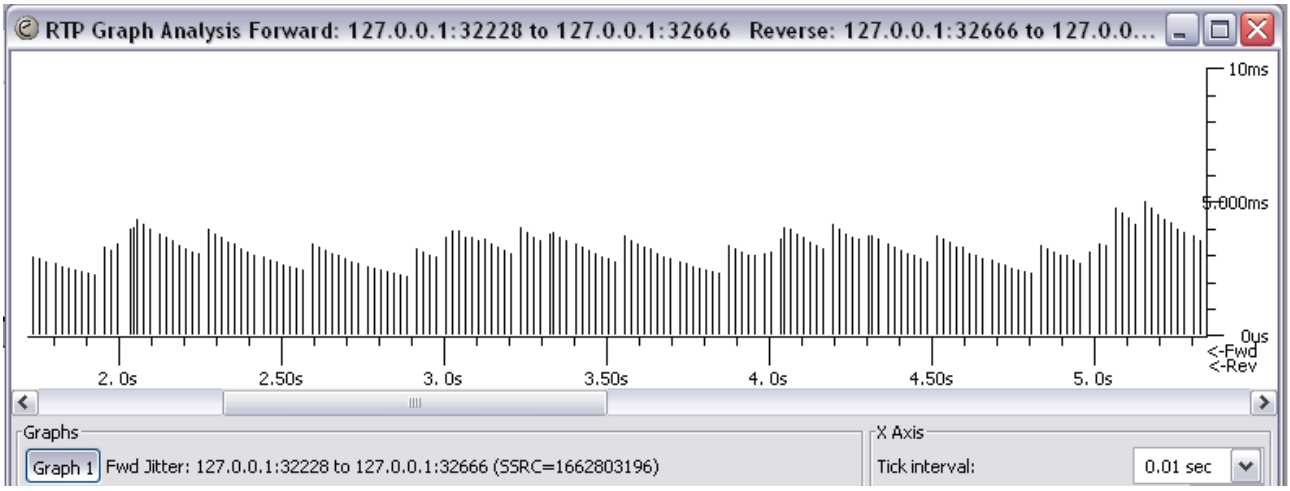

Figure 17 : Jitter of the RTP stream coming out of the UA-A

Now we will observe the RTP stream incoming to the UA-B after demultiplexing (on the local interface).

Src IP@	Src Port	Dest IP@	Dest Port	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)
192.168.0 .124	322790	192.168.0. 124	30004	G.711	843	**0.2 %**	45.21	4.42	2.97

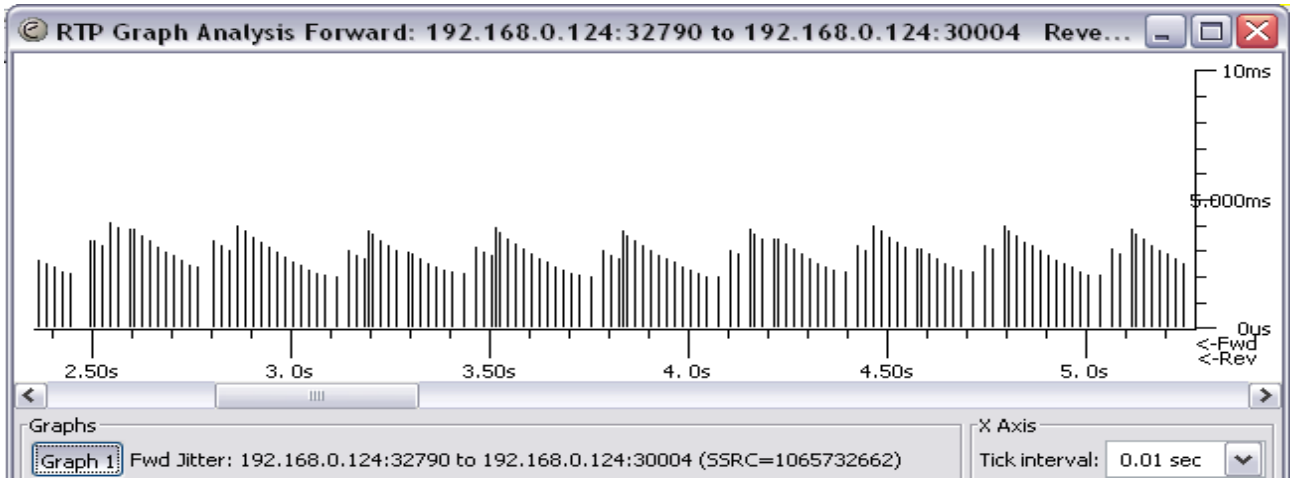

Figure X : Jitter of the RTP stream incoming on the UA-A local interface

We can see here that both RTP streams are very similar. For instance, the jitter follows the same pattern in both cases. However, we can notice that there is 0.2% lost packets. We think that these packets are lost due to the network since we tried to perform another capture and we observed 0% lost packets. The mean jitter in the second capture is very low, this shows that multiplexing and demultiplexing RTP stream has a very small effect on the communication. These results are confirmed by the audio quality of the communication. Indeed we were not able to notice any differences between a normal VoIP communication and this modified communication.

5.2 Uplink

5.2.1 Scenario

In order to verify that our implementation of the uplink protocol works correctly, we set up a communication between one outer node (User Agent A) and one inner node (User Agent B).

The UA-A uses a non-modified Minisip what implies that it sends directly its own RTP packet to the UA-B **(5)**.

