AALBORG UNIVERSITY

SOURCE AND NETWORK CODING ON MOBILE DEVICES

Aalborg University
Department of Electronic Systems

Project Group 992, 2009

Aalborg University

Institute of Electronic Systems

Fredrik Bajers Vej 7 ▪ DK-9220 Aalborg Ost Phone +45 96 35 87 00

Title: Source and Network Coding on Mobile Devices

Theme: Mobile Communications

Project period: September 1th 2008 to January 12th 2009

Project group:

992

Group members:

Javier Lopez Rubio

Idoia Longan Zarzoso

Supervisor:

Frank H.P. Fitzek

Pages:

72

Finished:

12th January 2009

Abstract: This project concerns the development and implementation of a full functional demonstrator for cooperative wireless networking and network coding, which is a technology for improving performance in wireless networks. An application for distribute a video among nodes forming a cooperative cluster is implemented. The implementation is done for the Maemo platform.

In Cellular Controlled Peer-to-Peer (CCP2P) networks, besides being connected to an *outside world* using cellular links, a group of mobile devices in close proximity form a cooperative cluster contributing their onboard capabilities and resources to exploit them a more efficient way. This project implements this kind of networks. Diverse cooperating phones agree on splitting a video to download and start to receive it through USB from a server (simulating the cellular link). Simultaneously the received data is exchanged over the short-range link using three different transmission schemes: Network coding and broadcast, which were implemented in a previous work, and a new hybrid schema developed as a combination of both, broadcast and network coding.

After implementation of the program, tests are carried out, to see the results achieved with this new schema and compare it with the previous ones, in terms of throughput, energy consumption and necessary time to distribute the whole video. The results obtained show an improvement in the three cases for the new hybrid schema, which use broadcast at the beginning, when all the nodes are interested in all the packets, and Network Coding for retransmissions of packets.

Preface

This report documents the work of group 992 in the 10th semester on the Mobile Communications line at the Institute of Electronic Systems, Aalborg University. The work was done in the period from September 1st 2008 to January 12th 2009.

In the report chapters are numbered using forth running Arabic numbers. Appendices are labeled by forth running Arabic letters. In both chapters and appendices subdivisions into sections are labeled in two levels. The first part describes either the chapter number or the appendix letter and the second part states the section number. A two level labeling is also used for figures, tables, and equations. As examples, "figure 2.3" and "table 2.4" refers to the third figure and the fourth table in chapter two respectively. The literature used to support this report is listed between the main report and the appendices in alphabetical order. References to for example the second literature entry is done as [2].

Javier Lopez Rubio

Idoia Longan Zarzoso

Happy Reading!

08gr992

Aalborg, January 12th 2009

Contents

Chapter 1

Introduction

The utility of mobile devices is directly impacted by their operating lifetime before batteries need to be recharged. With the convergence of new computing, communication and entertainment applications on wireless handsets, power demands are increasing rapidly, presenting an increasingly technical challenge to manufacturers to deliver high performance without dramatically increasing energy consumption. Functionalities and capabilities like advanced imaging features (camera, high-definition display, etc.) as well as versatile and high-performance wireless connectivity including short-range communication (Bluetooth, WLAN, etc.), and higher data rates over the cellular interface to support new services, lead to manufacturers face a serious problem as the mobile devices are using more and more energy.

This has a main problem: the mobile device's active usage time decreases and users continue to expect faster operation, greater functionality and long operative time from their computing devices. A solution for the problem of increasingly more complex mobile devices on one hand, and the need for supporting higher data rates over air interfaces on cellular communications was described in [7].

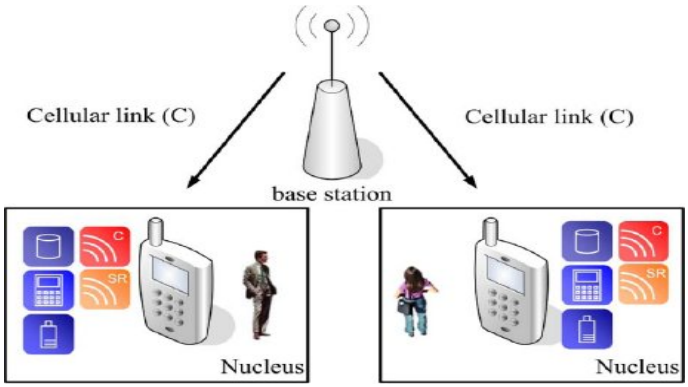


Figure 1.1: Conventional Cellular Communication Architecture.

In this contribution, it is advocated a novel paradigm which tightly and dynamically combines centralized networks (e.g., cellular) with distributed networks (e.g., ad-hoc, short-range, peer-to-peer) [8].

The current architecture of cellular communication systems is characterized by the communication between mobile devices and a base station, as the only access point to the network services. Figure 1.1 shows it (Figure taken from [7]).The idea proposed in [7] to solve the aforementioned drawbacks, is to enable directly communication among several mobile devices forming wireless grids, thus mobile devices can cooperate with each other with their different capabilities, opening a wide range of possibilities for enhance performance sharing functionalities, using resources better, etc. The wireless grid uses the short range communication technology to communicate among the devices and, as given in Figure 1.2, the cellular link is used to connect to the cellular system (Figure taken from [7]).

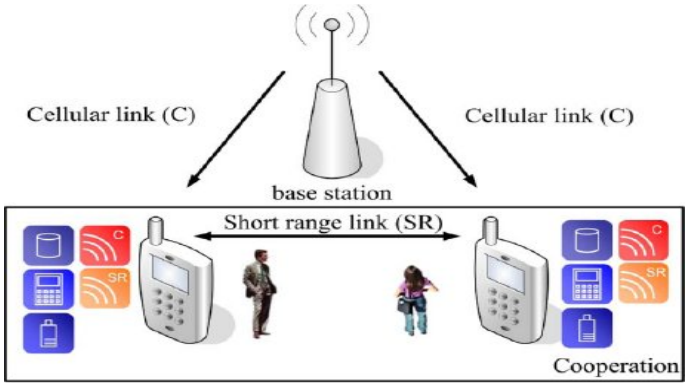


Figure 1.2: Cellular Controlled Peer-to-Peer Network.

Other problems arise exchanging information among nodes forming these wireless clusters. If several nodes are interested in the same set of data, this may be efficiently distributed among them with broadcast if no transmission errors are assumed. However wireless networks require retransmissions to reliably distribute the data. The end result of this is a decrease in throughput, since more transmissions are used to resend information.

In order to improve throughput in wireless networks a node could not just retransmit its received information in principle, could combine information received from many nodes to then forward that joint information to other nodes. This theory is called network coding and was introduced in [10] in 2000.

Figure 1.3 shows an oversimplified schema of three different data distribution methods. First an architecture of omnipresent cellular communication system, where all the packets have to be sent to every node. Second a cooperative communication architecture using

conventional network routing, where nodes retransmit its received information to the router first, who then forwards them one at a time. And finally a cooperative architecture using network coding where the router may encode the two packets together for a single broadcast, because boundary nodes can decode the encoded packet using their own original packet. Thus, the number of transmissions in the network in the last case is reduced.

Figure 1.3: Current omnipresent cellular communication system and conventional routing and network coding over a cooperative communication system

The objective of this project is to have a full functional demonstrator based on Nokia tablets N800 for network coding and cooperative wireless networking. Several devices are connected to an ad-hoc wireless network working as a cooperative cluster, to be able to exchange data through network coding.

Chapter 2

Analysis of Cooperative networks

This chapter explains why cooperative networking is an interesting field of study and discuss network coding as a cooperative technique for wireless networks. Also an overview and potentials of combining the Cellular and the P2P World are described.

2.1 Cooperative Networking

In today's world of cell phones, laptops, and internet tablets, an increasing number of devices use wireless communication to obtain and share data in networks. This increasing density of wireless devices gives the opportunity in most of environments to create short-range ad-hoc networks or cooperative clusters. Thus, new forms of information delivery among wireless devices emerge.

Information delivery from the cellular access network to the mobile device and viceversa are already evolutioning, starting from the conventional direct delivery in a centralized architecture, as given in Figure 1.1, to multihop cellular networks in which communication is not established directly between the user equipment and the base station, but intermediate devices act as repeaters between the base station and user equipment. But the omnipresent closeness of peer devices leads to delivering information through/from a short-range network as given in Figure 2.1 (Figure taken from [7]). A detailed account of this approach can be found in [7].

The most well known short-range wireless network technologies include wireless local area networks (WLAN), wireless personal area networks (WPAN), wireless body area networks (WBAN), wireless sensor networks (WSN) car-to-car communications (C2C), Radio Frequency Identification (RFID) and Near Field Communications(NFC).

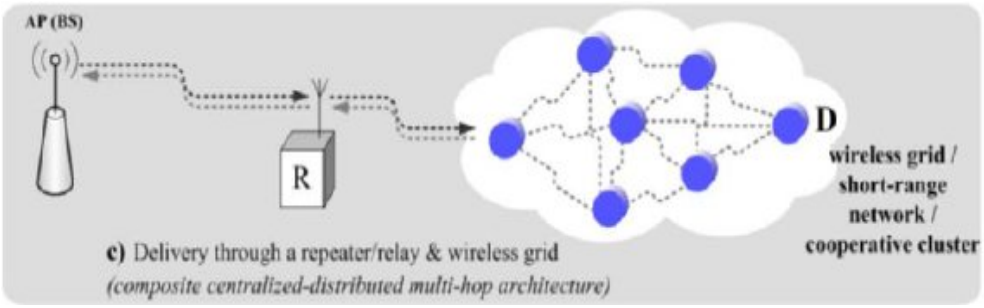


Figure 2.1: Evolutionary view of information delivery through the cellular access network.

As compared to long-range communications, short-range links require significantly lower energy per bits in order to establish a reliable link. They can achieve thus data throughputs of several orders of magnitude higher than the typical values for cellular networks[7].