The UA-B uses a modified Minisip which sends its RTP packets on a local socket to a program which add the needed delay before sending the packet. The remote IP address and the remote port are concatenated at the end of the issued packet.

The proxy is running and periodically sends a broadcast packet without of any RTP packet concatenated but only the list which determines the transmission order.

The program on UA-B listen to the broadcast packets and the multicast packets which are transmitted by the other nodes within the cell **(1)** and collects the RTP packet from the UA-B on a local socket **(2)**. If the node is the first one to talk in the list, it issues its RTP packet right after the broadcast packet for the local program. The local Otherwise, it listens to the multicast packets and waits its turn. In our case, there is only one node (UA-B), so when the UA-B receives a broadcast packet and since it is the first node to transmit in the cell, it sends the RTP packet by using a multicast address **(3)**. The RTP packet contains the remote PORT and IP address and the end. Then the proxy forwards the unicast packet to the UA-A after removing the added PORT and IP address **(4)**.

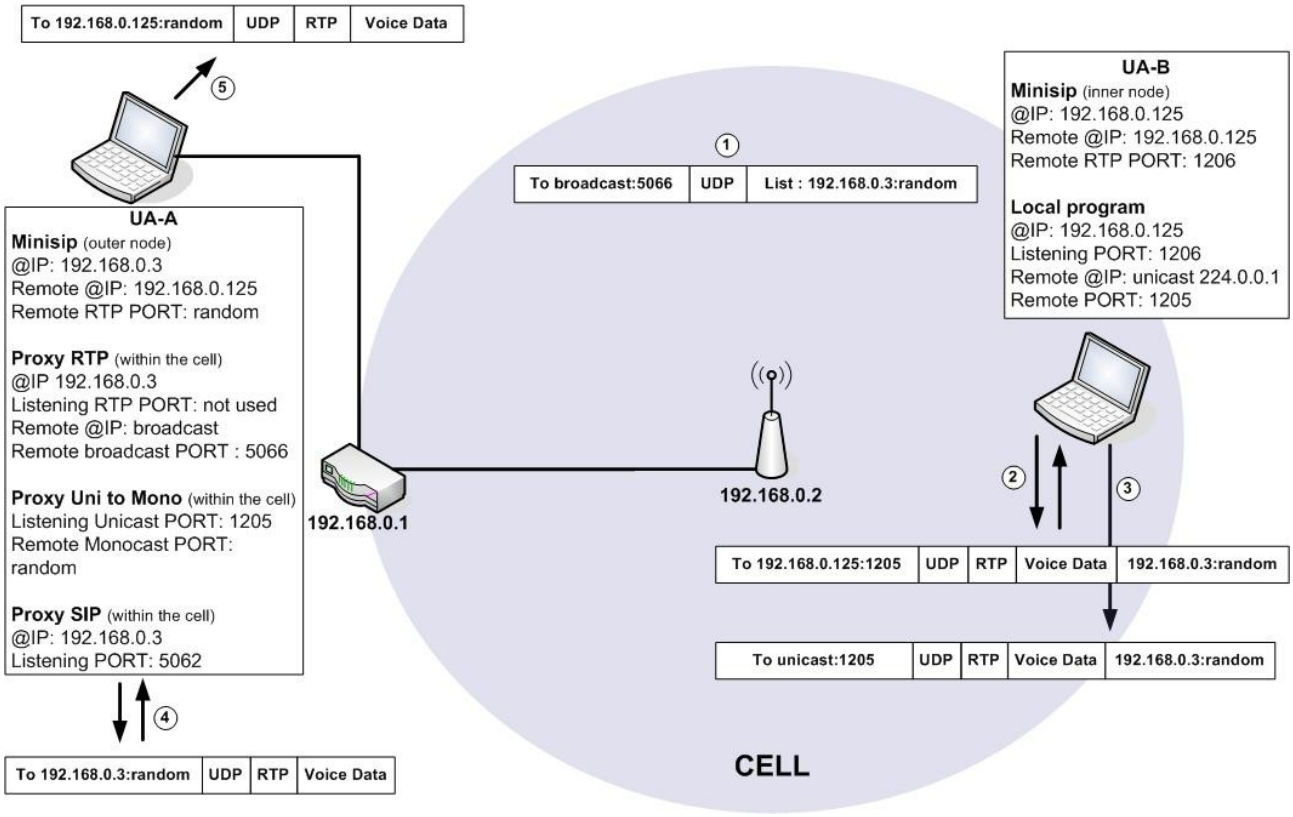

Figure 16 : Single uplink communication

5.2.2 Results

Here, it could be interesting to bring to the fore the differences between the RTP stream sent by the UA-B to the UA-A and the one received by the UA-A. It will show if the modified protocol on downlink affect the communications.
The RTP stream sent on the UA-B local interface has the same characteristics than the RTP stream sent by the UA-A in the previous test.

Src IP@	Src Port	Dest IP@	Dest Port	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)
127.0.0.1	33342	127.0.0.1	1206	G.711	1494	0%	39.83	6.78	3.77

If we observe the RTP stream incoming on the UA-A interface, we obtain the following results.

Src IP@	Src Port	Dest IP@	Dest Port	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)
127.0.0.1	32772	127.0.0.1	30774	G.711	475	**2**	**82.23**	7,69	4,58

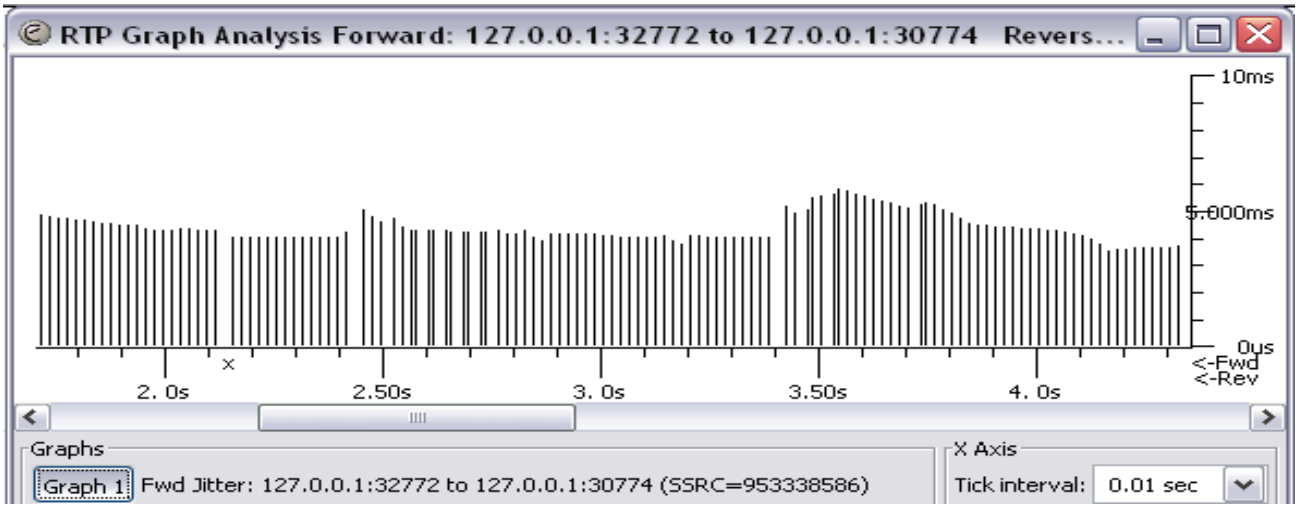

Figure 17 : Jitter of the RTP stream incoming on the UA-A local interface

In the same way than the previous experience, we can see that we have very similar results. However, we can notice that we have a maximum Delta relatively high what can be explained by the added delay at the beginning of the communication. Indeed, this high Delta takes place in the first second of the communication. We also notice that the pattern of the jitter is more regular but we were not able to find the reasons. Two RTP packets were lost during the communication. We think that it is due to the network. Finally, these results were also confirmed by the audio quality of the VoIP communication since we did not hear any cut or delay in the conversation.

5.3 Downlink and uplink

After having verified that the downlink and uplink worked well independently, this test aims to check the compatibility of both implementations.

5.3.1 Scenario

In order to check the complete new system of communication, the two previous tests are set up together. Thus we have in the same time two RTP streams: one on the downlink and one on the uplink.

5.3.2 Results

Here are the results that we obtained.

Src IP@	Src Port	Dest IP@	Dest Port	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)
127.0.0.1	32974	127.0.0.1	1206	G.711	787	0	36.46	5.93	2.93
192.168.0.124	32807	192.168.0.124	32974	G.711	781	**8**	55.25	9.3	3.98

Downlink Uplink

Figure 18 : Characteristics of the RTP stream incoming and out going from the UA-B local interface (inner node)

Src IP@	Src Port	Dest IP@	Dest Port	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)

127.0.0.1	32872	127.0.0.1	32666	G.711	398	0	35.99	4.58	2.92
127.0.0.1	32789	127.0.0.1	32872	G.711	379	2	56.55	8.22	4.76

Downlink Uplink

Figure 19 : Characteristics of the RTP stream incoming and out going from the UA-A local interface (outer node)

The results are similar to the two previous tests except the abnormally high amount of lost packets on the downlink within the cell. We tried to make other captures and we still have between 2 and 8 lost packet for 500 captured packets in average. That could be due to the demultiplexing task what would be inherent in our implementation. We will see later if that affects the communications in the case of a busy network.

5.4 Normal communication on downlink

This test aims to find a practical limit of the Access Point capacity in absorbing the normal VoIP communication flow.

5.4.1 Scenario

A single communication is set up between one outer node (User Agent A) and one inner node (User Agent B).

The UA-A and UA-B use a non-modified Minisip **(1 & 2)**. At the same time the network is stressed by a flow of fake RTP packets **(3)**. This flow is generated by the outer node by issuing UDP packets with a size of 172 B for the payload (the same constant size than the real RTP packets issued by Minisip). The inner node receives these packets and deletes them. A one way fake communication is represented by the send of a fake RTP packet every 20ms. The number of communications is increased regularly (every 1001ms to avoid synchronized transmission). Thus, the effects of the RTP flow on the real communication can be observed by measuring the delay and the jitter or by listening the quality of the communication. We start to send fake RTP packets after 10 seconds of normal communication.