2.1.1 Cooperation's potentials

Recently, cooperation concepts are emerging in a lot of engineering fields. Also, in wireless networks scope it has been introduced, with the overall purpose to join forces in order to reach a common QoS improvement. In fact, cooperation is deeply embedded at any wireless communication network, because it is assumed that all entities will agree on following pre-described protocols. However, users may modify behaviour or protocols for self or group interests.

In Figure 2.2 the principle cooperative scenario for a cellular environment is illustrated. Terminals receive their application from the base station over the centralized wireless link, and then use a short range wireless protocol for cooperating on a joint enhancement of the application QoS. Cooperation is made by exchanging information locally using the short range link.

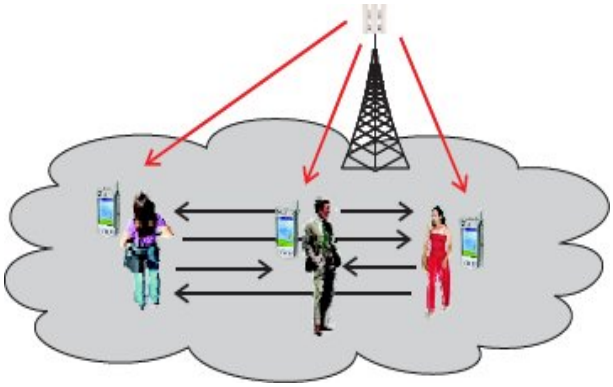


Figure 2.2: Principle of cooperation between terminals within a cellular network.

An overview of the characteristics, scenarios and potentials of cooperative techniques applied in wireless networks is presented in Figure 2.3 extracted from [7], where is ex-

plained that cooperation has the potential to improve the most important link and network performance figures, including achievable data throughput, quality of service, network capacity and coverage. Also it is presented the fact that cooperation can in principle enhance the efficiency in the use of radio resources (a really useful capability for systems limited by spectrum, power or energy) and also can be used in order to share and increase the capabilities of the interacting wireless devices, creating virtual devices with scalable architecture and enhanced features.

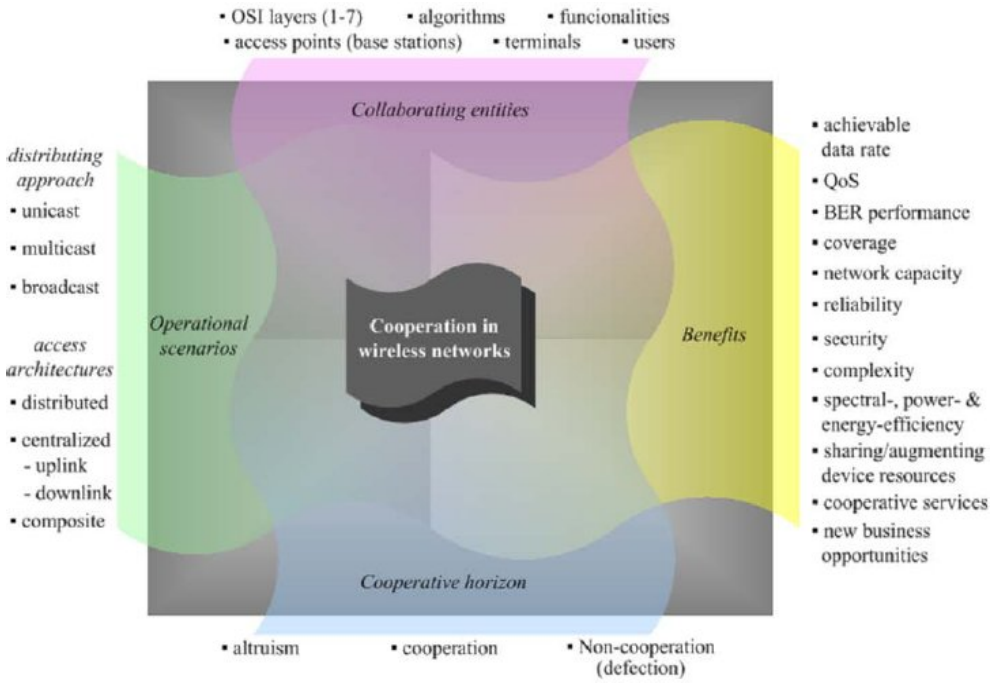


Figure 2.3: Cooperation in wireless networks: An overview of scope and benefits.

Figure 2.3 shows three main cooperative cases: altruism, where cooperation focuses on supporting or benefiting a third party (e.g., a relaying station) and thus no pay-off is expected, cooperation, where the interactions are mainly driven by selfishness (e.g., cooperating with other entities in order to obtain a clear benefit, or pay-off driven cooperation), and non cooperation, characterized by autonomic operation.

2.2 Network coding

A form of cooperation which emerged rapidly in recent years is network coding. Network coding is defined as "allowing and encouraging mixing of data at intermediate network nodes" [1] or "Network coding is the coding of network packets to maximize throughput and reduce the number of transmissions in a network" [9].

The simplest example based on [11] that outlines the benefit of network coding is a scenario with three wireless nodes: Alice, Bob and a router, see Figure 2.4 (Figure taken from [9]). Alice and Bob can both see the router, but not each other. If both Alice

and Bob want to send a packet to each other, they both have to send it to the router first, who then forwards them one at a time. Here the router becomes the bottleneck, because the flow of incoming traffic is larger than the possible flow of outgoing traffic. This scenario therefore costs four transmissions.

The router may encode the two packets together for a single broadcast, because Alice and Bob can decode the encoded packet using their own original packet. Thereby the scenario only costs three transmissions, and the router is no longer a bottleneck. Thus the throughput is improved because a decrease in the required number of transmissions to send the same amount of data.

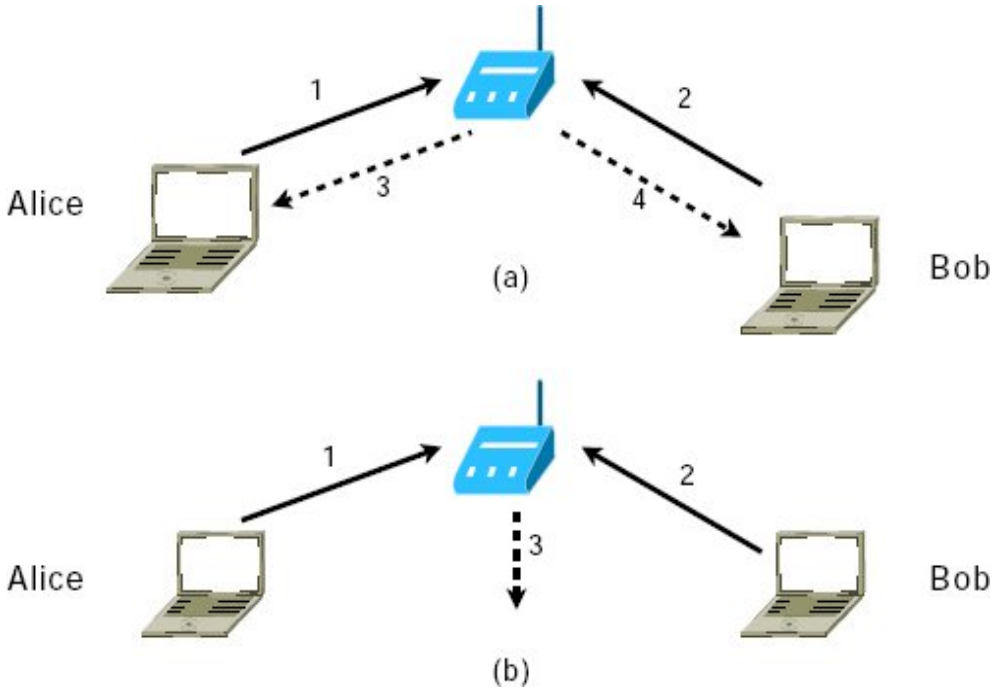


Figure 2.4: A simple example of how COPE increases the throughput. It allows Alice and Bob to exchange a pair of packets using 3 transmissions instead of 4 (numbers on arrows show the order of transmission).

2.2.1 COPE – A Practical Implementation of Network Coding

In the paper [11] a practical implementation of network coding is presented, using a simple form of coding, namely exclusive-or (XOR). This means the payload of packets are XOR´ed together. A shim layer is inserted between the IP Layer and the MAC Layer, which performs network coding, transparently to upper layers. This identifies coding opportunities, and benefits from them by forwarding multiple packets in a simple transmission.

COPE incorporates three main techniques:

- Opportunistic Listening – COPE sets the nodes in promiscuous mode and stores all overheard packets for a limited period T. In addition, each node broadcasts *reception reports* to tell its neighbors which packets it has stored.

- Opportunistic Coding – It is used by making each node use the following rule in order to ensure that all nexthops of an encoded packet can decode their corresponding native packets:

  To transmit n packets, p1 , ..., pn , to n nexthops, r1 , ..., rn , a node can XOR the n packets together only if each next–hop ri has all n – 1 packets pj for j = i.

- Learning Neighbor State – Nodes cannot rely solely on reception reports because they can get lost in collisions or arrive too late, after the node has already made a suboptimal coding decision. So, nodes may need to guess whether a neighbor has a particular packet. For that each node estimates the probability, that another particular node has a packet, as the delivery probability of the link between the packet´s previous hop and that node. This is possible because wireless routing protocols, including the one used in COPE, compute the delivery probability between every pair of nodes, to identify good paths through the network.

The packets are sent with a COPE header, which has three sections:

- A list of the native packets XOR´ed together in the packet,
- reception reports, and
- acknowledgements of received packets.

With the information distributed in the COPE header, each node is able to perform network coding.

This design has been evaluated on a 20-node wireless network and it has yielded a number of interesting practical results in the field of network coding:

- When the wireless medium is congested, with many random UDP flows, COPE increases throughput 3 or 4 times.
- In environments with hidden terminals, TCP does not use the medium, because TCP handles this as congestion, and therefore lowers the transmission rate, and few coding opportunities occur. If hidden terminals are eliminated, the increase in throughput reached 38%.
- For a mesh network connected to the Internet via a gateway, the increase in throughput ranges from 5 to 70%.