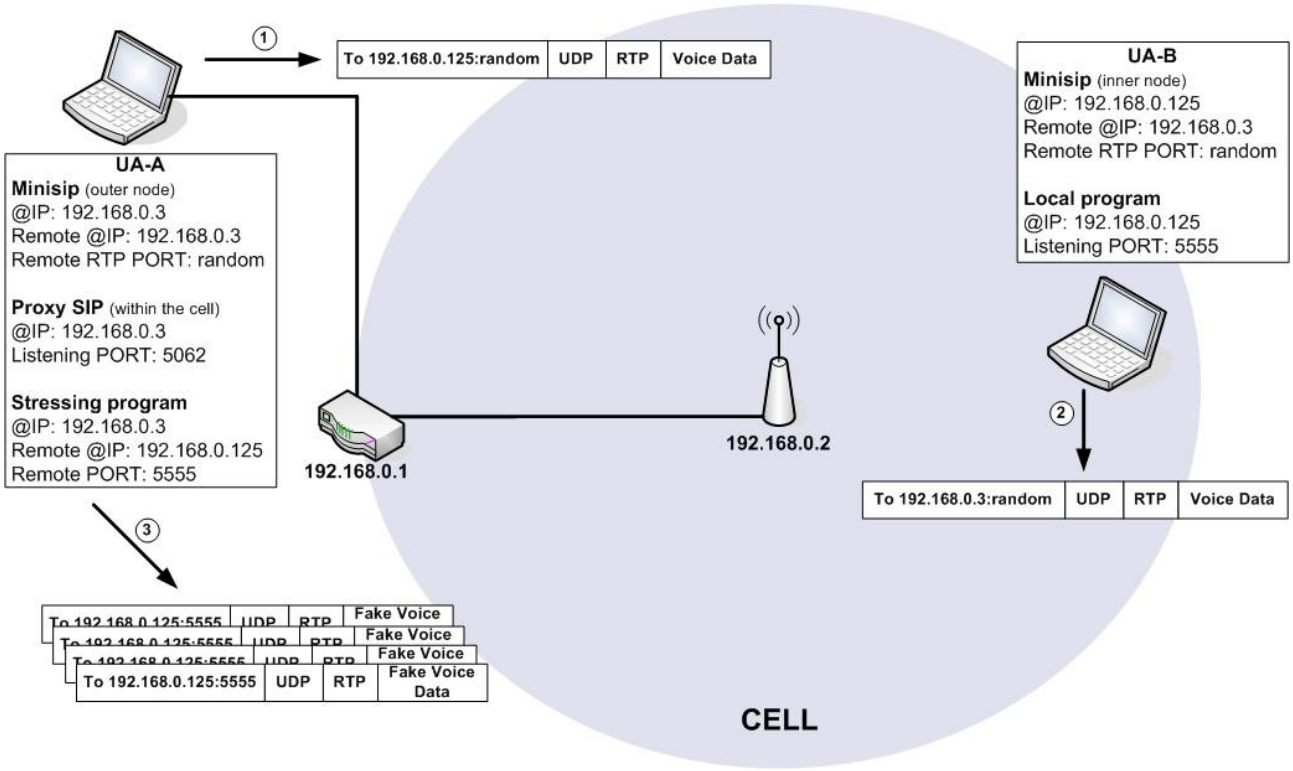

Figure 20 : Stressed downlink communication

5.4.2 Results

Observations of the inner node UA-B:

We have two RTP streams, one on the downlink and the other on the uplink. We will focus on the downlink. The following capture was done on the UA-B WiFi interface. Thus, the stressing program should only affect the downlink. Here are the results:

Src IP@	Src Port	Dest IP@	Dest Port	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)
192.168.0.124	34254	192.168.0.3	33222	G.711	3484	0	50.47	6.42	3.23
192.168.0.3	33222	192.168.0.124	34254	G.711	3172	**285**	**361.38**	44.07	5.04

Downlink Uplink

Figure 21 : Characteristics of the RTP stream incoming and out going from the UA-B WiFi interface (inner node)

As we said, we can see clearly that the RTP stream on downlink is disturbed by the stressing program since we have a lot of lost RTP packets. That is even more obvious by looking at the graph of the jitter. We have to notice that the fake communications are not taken into account in the measures. They are only here to stress the network and do not take part in the real communication.