2.3 Combining the Cellular and the P2P World

As mentioned before the idea proposed in [7] is to enable direct communication between a group of mobile devices in close proximity using the short-range communication technology and forming a cooperative cluster contributing their onboard capabilities and resources to exploit them a more efficient way, while the cellular communication link is still enabled, as illustrated in Figure 1.2. This approach to bridge cellular and peer-to-peer network architectures, is referred as Cellular Controlled Peer-to-Peer (CCP2P) and has the potential to overcome many important limitations of current cellular networks, as introduced in the chapter 1.

For the case of a conventional cellular communication architecture, to support a requested service, cellular link has to provide a given data rate. In a cooperative case several wireless devices collaborate by each receiving a lower rate component stream of the whole signal and exchanging these streams over the short-range links. The same data rate is obtained, but through a less complex cellular air interface. As the short-range link supports higher data rates and exhibits a better energy efficiency per bit, the overall costs to support this service are less than for the first case. On the other hand, the cooperative case requires cooperative peer devices.

In this project a demonstrator for network coding will be developed using this network architecture, see Figure 2.5. Some nodes, forming a cooperative cluster, request for see a video. A PC, working as a server, sends different parts of the video to each node through a USB connection simulating the cellular link. Then, nodes exchange the video over the short-range links. The nodes thus are able to get the whole video cooperating among them.

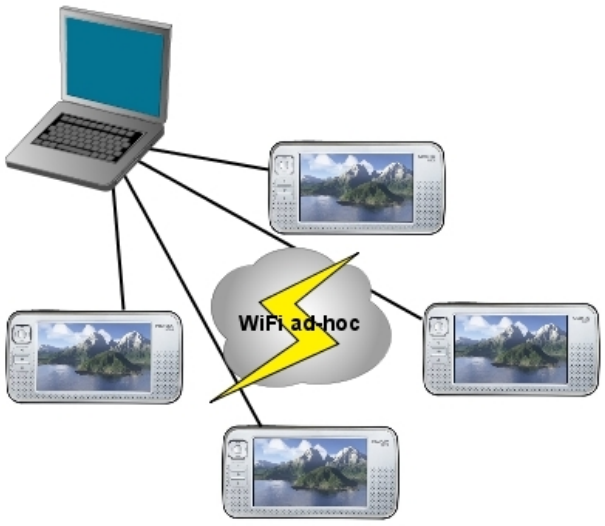


Figure 2.5: Exchangeing a video using a cooperative network

Chapter 3

Technologies

In this chapter will be described all the technologies that will be used during our master thesis. It will focus on the devices chosen as development platform for implementing the program, showing a technical description of them, and some limitations as well that we should take into account. Also Maemo technology will be explained. Maemo is the Operating System of the Nokia N800 Internet Tablet used in this project. Maemo website [4] offers developers very helpfull tools, making the development for this platform easier.

Finally we will discuss about the development environment used, and all the benefits that it offers to us.

3.1 Internet Tablet devices

During the whole project we used two different Nokia Internet Tablet devices: N800 and N810; these devices usually are smaller than a laptop, larger than a PDA, and quite lightweight. Some of them (e.g. Nokia N810) have a small keyboard, camera, and all of them have a stylus and a touch-sensitive screen, the stylus-driven GUI caused to develop a new graphic environment adapted to its needs, and also causes that all the applications have to be pre-designed to work with it.

Figure 3.1: Nokia N800 and N810 Internet tablet devices.

The hardware of both devices used within this project is quite similar, they share the same base, and the most distinctive difference is the keyboard and the integrated GPS receiver. In the Table 3.1 the hardware information is shown.

Technical specifications	
General info	800x480 pixel, 225 pixels-per-inch (PPI) wide-screen touch screen display with 16-bits per pixel color depth. Hardware buttons with a layout optimized for Web surfing. Virtual Keyboard Small slide-out keyboard (N810). Built-in VGA resolution webcam. 3.5 mm stereo audio out socket (works also as microphone input).
Connectivity	Wi-Fi (802.11b/g), Bluetooth 2.0. USB 2.0 port (in target mode and host mode).
GPS capacities	External GPS via Bluetooth (N800). Integrated GPS (N810).
Battery	1500 mAh battery.
Memory	128 MB of RAM. 256 MB flash memories with JFFS2 file system. 2 memory card slots, SD, MicroSD, MiniSD, MMC, RS-MMC. 1 memory card slot, MiniSD and MicroSD.
Processor	TI OMAP 2420 multi-core processor (max 400Mhz). TMS320C55x DSP logic (Backward compatibility with the 54x-series). ARM1136 core ("ARMv6") with an MMU (Backward compatibility with ARM926).

Table 3.1: Technical specifications of Nokia Internet Tablet devices

These mobile devices run a Linux-based operating system on an ARM architecture which is called Maemo (This will be covered in the next section).

As we see in Table 3.1 the processing capacity is limited but it is enough for our necessities, also the available memory is reduced, it only offers 256 MB integrated flash memory, 64 are being used from the base system, and only 192 are available for other purposes. Other limitations of the devices are the lack of graphics hardware acceleration in 3D or 2D modes, which is a considerable limitation for multimedia applications.

3.2 Maemo

Maemo is an operating system for the Nokia Internet Tablet line of handheld computers. It was originally named "Internet Tablet OS". This software platform for mobile devices was developed by Nokia, based on GNU/Linux and GNOME/GTK+ technologies. It includes proprietary components to make it work on the Nokia Internet Tablets.

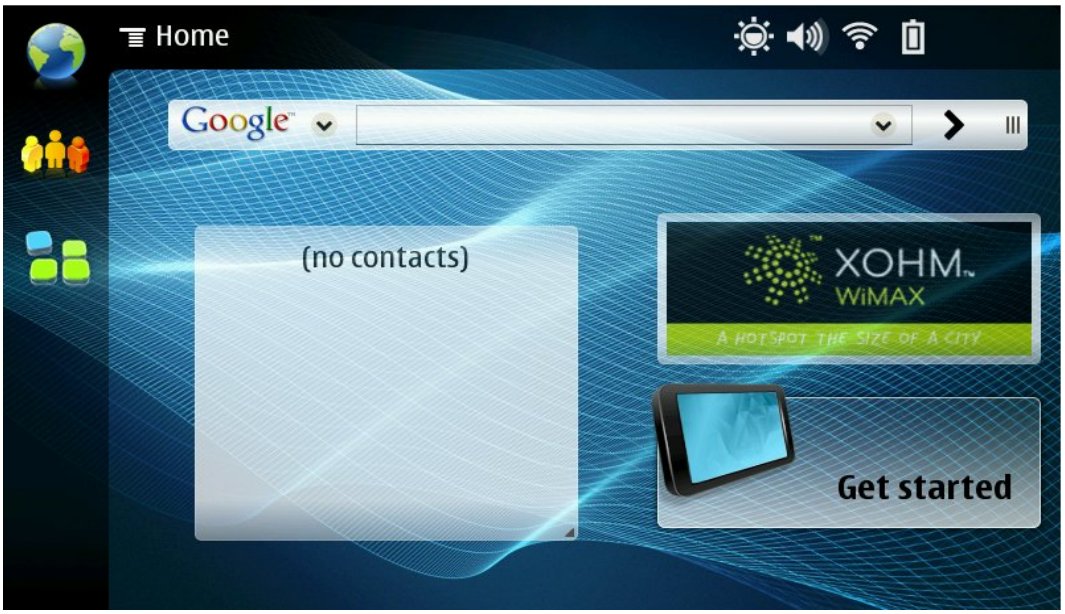


Figure 3.2: Maemo desktop screenshot.

Maemo is based on Debian GNU/Linux and draws much of its GUI, frameworks, and libraries from the GNOME project. It uses the Matchbox Window Manager, and the GTK-based Hildon as its GUI and application framework.

Maemo also uses BusyBox package which is a software package for embedded and mobile devices that replaces the GNU Core Utilities, trying to reduce memory usage and storage requirements (at expense of some functionality).

Although Maemo implements pre-emptive multitasking, the Matchbox Window Manager limits the screen to showing a single window at a time, although the home page allows multiple applications to have visible representation, and the system bar icons are dynamic allowing some display of live activity.

Despite being based on Linux and open source software in general, some parts of the Maemo remain closed source. For example the Wi-Fi drivers, which contain a binary blob, and some user-space software, like certain status bar and taskbar applets (including the display brightness applet) and applications.

3.3 Development Environment

In this section the development environment will be discussed, focusing basically in the two essential tools used Scratchbox, and EsBox [2].

3.3.1 ScratchBox

The Maemo SDK provides a "sandboxed" Maemo development environment on a GNU/Linux desktop system called Scratchbox.

Scratchbox is licensed under the GPL and it is open for outside contributions. This virtual environment behaves like the original operating system of the device, but with some extra development tools. This causes that development is quite similar to a desktop GNU/Linux development, and all the complications of embedded development, such as cross-compiling, are handled transparently by the "Scratchbox".

As we said before Scratchbox provides sandboxed development environment, which is a safe an isolated working environment, to develop Maemo applications. The name Scratchbox comes from "Linux from scratch" and "chroot jail".

While working inside Scratchbox, programs will be running in a changed root environment (chroot). In Linux systems, it is possible to change the part of file paths that a process will see. Scratchbox uses this mechanism on start to switch its root directory to something else than the real root. This is part of the isolation technique used. Because of this, the environment is called a sandbox, a private area where it is possible to play around without disturbing the environment, and without all the mess that real sand would cause. The other parts of the isolation technique are library call diversions, wrapping of compiler executables and other commands.

What Scratchbox can offer to the developer?

- It is a software package to implement development sandboxes (for isolation).
- It contains easy-to-use tools to assist cross-compilation.
- It supports multiple developers using the same development system.
- It supports multiple configurations for each developer.
- It supports executing target executables on the hardware target, via a mechanism called sbrsh.
- It supports running non-native binaries on the host system via instruction set emulators (Qemu is used).

Beside these main features, it is possible to develop own software packages that can be installed and used inside a Scratchbox environment. Scratchbox also includes some integration for Debian package management as the Internet Tablet also uses a similar packaging system, this means that packages built using Scratchbox and the SDK can be installed on the real device.

3.3.1.1 Hardware architectures

Maemo SDK comes by default with two environments for two different architectures. The x86 environment is used for the main development as it provides native performance, and better tools support since nothing needs to be emulated. And the ARMEL environment is used for working with the emulated device architecture. As we will see both have advantages and disadvantages, so the combined use of both could be a good solution to develop Maemo applications

Generally the x86 environment is used in active development, because it provides practically the same performance as normal GNU/Linux applications. Also, forgetting that the underlying architecture is different from the device one, programs usually behave exactly as they would, when compiled and run on the device.

When an application is running fine in the x86 environment, the next step is to compile it for the ARMEL architecture. The process for compilation and packaging is exactly the same as in x86, but a bit slower, because some of the required software is emulated. The developer needs not to worry themselves with cross-compilation, as Scratchbox will do everything for him transparently.

The applications compiled in ARMEL environment can be run straight on the device. Additionally, it is possible to run some of the applications inside the ARMEL environment in Maemo SDK. This is possible, because of the automatic emulation that Maemo SDK provides. The emulation is not complete, and things like multi-threading will cause problems, so actual testing must be performed on the device.

Using emulation for the whole development process should not be used because of the effect on performance and the problems commented before.

3.3.2 EsBox - An Eclipse plug-in for Maemo development

ESbox is an Eclipse plug-in that helps programmers to develop applications for Maemo platform on Scratchbox. It supports C/C++ and Python programming languages, different SDKs are supported as well.

It offers a deep integration with different applications to achieve a better developing performance. It uses Eclipse IDE as base, and actually only supports Scratchbox 1.x (e.g. Apophis R4), Scratchbox 2 support is in development at this time.

It also offers support for Chinook and Diablo SDK release, and both Nokia devices that have been used during the development of the project (Nokia N800 and N810).

The plug-in offers the user multiple possibilities, as choosing the desired SDK release that will use for this application and the architecture (x86 or ARMEL).

The whole application development process (with C/C+ +) can be done inside ESbox, it is possible to create a new project using one of the templates and examples that it includes like, “HelloWorld”, GTK+ applications, or Hildon GUI applications or create a new empty project to define it from the beginning.

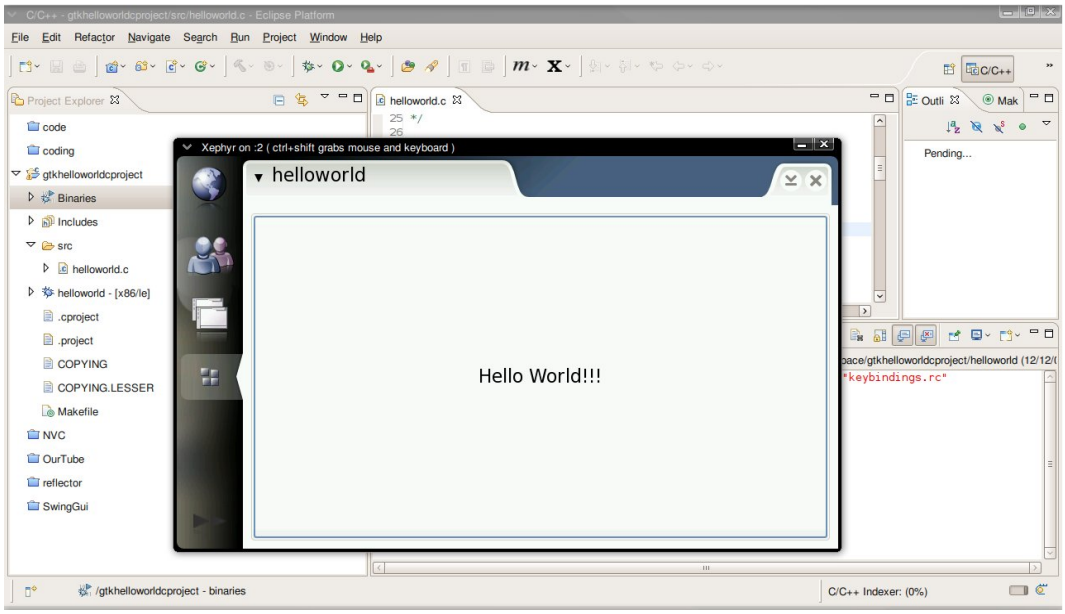


Figure 3.3: Esbox and Scratchbox running.

Within this environment it is possible to add new or modify code, choice the binary architecture, launch it in any of the scratchbox architectures, or upload and launch it directly in the real device. Moreover local and remote debugging is possible to perform.

Finally is possible to generate an installable package for the developed application to simplify the use for the final user.

Chapter 4

Requirement Specification

In this chapter the focus of the project is specified. To this point cooperative networking and network coding have been introduced as well as the Maemo software platform. Based on these approaches some improvements will be proposed and the developed demonstrator will be explained in detail, such as the equipment, layout and the overall flow of control throughout the program. The sequential and parallel flow is specified using activity diagrams, and these diagrams are used to illustrate requirements for the design and implementation.

Before activity diagrams are introduced, the next section focuses the main objective and the overall structure of the implementation will be analyzed.

4.1 Delimitation

As explained in chapter 2, the most important link and network performance figures can be improved by cooperative techniques. Network coding has been also introduced like a form of cooperation that increases the throughput of a network, decreasing the number of transmissions.

The objective is to have a demonstrator for network coding. Parts of a video will be distributed among several nodes forming a cooperative cluster. Through the short-range link nodes may exchange the complete video using network coding to codify the packets.

In [9] a implementation of network coding based in COPE on the Nokia N810 platform was developed, as well as a test application for evaluating its perfomance compared to broadcast. The test results evaluated the total number of packets available on the nodes as a function of time since the start of the test. Figure 4.1 shows the results obtained in [9] from this test.

The most interesting part to study in the figures is the shape of the two graphs. For BC it can be seen that the packet reception rate is quite linear in the middle part, but lower towards the end of the test. This have a main explanation: the nodes have most of the packets at the end of the test, so many receptions are not useful for a given node. Mostly of the last missing packets may be retransmissions which only satisfies a few recipients

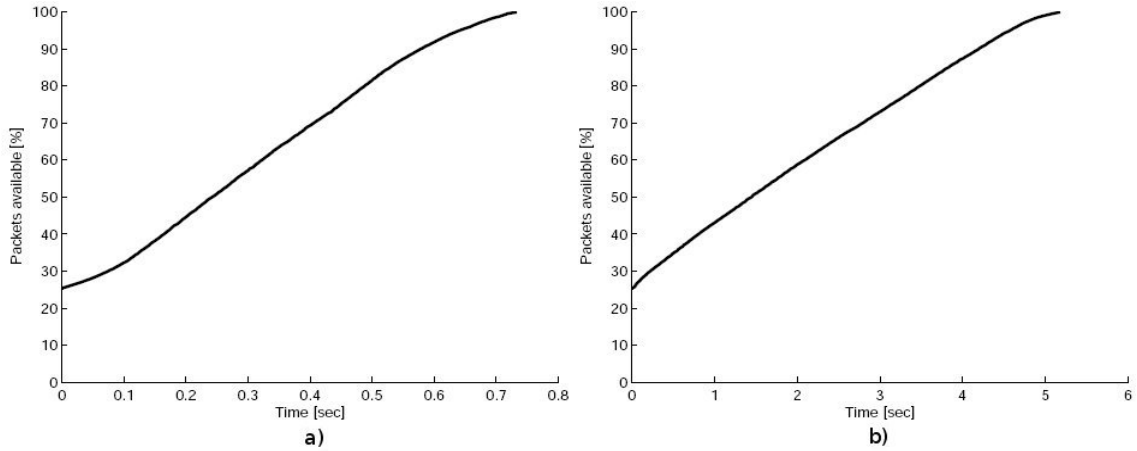


Figure 4.1: Average number of packets available on the nodes as a function of time when using a)broadcast b)network coding.

in cases where e.g. only two nodes received a given packet when it was sent for the first time. Furthermore, as most of the packets have been distributed at this time, more than one node is able to transmit a given packet to the nodes needing it. This leads to increased load on the medium and contention could pay a role.

4.1.1 Combination of Broadcast and Network Coding approaches

From last section we know that BC works properly at the beginning of the test, when transmissions satisfy most of the nodes. Then, NC is not useful, because correlation between packets in each node is very low or null. Therefore, encode packets together wouldn't be possible since any node in the network would be able to decode them. Moreover, processing time would be wasted to do the search. However, when the nodes already have most of the packets and retransmissions are necessary, reception rate drops using BC, then NC should have better performance, since sending packets encoded together would be possible.

So, why not combining both schemas? We will get a better performance during the whole test, since the resulting curve (total number of packets available on the nodes/time) should keep linear at the end of the test. Moreover at the beginning it would be faster because broadcast schema doesn't spend time looking for packets to encode with.

But, which is the proper moment to switch from broadcast to NC? First, each node gets a set of packets, e.g. in a scenario with four nodes each has $\frac{1}{4}$of the total set of packets. The nodes should broadcast all those packets, because everybody in the network would be interested in them. Then it is the moment to change to the network coding schema to retransmit lost packets since the nodes will have packets in common and encode them together will be possible.

This is defined as the main objective of the project: *To have a full functional demonstrator for network coding combined with broadcast at the beginning on the Nokia N800*

platform, and evaluate its performance in terms of delay, throughput and energy consumption compared to broadcast and other developed schemas.