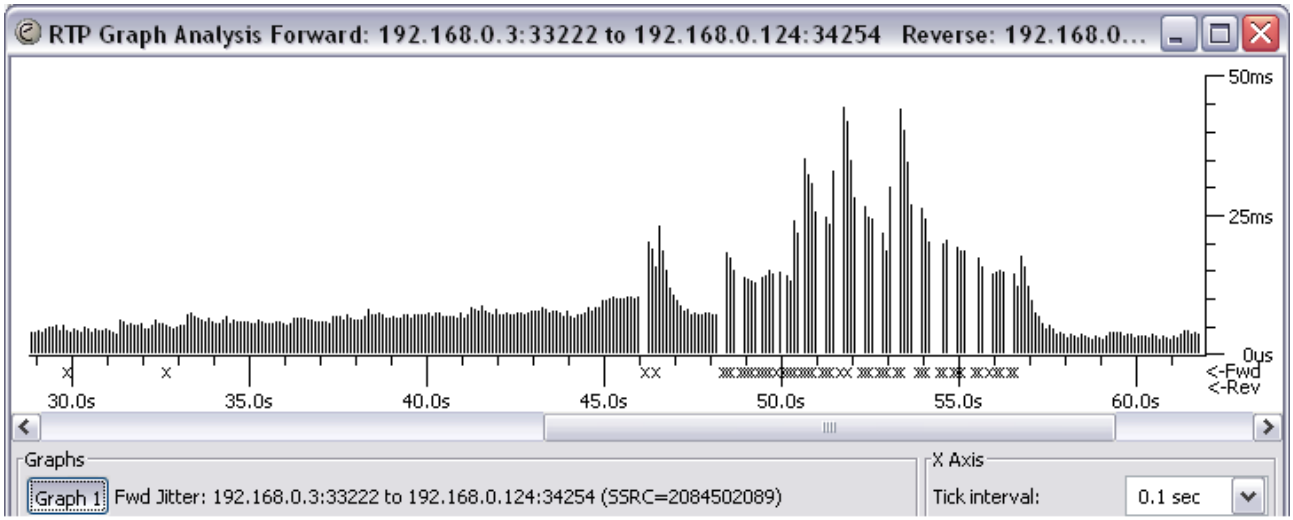

Figure 22 : Jitter of the RTP stream incoming on the UA-B WiFi interface

The graph shows that between 30 seconds and 45 seconds the jitter regularly increases. After 45 seconds there are the first lost RTP packet indicated by an "x". Then we see that the jitter is high right after the lost RTP packets and then becomes normal again. After 46 seconds the number of lost packets considerably increases as well as the jitter. We obtained a maximum between 52 and 54 seconds. Then we stop stressing the network what explains that the jitter becomes normal and that we don't have any more lost packets after 55cseconds.

The pikes on the graph and the lost RTP packets (x) correspond to cuts and delay in the audio communication. We had approximatively 35 one way fake communications at that time. Now, it will be interesting to compare those results with the ones which we obtained with the modified communication.

5.5 Modified communication on downlink

This test aims to find out the limits of the implementation of the new protocol for a down link communication and then compare the performances with a normal VoIP communication.

5.5.1 Scenario

A communication between one outer node (User Agent A) and one inner node (User Agent B) is set up.

The configuration settings of the UA-A and the UA-B are the same than the downlink test (i.e. 7.1. downlink). The UA-A uses a modified Minisip and the UA-B uses a non-modified Minisip. The RTP packets from the UA-A go through the proxy in order to be aggregated with the other RTP packets **(1)** (from other outer nodes). The UA-B sends directly its RTP packet to the UA-A. A stressing program sends fake RTP packet through the network. A one way fake VoIP communication is represented by the transmission of a RTP packet every 20ms.These packets have exactly the same size than the packets which are sent by the UA-A, namely 188 B **(2)**. Then the fake RTP packets are aggregated with the real packets by the proxy. Every 20ms, the proxy sends a broadcast packet which contains the aggregated RTP packets **(3)**. The local program on the UA-B receives the broadcast packet, extracts the RTP packets from it and forwards them to the right destination **(4)**. In our case, there are two destinations : the UA-B and a local program which consumes the fake RTP packets. The number of communications is increased regularly (every 1001ms to avoid synchronized transmission). We start to send fake RTP packets after 10 seconds of normal communication.