4.1.2 Demonstrator Setup

Figure 4.2 illustrates the layout of the devices. As it has been mentioned before, we are using Nokia N800 Internet Tablets, that are connected to a computer acting as the data server through a USB hub. At the same time the tablets are connected to an ad-hoc wireless network, forming a cooperative cluster.

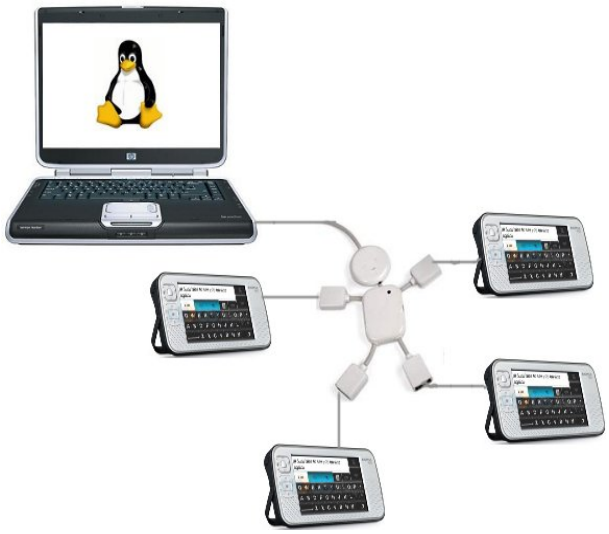


Figure 4.2: Layout of the devices

Server should transfer the video to the devices through USB. In the case of Figure 4.2, with 4 clients, each one should recibe $\frac{1}{4}$ of the total number of packets. Nodes will broadcast those packets received from the server to all the nodes in the ad-hoc network. Afterwards, retransmissions of packets will be sent using network coding. Finally, devices should play the video using mplayer.

This implementation is based on the program developed in [9], whose objective, as said before, was to implement network coding based on COPE. We have reused most of the functions related with NC. They defined a scenario to consist of 250 packets sent out, where the content of the packets was irrelevant. Now we have a scenario quite different. We have a video of around 11 MBytes, which is split in packets of 1024 Bytes to be exchanged. So, our scenario has about 11 thousand packets to be sent.

We find some problems to use the network coding implemented by [9] on our scenario. The first one is that NC was designed to support data sets with a maximum size of 256 packets. This was done for ease of implementation, because the packet id may then be expressed using one byte, what is not enough in our case. But the main problem is the reception reports which informs the recipient of which packets the sending node has. It is an essential part in ensuring that all nodes get all packets and are transmitted with

each packet. The length of reception Vectors is given by the total number of packets and they are stored by each node in a data structure. Therefore, the amount of packets in our scenario may not be supported, because of the packets payload would be huge and very hard to handle.

The solution is inject packets to NC in sets of two hundred and forty. As can be seen in Figure 4.3, when using the new hybrid schema first the set of data is broadcasted and afterwards lost packets are resent using network coding, being the maximum number of retransmissions 240 times the maximum packet loss probability.

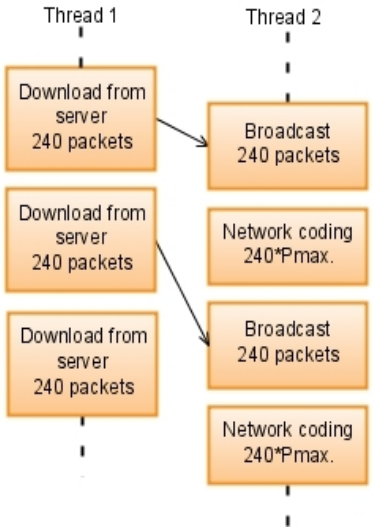


Figure 4.3: Shows how sets of packets are injected to NC

Moreover, the content of the packets is very important in this application since lost of content suppose frames lost in the video transmission. The delay takes an important role in this scenario as well if e.g. users are watching a football match.

The borders of the project are now defined. Based on this delimitation, the requirements for the program are specified in the following chapter.

4.2 Overall structure of the implementation

The implementation must do two things:

- Make it possible to use network coding.
- Introduce a video application for testing network coding.

These two tasks have been implemented independently in order to be able to interchange any of the two parts if e.g. we want to compare the performance of the application using network coding or using pure broadcast.

The two tasks are denoted as Video Application (VA) and Network Coding (NC) in the remainder of the report, where NC denotes the implementation made in this project of network coding (including different possible schemas), not the general technology of network coding. In [9] was also implemented a broadcast interface to distribute a set of data. We also use it denoted as Reliable Broadcast (BC). So, the video application can be tested for a network coding interface as well as for a broadcast interface.

We have reused some functions from the network coding implementation made in [9] to make NC as transparent as possible placing it in the network stack, as COPE has done it, see Section 2.2.1. Therefore communication between VA and NC have to go through sockets.

VA must have a way of starting and stopping NC and BC, as well as a way of restart NC since, as explained before, we inject data to NC in sets of two hundred and forty packets and, therefore, we need to initialize some counters and vectors before every new injection of a set of packets to NC. To do this, a states machine is used. It can be seen in Figure 4.4 and has the following modes:

- NOT_STARTED – It is the default mode from the start. It is also set during the restart function after each exchange of 240 packets.
- RUNNING – It is set when the node shall start transmitting packets, after a START packet is sent to NC.
- STOPPED – It is set when the whole video has been received, after a STOP packet is sent to NC.

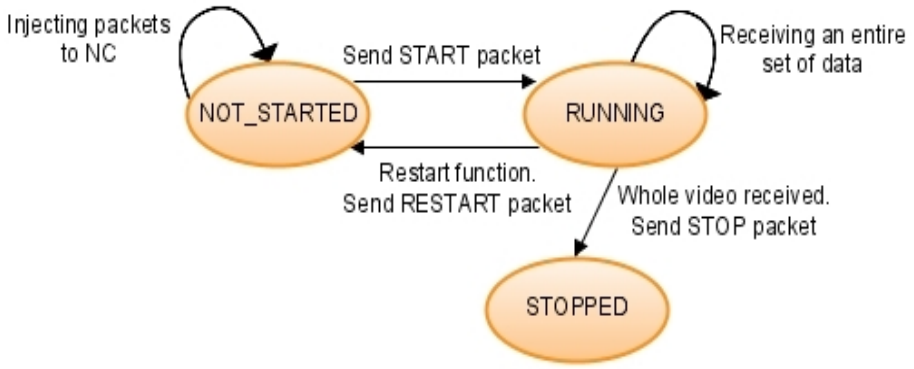


Figure 4.4: States machine of VA to be able to stop and start NC

Figure 4.5 shows the states machine of NC and how VA is able to stop or start NC. Other modes have been needed to synchronize NC since we need that every node of the cooperative cluster begins exchanging packets at the same time after each injection of a set of data. These modes are:

- READY – It is the default mode from the start. It is also set after each exchange of 240 packets when a RESTART packet is received from VA.

- RUNNING – It is set when a START packet is received from VA.
- COMPLETE – When a whole set of data has been received in NC a COMPLETE packet is sent through the ad-hoc network. It is done, as has been said before, to synchronize NC, that thus has to wait until receive a COMPLETE packet from every node in the wireless grid to be restarted.
- ALL_COMPLETE – It is set when a COMPLETE packet has been received from every node in the wireless grid.
- STOPPED – It is set when a STOP packet is received from VA.

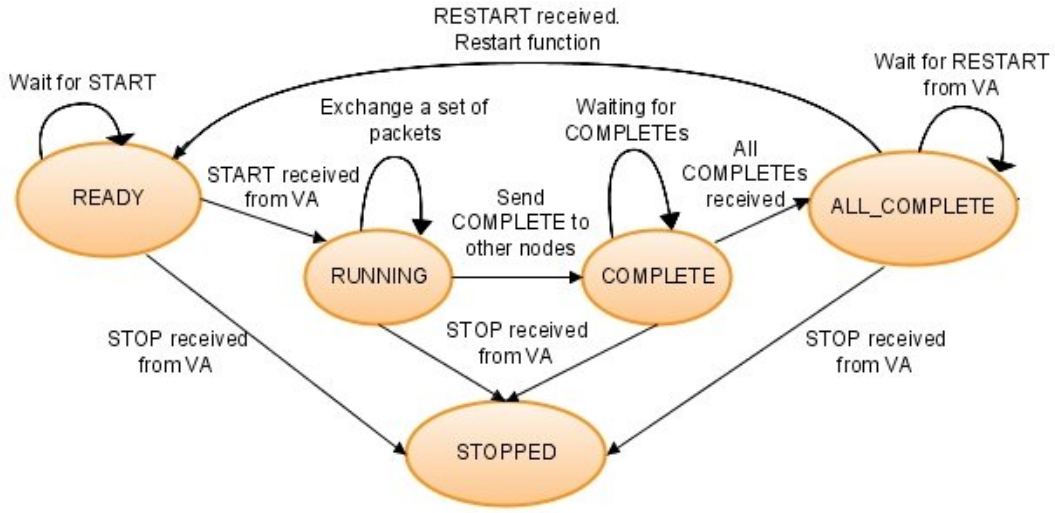


Figure 4.5: States machine of NC

4.3 Activity Diagrams

Figure 4.6: Notation used in activity diagrams

This section analyzes first the VA part and then the NC part of the implementation to identify overall control flow in the program. The notation used in the diagrams is shown in Figure 4.6 and is explained in the following:

- *Begin / End* – The entry and exit point of an activity diagram.

- *Activity* – An activity or task.
- *Nested Activity Diagram* – A list of activities, considered complex enough for an individual diagram.
- *Question* – A split in program flow, depending on a statement.
- *Sleep* – A thread sleeping until the conditional variable becomes set.
- *Fork* – A fork of one thread into two or more.

To give an overview of the activity diagrams in the following two sections, Figure 4.7 shows how the principal functions and threads are related.

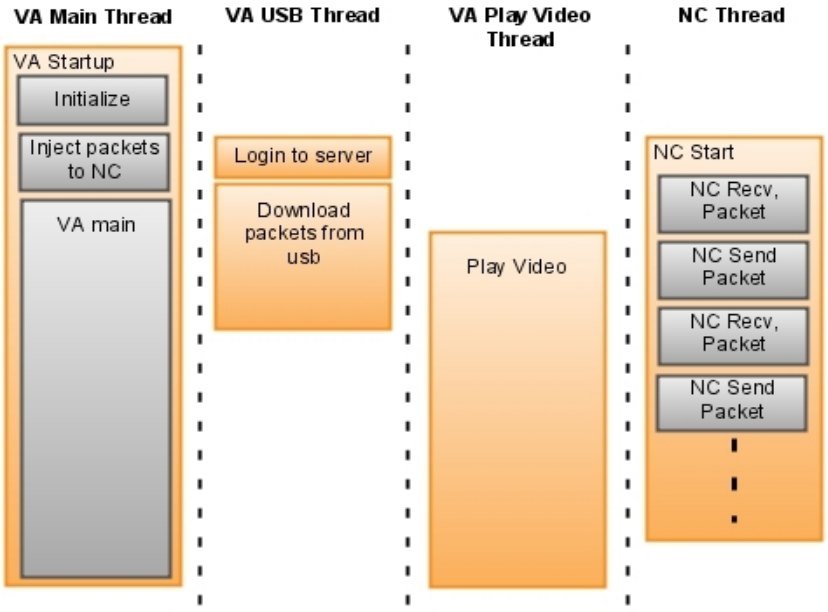


Figure 4.7: The correlation between principal functions separated in threads.

4.4 Video Application

VA is where the management of the video is located. It downloads part of the video from the server through USB and gets the rest of data from NC. Moreover, it enqueues the packets to be reproduced by mplayer.

4.4.1 VA Startup

When VA is first started, the activity diagram in Figure 4.8(a), is the entry point. Because some parameters have to be configured, as the schema used to exchange the video (NC or pure broadcast), a configuration file is used.

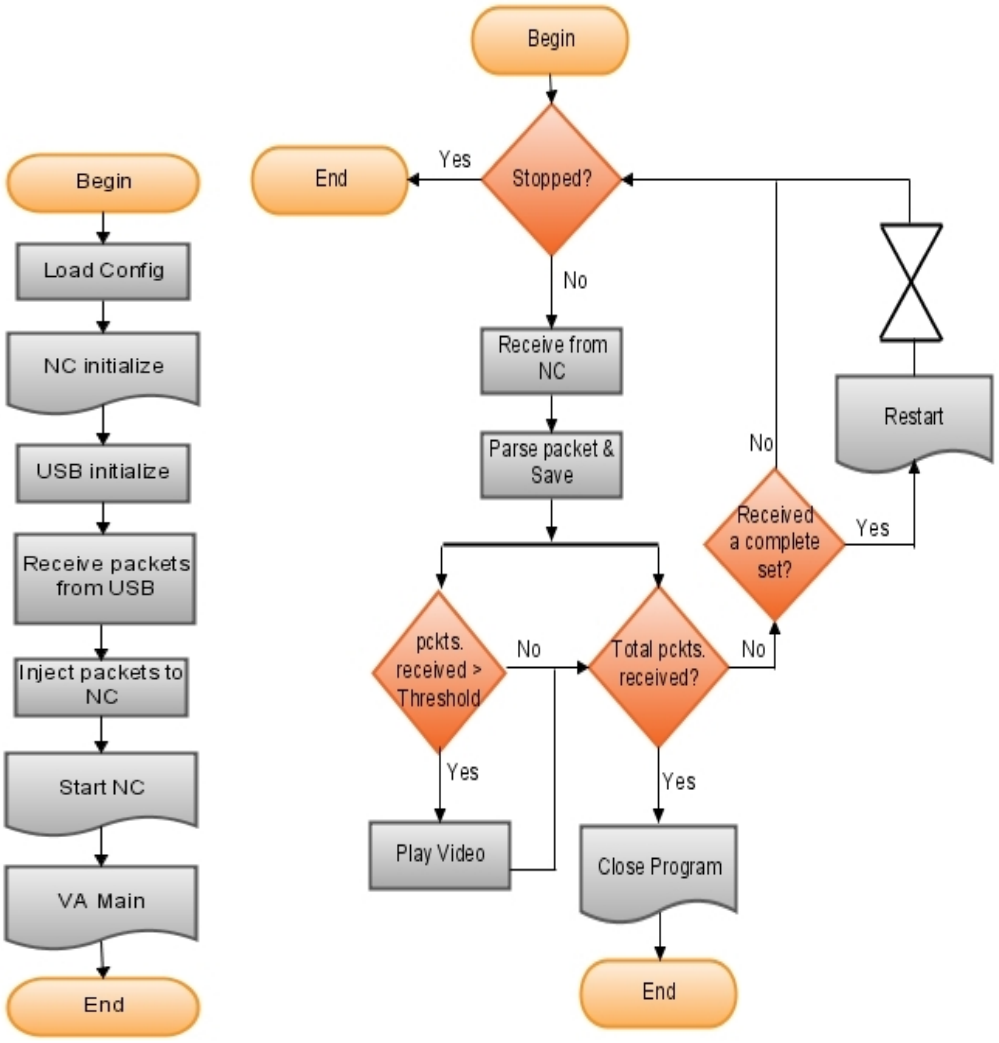


Figure 4.8: (a) Activity diagram of *VA Startup* (b) Activity diagram of *VA Main*

After the configuration file is loaded, NC is initialized as will be explained in section 4.4.4. Then USB interface is initialized. Two UDP sockets are opened, a "dataSocket" to receive data from the server and a "signalSocket" for signal messages with the PC. Next, client logs in by sending a "hello" message to the server and reception of packets from USB becomes active in a separated thread. Those packets received are injected to NC in sets of two hundred forty. It is done through a socket since it is the only way of communication between VA and NC. Injected packets are different from other packets like START or STOP packets, also sent by VA. Injected packets have a small header in the payload to specify the source IP address and a packet ID for NC.

After the packets are injected, NC is started, as will be explained in section 4.4.5. Then, the activity diagram in figure 4.8(b), which loops until program is terminated, becomes active.

4.4.2 VA Main

This activity diagram, see Figure 4.8(b), describes the main loop of VA. It receives data from NC through a socket and extracts and saves the information into a buffer.

When the number of received packets reaches a threshold, the video reproduction begins in a separated thread, as can be seen in Figure 4.7. It is done because we need to wait until having some packets buffered to start playing the video. When a complete set of data, that is 240 packets, has been received, the activity diagram in the Figure 4.9 becomes active. When it finishes we are able to receive new packets from NC again.

The video itself is played and rendered by mplayer, and the nodes write the incoming data packets into a FIFO that is read by mplayer. A secondary thread is responsible for writing into this FIFO, and freeing memory from data that has already been written into the FIFO. Using this separate thread is mandatory because you can only write some data into a FIFO if somebody is reading it at the other end, otherwise the write operation will be blocked.

4.4.3 VA Restart

After an entire set of data is exchanged and received in VA, a RESTART packet is sent to NC, then the mode is changed to NOT_STARTED and some counters and variables are initialized. Now we are able to inject a new set of data to NC and start NC again (see section 4.4.5).

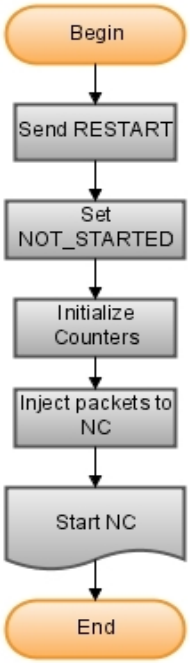


Figure 4.9: Activity diagram of *VA Restart function*

4.4.4 VA Initialize NC

The initialization of NC is done sending a special packet from VA with relevant information that NC needs, like number of nodes in the network, total number of packets to exchange, IP address of the node, etc.

4.4.5 VA Start NC

When VA has finished initializing and injection of packets has finished, to initiate NC a START packet is sent and the mode is set to RUNNNING, as shown in Figure 4.10(a). Following this, transmission of packets is started.

4.4.6 VA Close Program

Activity diagram in Figure 4.10(b) is activated. A STOP packet is sent and the mode is set to STOPPED, then the program should be ended.

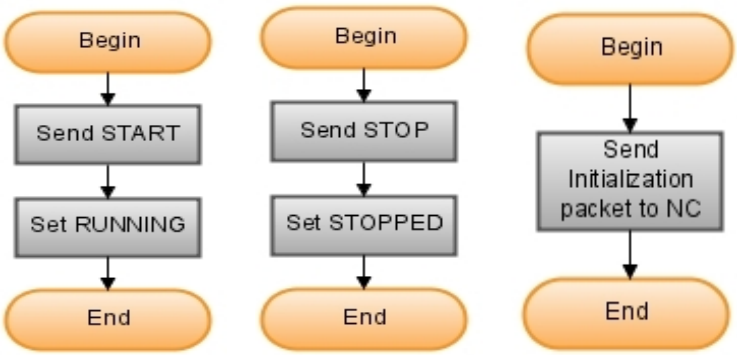


Figure 4.10: (a) Activity diagram of *VA Start NC* (b) Activity diagram of *VA Close Program* (c) Activity diagram of *NC Initialize*

4.5 Network Coding

NC is started from VA but runs in a thread of its own. As explained before, this implementation is based on the work done in [9] based in COPE (see [11]), but some improvements have been introduced, mainly the introduction of broadcast sendings at the beginning when it has a great performance. In this section some of the data structures are abbreviated in the figures, so a list is given here:

- RR – Reception report
- PP – Packet pool
- CV – Complete vector

The main activity diagram of NC is NC Main, which is explained next:

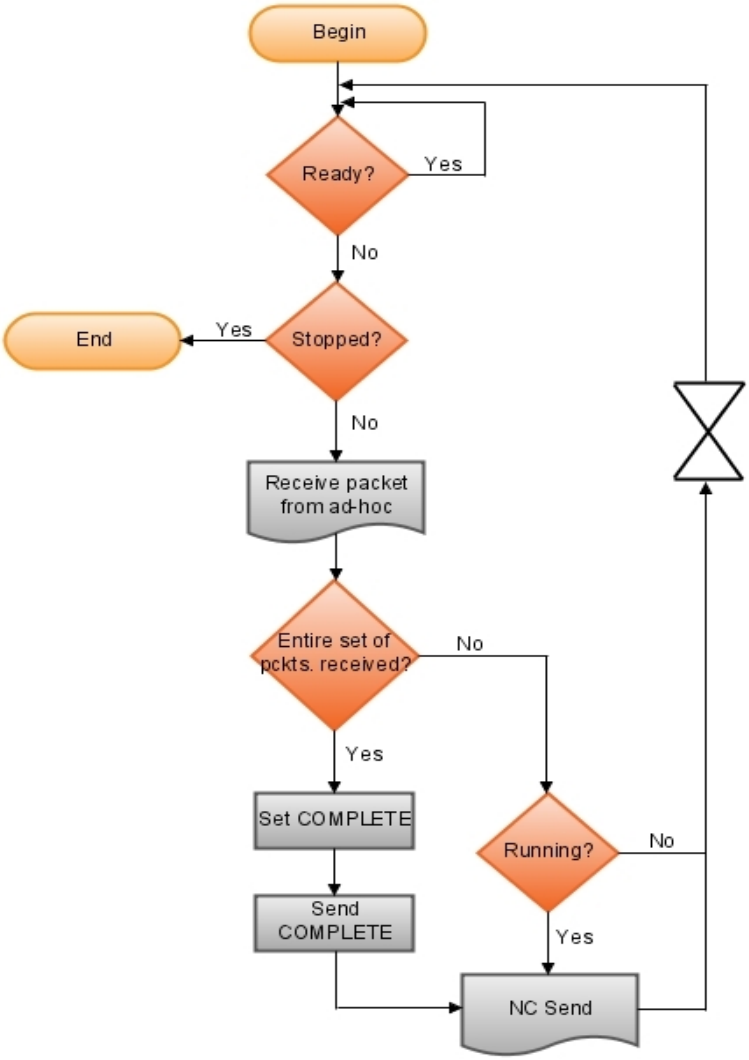
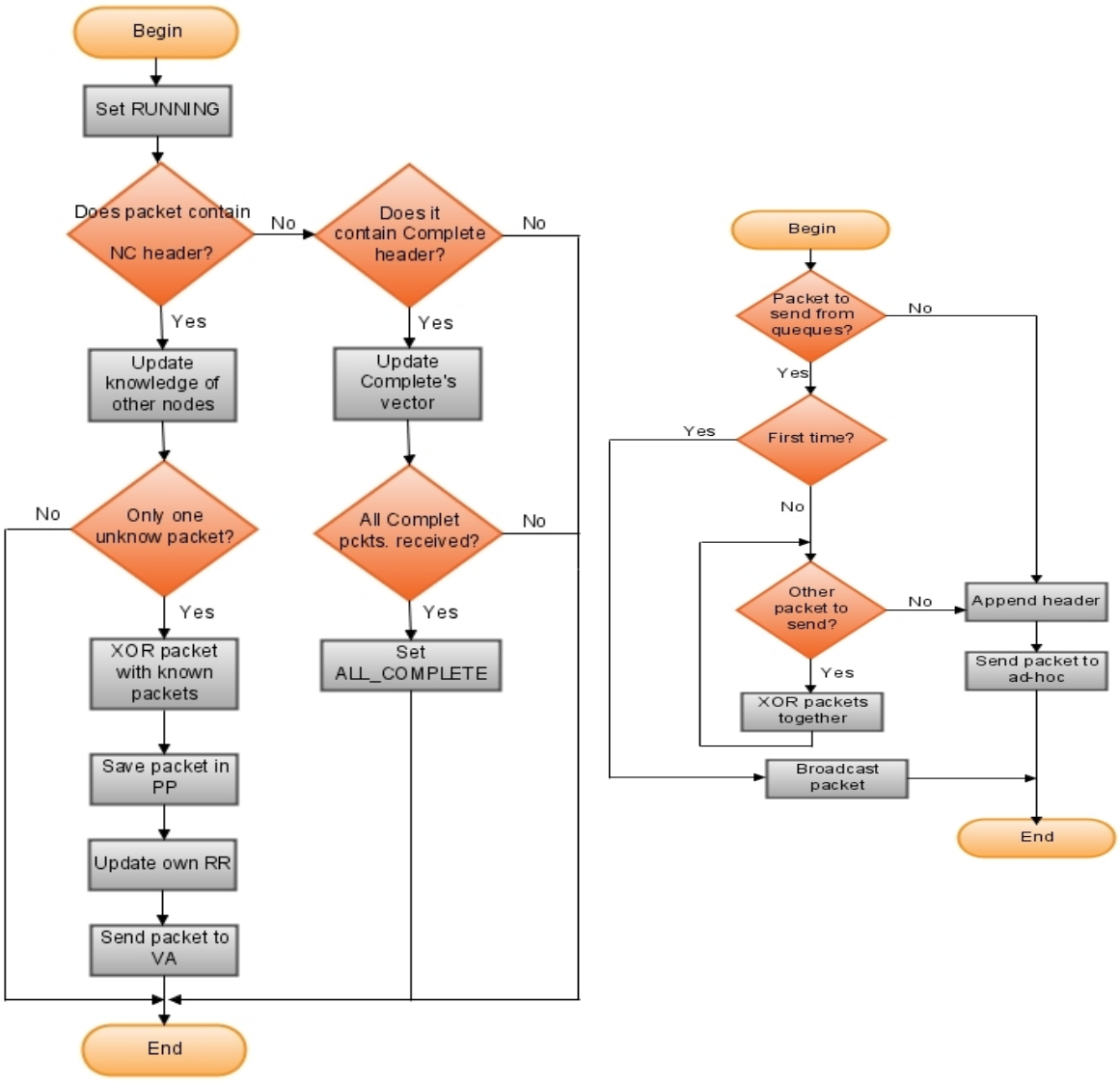


Figure 4.11: Activity diagram of *NC Main*

4.5.1 NC Main

This activity diagram is shown in Figure 4.11. As long as the mode is not set to STOPPED, when the loop ends, or to READY, when it keeps waiting because the system is being restarted, packets may be received from the other nodes in the cooperative cluster through the ad-hoc network, as will be explained in section 4.5.2. If mode is set to RUNNING packets may also be sent through this network. When a node has received the whole set of data COMPLETE mode is set and a COMPLETE packet is sent to the other nodes. This is done in order to synchronize all the nodes to begin at the same time to exchange packets. Moreover, the complete vector, a vector used to know which nodes are COMPLETE, is updated.



(a) (b)

Figure 4.12: (a) Activity diagram of *NC Receive packet from ad-hoc (b) Activity diagram of NC send packet*

4.5.2 NC Receive packet from ad-hoc

This activity diagram is shown in Figure 4.12(a). When a packet is received from the network interface, and it is identified as a NC packet, it is processed. Using the information from the packet, the knowledge of the sending node can be updated and the packet can be decoded. If the packet is coded, the known packets are retrieved, and XOR'ed with the received packet. After decoding, the node's own reception report is updated and the packet is sent to VA. Otherwise, if the packet is identified as a COMPLETE packet, the complete vector is updated. Then, the complete vector is checked and if it is all filled in with true, the mode ALL_COMPLETE is set. That means that all nodes are ready to restart.

4.5.3 NC Send packet

When a packet may be sent, it is first checked to see if the node has any relevant packet to send in its queues. If it has any and it is the first time that this packet is sent it is just broadcasted. After broadcast all packets in queues injected from VA, next times it attempts to find a packet which it can XOR, using an algorithm based on COPE uses (see [9]). This continue for as many packets as possible, and when no more packets can be found, it appends a NC header and sends the packet.

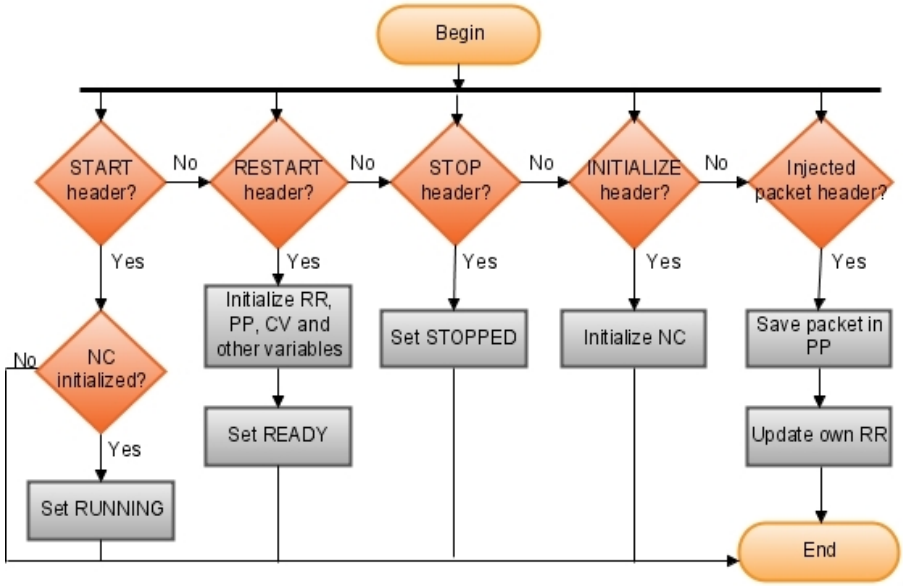


Figure 4.13: Activity diagram of *NC Receive packet from VA*

4.5.4 NC Receive packet from VA

As explained before VA is able to control NC somehow. There is an independent thread waiting for packets from VA. When a packet arrives, the header is checked and depending on which packet it is, different actions are carried out:

- START packet – if NC has been already initialized, RUNNING mode is set.
- STOP packet – STOPPED mode is set.
- RESTART packet – Packet pool, reception vector, complete vector and other variables are reinitialized in order to be able to exchange a new set of data. READY mode is set.
- INITIALIZE packet – This packet is parsed and some variables are initialized with the info in it.
- Injected packet – The packet is stored in the packet pool and the own reception report is updated.