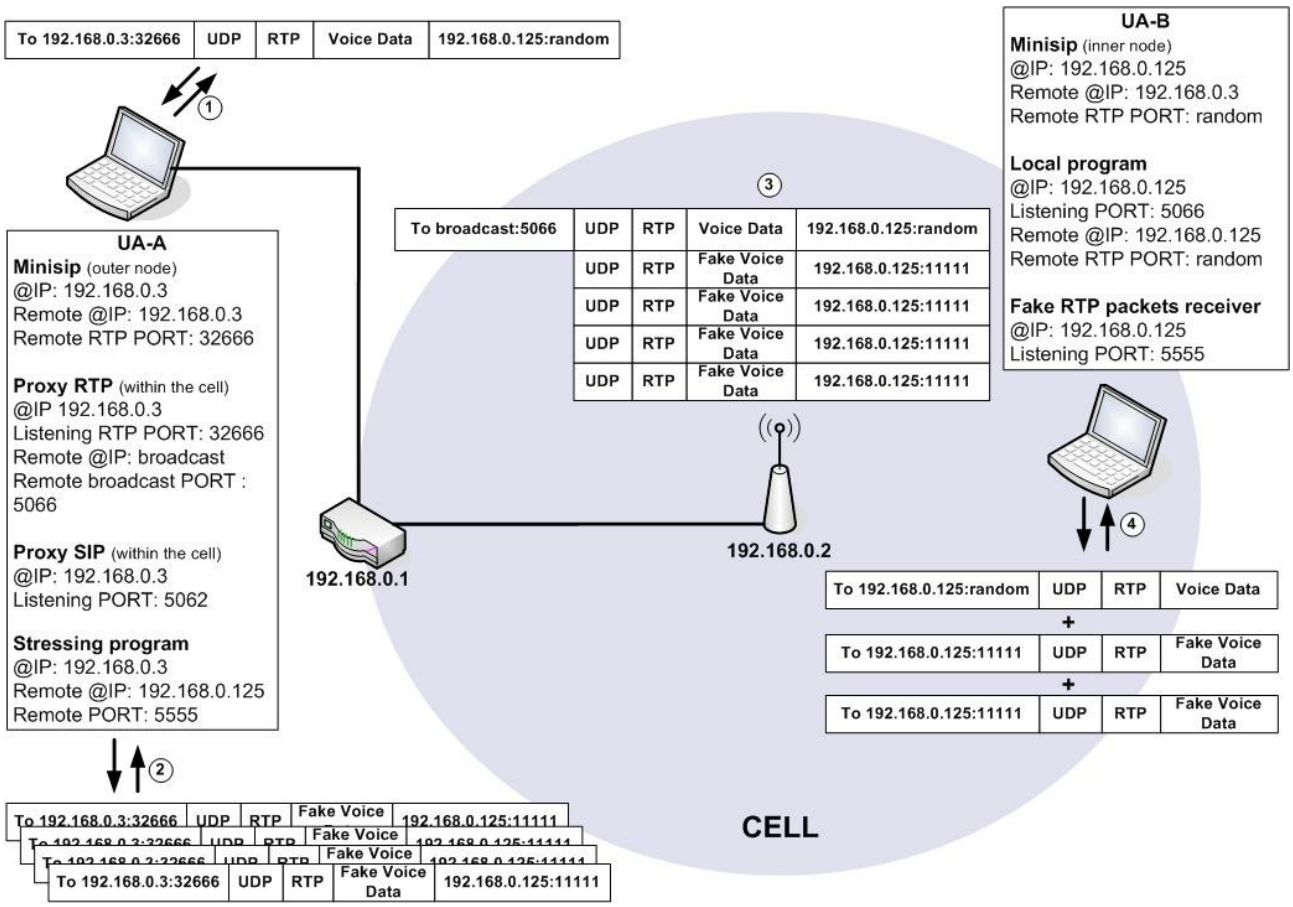

Figure 23 : Stressed modified downlink communication

5.5.2 Results

Here we have a modified communication to which we added several fake RTP streams. As we saw previously, only the downlink should be affect by the stressing program. We obtained the following results.

Src IP@	Src Port	Dest IP@	Dest Port	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)
192.168.0.124	32815	192.168.0.124	30254	G.711	2998	**560**	**397.08**	15.81	8.81

The first observation that we can do is that there are a lot of lost RTP packets. Indeed, we have the double of lost packets than a normal communication for approximatively the same amount of captured packets (560 against 285 for ~3000 packets). Nevertheless, the other parameters such as the mean jitter or the max jitter are similar.

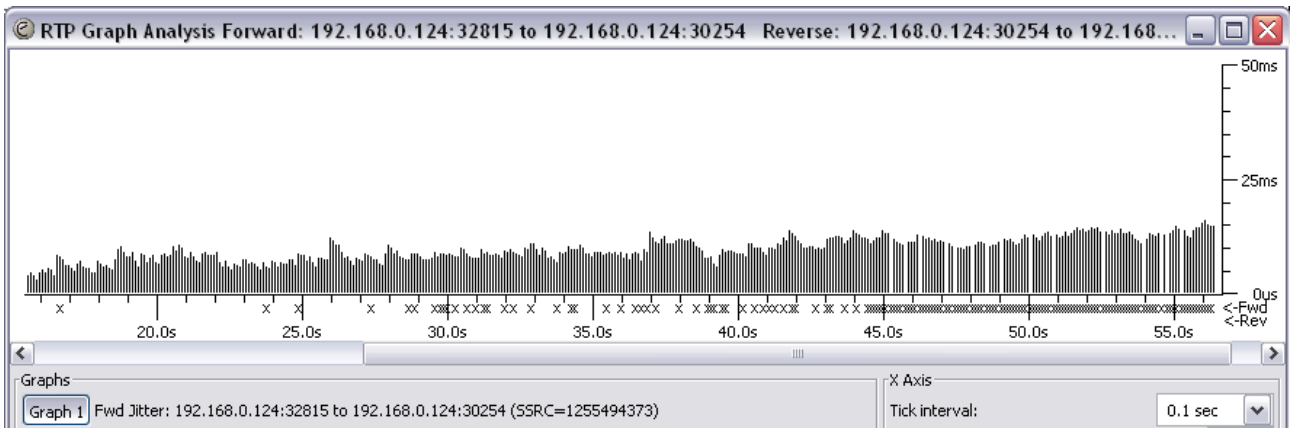

Figure 24 : Jitter of the RTP stream incoming on the UA-B WiFi interface

By looking at the graph, we can see that the jitter increases regularly between 15 seconds and 56 seconds. We don't have any pikes as we could have with the previous test. However, the firsts lost RTP packets appear earlier (about 30 seconds contrary to 48 for a normal communication). The jitter never exceeds 15 ms. We were able to make the same conclusions by listening the audio quality of the communication. Indeed, we start to have big cuts and delay at about 40 seconds what was surprising since we had lost RTP packet at about 30 seconds. We had approximatively 28 one way fake communications at that time what is less than the previous experience.

5.6 Conclusion

Now that we saw the results of the tests and we underlined the important facts, we will try to find some hypothesis that could explains those differences between a normal and a modified communication. The important number of lost RTP packets might be due to several parameters.

The first and most likely hypothesis is that our program in charge of demultiplexing is not able to absorb a heavy traffic and then not able to forward the RTP packets. That can explain the loss of an important number of RTP packets. From another point view, the problem can take place before sending the Broadcast packet in which the RTP packets are multiplexed. Indeed, we run on the same computer, the RTP proxy (in charge of buffering and multiplexing), the user agent (which were making the calls), the SIP proxy and the stressing program. It appeared that after a few seconds when we were stressing the network, the load of the processor was relatively high, then we wonder if our computers were powerful enough to achieve such a task. We can make the same remark for the remote user agent on which the program in charge of receiving the fake RTP packets was running and was using a lot of hardware resource. Then transmission of th RTP packets after demultiplexing was not fast enough.

The second and less likely hypothesis is that the wireless network is less efficient in managing large packets than small packets. It could explain the loss of the RTP packet with a heavy loaded wireless network. Then comes the problem of the packets fragmentation. Indeed, since we multiplex several RTP packets (from several RTP stream) within a same larger packet, the size of this new packet can exceed the MTU. Considering a MTU of 1500 Bytes, the maximum payload of a UDP packet is 1472 : MTU(1500) - IP header(20) - UDP header(8). Our modified RTP packets

have a size of 188 Bytes, then we are able to multiplex seven communications without fragmentation. We have to wonder if it is still interesting to multiplex the RTP streams in the case of more than seven communications (with fragmentation) or if we can issue a large packet which size is bigger than the MTU. We can also use the option “don't fragment” in the IP header but that could have an effect on the performance of the wireless network. Finally, the best solution could be to group the RTP streams together by seven and then issue each packet on the network.

6 Possible improvements

6.1 Proxy SIP

The implementation of our protocol on the downlink requires that the SIP User Agent outside of the wireless cell is modified and adds additional informations at the end of the RTP packet (the remote IP address and the remote port). We can see that this a constraint for this user agent since it has to be modified in order to make possible the use of a protocol in which it is not involved.

In order to avoid this problem, it could be better to modify the RTP packets at the same time they are aggregated (on the access point or on the proxy). Several solution can be implemented. We will focus on the solution which seems the most appropriated.

We can use the SIP protocol and a SIP proxy to make this solution possible. In order to explain the solution, we will take an example of a VoIP session between a SIP User Agent A (UA-A) and a User Agent A' (UA-A'). The UA-A initiates the VoIP session and the UA-A' answers with a OK.

The UA-A knows that it can use the multiplexing/demultiplexing process since it has the local program running. Then it would like that the incoming RTP packet goes through the proxy so that they can be aggregated and broadcast. The UA-A in the wireless cell sends a SIP REQUEST to the UA-A'. The needed informations to set up the VoIP session are written in this SIP packet, especially in the SDP part. The RTP port, the codec or the source IP address are very important. The SIP proxy receives the SIP packet and analyses it. Then the proxy stores the source IP address, the recipient IP address and the RTP port in its memory. Once the the RTP port and the addresses are stored, the proxy replaces the source IP address by its own address and the RTP port by its own RTP port in the SDP packet. Then, the SIP packet is forwarded to the UA-A'. The UA-A' answers to the request with a SIP packet which specifies the RTP parameters such as RTP port or the codec. The proxy directly forwards this packet to the UA-A.

Now the proxy has all the needed informations about the VoIP session in order to be able to intercept the RTP packet from the UA-A', aggregate them and broadcast them. The parameters are the proxy RTP port, the UA-A RTP port, the proxy IP address, the UA-A IP address and the UA-A' IP address. Indeed, since we change the RTP parameters in the SIP REQUEST packet, the UA-A' will send its RTP packets to the proxy instead of the UA-A. Then, the proxy will add the UA-A IP address and the UA-A RTP port at the end of the packet before broadcasting it, so that the local program on the node within the wireless cell will be able to forward the right RTP packet to the right destination.

We can notice that this solution is close to the main goal of the stun protocol.

6.2 SecureRTP

Since the new mechanism we have settled makes everyone sends its packets to everyone else within the cell by using the multicast socket, everyone is so able to read all the voice calls going on over the media. By receiving the RTP packets, modified or not, of a VoIP session is enough to recomposed the voice and then listen to the outgoing conversation. Besides the other side of the communication is broadcast from the Access Point to everyone so that anyone can build any incoming voice. The solution would be to use SRTP which allows to encrypt the content of the RTP

packets, i.e the encoded voice, so that only the real receiver can decrypt it. And as we implemented our protocol, it turns that it doesn't need a lot modifications to use SRTP instead of RTP since SRTP doesn't change the header but only encrypt the data of a RTP packet in a secure way.

7 CONCLUSIONS

We designed this protocol and implemented a prototype in order to see how it behaves in real situation and to understand whether the assumptions were good and whether the expected results were reachable. As a matter of fact the results shows that it is worth to keep working and digging in this area.

The obtained results presented in this report must be balanced between the implementation and the protocol itself. What the results show is that the protocol we proposed has got a great strength which is to control and contain the jitter even when the number of VoIP sessions is increasing. Whereas the prototype shows some weakness especially in the way to handle the buffering process. As the jitter is the most sensible and the main parameter in a VoIP communication this is a very good issue. We can enable more communications in the same time in a single wireless cell, we can ensure a better quality by containing the jitter, and last but not least we don't stop the other kind of traffic that need also the bandwidth.

If the results are confirmed by larger tests and with the improvements that are possible in the future such a protocol may become part of a bigger context-aware architecture and for instance could be enable when enough nodes within the same wireless cell can take advantage of it (or preset their willing to use this service) : the router and wireless point will switch between different modes according to the users' preferences.

Finally, this shows as well that we can both fairness and high performances. In our case, the cooperative behavior of the nodes is the basis of everything and when common interests are at stake it is better to gather energies and work all together.

8 REFERENCES

[1] www.minisip.org (main page, download page, contact page) - last access 07/01/06
[2] SIP, http://www.cs.columbia.edu/sip/ - last access 06/12/03
[3] Ubuntu, www.ubuntu.com – last access 07/02/02
[4] iPAQh5550,
http://h20000.www2.hp.com/bizsupport/TechSupport/Home.jsp?&lang=en&cc=us&prodTypeId=215348&prodSeriesId=322916&lang=en&cc=us – last access 07/01/06
[5] Minisip build page, http://www.minisip.org/develop_build.html – last access 07/01/06
[6] http://www.openser.org (download page, documentation page) - last access the 070106
[7] SJPone, http://www.sjlabs.com/sjp.html - last access the 070106
[8] http://web.it.kth.se/~it02_mka/exjobb/INSTALL.ubuntu – last access 07/01/06
[9] RFC 3551, RTP Profile for Audio and Video Conferences with Minimal Control
[10]www.wireshark.org (main page, download page) - last access the 070106

9 APPENDICES

9.1 Mediastream.cxx

[...]

```
#include <config.h>

#include<libminisip/mediahandler/MediaStream.h>

#include<libmikey/MikeyPayloadSP.h>
#include<libmikey/keyagreement.h>
#include<libminisip/sdp/SdpHeaderM.h>
#include<libminisip/sdp/SdpHeaderA.h>
#include<libminisip/sdp/SdpPacket.h>
#include<libmnetutil/UDPSocket.h>

/* added by g collin */
#include<libmnetutil/IPAddress.h>
#include<libmnetutil/IP4Address.h>
/* end */

#include<libmutil/stringutils.h>
#include<libmutil/Timestamp.h>
#include<libminisip/mediahandler/Media.h>
#include<libminisip/mediahandler/RtpReceiver.h>
#include<libminisip/codecs/Codec.h>
#include<libminisip/ipprovider/IpProvider.h>
#include<iostream>
```

[...]

```
      /* added by g collin */
      memcpy(dataIpPort,data,length);
      memcpy(dataIpPort+length,remoteAddress->getString().c_str(),
remoteAddress->getString().size());
      memcpy(dataIpPort+length+remoteAddress->getString().size(),":",1);
      memcpy(dataIpPort+length+remoteAddress->getString().size()+1,
&remotePort,sizeof(remotePort));
      uint32_t             lengthIpPort=length+remoteAddress->getString().size()
+sizeof(remotePort)+1;
      packet = new SRtpPacket( dataIpPort, lengthIpPort, seqNo++, lastTs, ssrc )
;
      //packet = new SRtpPacket( data, length, seqNo++, lastTs, ssrc );

      if( dtmf ){
            packet->getHeader().setPayloadType( 101 );
      }
      else{
            if( payloadType != 255 )
                  packet->getHeader().setPayloadType( payloadType );
            else
                  packet->getHeader().setPayloadType(             selectedCodec-
>getSdpMediaType() );
      }

      if( marker ){
            packet->getHeader().setMarker( marker );
      }
```

```
	packet->protect( getCryptoContext( ssrc, seqNo - 1 ) );

	/* added by g collin */
	proxyAddress = new IP4Address("127.0.0.1");
	/* end */
	if( remoteAddress->getAddressFamily() == AF_INET && senderSock )
		//packet->sendTo( **senderSock, **remoteAddress, remotePort );
		packet->sendTo( **senderSock, **proxyAddress, 1206);
	else if( remoteAddress->getAddressFamily() == AF_INET6 && sender6Sock )
		packet->sendTo( **sender6Sock, **remoteAddress, remotePort );

	delete packet;
	senderLock.unlock();
```

[...]

9.2 Minisip.conf

```
<version>
	2
</version>
<network_interface>
	eth1
</network_interface>
<account>
	<account_name>
		My account
	</account_name>
	<sip_uri>
		bchazalet@192.162.0.124
	</sip_uri>
	<proxy_addr>
		192.168.0.3
	</proxy_addr>
	<register>
		no
	</register>
	<proxy_port>
		5062
	</proxy_port>
	<proxy_username>
		user
	</proxy_username>
	<proxy_password>
		password
	</proxy_password>
	<pstn_account>
		no
	</pstn_account>
	<default_account>
		yes
	</default_account>
	<secured>
		no
	</secured>
```

```
        <ka_type>
                psk
        </ka_type>
        <psk>
                Unspecified PSK
        </psk>
        <certificate>
        </certificate>
        <private_key>
        </private_key>
        <ca_file>
        </ca_file>
        <dh_enabled>
                no
        </dh_enabled>
        <psk_enabled>
                no
        </psk_enabled>
        <check_cert>
                yes
        </check_cert>
</account>
<tcp_server>
        yes
</tcp_server>
<tls_server>
        no
</tls_server>
<secured>
        no
</secured>
<ka_type>
        psk
</ka_type>
<psk>
        Unspecified PSK
</psk>
<certificate>
</certificate>
<private_key>
</private_key>
<ca_file>
</ca_file>
<dh_enabled>
        no
</dh_enabled>
<psk_enabled>
        no
</psk_enabled>
<check_cert>
        yes
</check_cert>
```

<local_udp_port>
5060
</local_udp_port>
<local_tcp_port>
5060
</local_tcp_port>
<local_tls_port>
5061
</local_tls_port>
<sound_device>
/dev/dsp
</sound_device>
<mixer_type>
spatial
</mixer_type>
<codec>
G.711
</codec>
<phonebook>
file:///home/bchazalet/.minisip.addr
</phonebook>