4.6 Data Server

The server deals with sharing out the video amongst the nodes in the wireless grid. First, it divides the video up into packets of 1024 bytes and two UDP sockets are opened: “dataSocket” that will be used to transfer video data to the clients and ”signalSocket” that will function as a communication medium between the clients and the server.

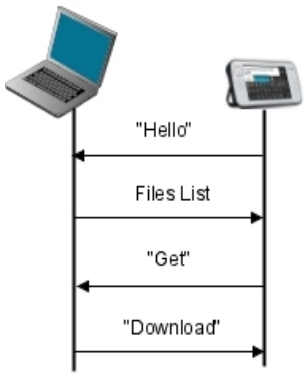


Figure 4.14: Messages sended during the log in process

The server is waiting for a number of clients to log in via the signalSocket. As can be seen in Figure 4.14 When a client logs in by sending the “hello” message, it gets a file list from the server. It chooses a file from this list, and sends a “get” message. If the server receives enough “get” messages for a specific file, then it starts the data transfer towards all the waiting clients at the same time. The beginning of the transfer is indicated by a “download” message which is sent to all participating clients.

After the login period, the clients will receive numbered data packets from the server. If there are 4 clients, then the client with ID 1 will receive the packets: 1, 5, 9, 13, 17... all packets having 1 modulus with 4. The clients will inject the received packets to NC to be distributed over the Ad-hoc WLAN network to all the other clients participating in the transfer. When they receive packets from NC the correct video stream is assembled based on the packet numbers.

Chapter 5

Design

The design of the program is derived from the activity diagrams in the previous chapter. The purpose of this chapter is to document the design of the program developed for this project.

The first section contains a design overview of the Video Application and the contents and functionalities that it will support. As it has been explained in the previous chapter, we want to develop an independent and modular solution. Two basic modules have been developed, one the Video Application which will be the responsible to process the received packets and play the whole video, and the Network Coding framework which will be the responsible to exchange the packets using the Network coding algorithm. As they are independent, it's necessary to design an interface between them, that will be described deeply during this chapter.

The most important parts inside the Network Coding framework are the data structures, used to save all the needed information and the algorithms, to choice which packets are sent at each time (called schemas). The design of this will be covered during this chapteras well.

To help the reader understanding the algorithms and code developed for the application, all the commented functions will be showed with simplified pseudocode. This simplified pseudocode do not show the whole complexity of the application, but gives the reader the basic idea back of it.

5.1 Video Application

The video application will be the manager of the whole system, from the previous activity diagrams we know that it has to offer the next functionalities:

- Communicate with the streaming server.

- Save the received packets from USB.
- Communicate with the network coding framework through the interface.
- Inject and get the necessary packets from/to the NC framework.
- Buffering and write the video file for the multimedia player (MPlayer).

As these functionalities should be provided at the same time, the use of multi threading techniques is needed to achieve the desired results. For this purpose the "Posix Thread" library, usually called pthread, is used. This library defines an standardized API for creating and manipulating threads that helps on the development of multi-threading applications (see [5] for more details).

To achieve the multiple tasks previously commented, the VA will need five different threads:

- NC_thread, which has to launch the underlaying NC framework.
- NC_receive, which will save the received packets through the NC framework.
- USB_thread, which will establish a channel to communicate with the streaming server.
- Play_thread, which saves all the video in a queued file and also launch the Mplayer.
- Main_close thread, which provide the logic to the application.

Also a state machine should be implemented to permit some synchronization and control of the different running threads on each moment and how they will perform in each state. This state machine is defined by an int variable called mode, which is protected with a mutex to avoid that multiple threads access to the same memory position at the same time, so the access at this variable should be done through the *setmode()* and *getmode()* methods provided.

The VA state machine is based on the theoretical state machine described in section 4.2, but during the design and implementation two additional modes have been added. These are QUIT_PREPARE and QUIT, which are used to shut down VA and NC.

Next list shows all the possible modes and their generic behaviour:

1. NOT_STARTED: NC exchange has not been started or is restarting (default value on program startup).
2. RUNNING: VA and NC are running, this the normal operation status.
3. STOPPED: NC exchange and video play have finished.
4. QUIT_PREPARE: Program begins shutdown.
5. QUIT: Program terminates gratefully.

As we commented before this state machine will control the whole application, by default mode is set to NOT_STARTED on startup. In this mode the application is waiting for an event. Only two possible events can make the mode change from NOT_STARTED to RUNNING, this could happen because the user injects information to NC, or it could also happen if NC sends information to VA. During this state the node always try to exchange information with other neighbors. After a successful exchange of a set of data the mode can change to different modes, if there is more video information to exchange, the mode is set to NOT STARTED again, and it is repeated all the necessary times until all the video data has been completely exchanged. On the other hand, if all the nodes have exchanged the video data, then mode is set to QUIT_PREPARE. When this mode is set, the program begin shutting down gracefully all the active threads and when QUIT is finally set, the program closes automatically freeing all the used resources.

The main objective of the Video Application is to offer an integrated demonstrator that will be able to transmit multimedia content over a cooperative network, to aim this the application should be enough flexible, allowing the final user choice multiple different configurations. Like using different NC schemes, different number of packets for each NC exchange round, or different packet length.

The list bellow shows the source files that form part of the VA:

- main.c
- confload.c
- USB_receive.c
- play_video.c

Due the complexity of the whole application before focusing in the behaviour and the source code of our application some important aspects of it will be explained.

5.1.1 VA Data structures

In the main application some important data structures are defined. These data structures are used intensively by the application to manage the processed video information and are explained next.

5.1.1.1 VideoBuffer

It is necessary to define a Buffer called *videoBuffer* to hold all the video data (called *bufInd*). This is achieved using a double pointer static vector.

Several position pointers are related with this buffer, since different threads have to write information in it or read from it. Each thread has to know at which position has to read or write information.

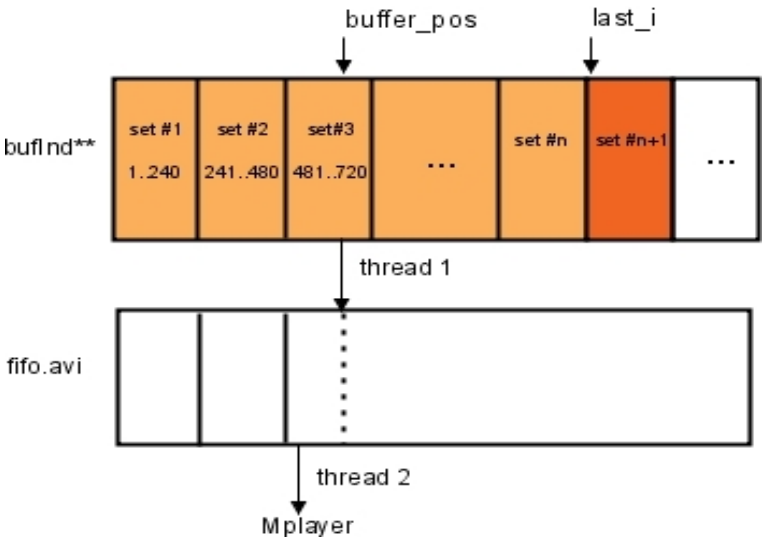


Figure 5.1: Handle of packets to assemble the video stream

The *last_i* pointer (see Figure 5.1) points to the first packet of the last set of data injected to NC which are being exchanged during the current round (calling round to an exchange of one data set). In Figure 5.1 can be seen as the sets of packets from 1 to n+1 have already been injected to NC, all of them have been received by VA except set number n+1 which is being exchanged by NC currently, so these packets are not available in VA yet.

The *buffer_pos* pointer forwards all the buffer writing the available data into a FIFO that is read by mplayer. This writing is done until the *last_i* pointer is achieved, if both pointers arrive to the same buffer position the *buffer_pos* will stop until the *last_i* pointer is updated to a new position.

5.1.1.2 PacketCounters

Different packet counters are needed to maintain a control of the information that VA is managing. It has been already explained that NC framework cannot exchange more than 256 packets at the same time, so we need to inject packets in blocks smaller than this quantity. Therefore it is necessary to have a *packet_counter* that will control each exchange of a set, and also a *totalpacket_counter* that will be used for control the whole amount of packets exchanged at the moment.

5.1.2 Configuration

The nodes are configured using an XML file, which is distributed before launching the application. By default, the video application looks for a file named conf.xml, but also is possible to set an alternative name by giving a parameter when starting the program. The configuration file contains the basic information to configure each node.

The use of the Extensible Markup Language (XML) [3] is not an arbitrary choose, its purpose is to aid information systems sharing structured data in a human readable way. Other aspects that make XML a good choice is the flexibility that offers, easy to use, and also the huge quantity of parsers that are available for the development. The Doctype declaration of a configuration is shown below to outline the structure of the configuration file.

```
1 <!DOCTYPE vaconf[
2  <!ELEMENT vaconf(type,scheme,packetlength,packetno,nodeno,
      severaddr)>
3   <!ELEMENT type (#PCDATA)>
4   <!ELEMENT scheme (#PCDATA)>
5   <!ELEMENT packetlength (#PCDATA)>
6   <!ELEMENT packetno (#PCDATA)>
7   <!ELEMENT nodeno (#PCDATA)>
8   <!ELEMENT severaddr (#PCDATA)>
9 ]>
```

The configuration contains the basic parameters that a node needs. First of all, the type of coding is given, being either NC or BC. In our application direct BC is never used as we always use the NC framework within different schemes. Next element is the scheme used, this could be a value between 0 and 3 depending on chosen schema. 0 is the value used for the broadcast schema, 1 is for the most_have NC schema, 2 for least_have NC schema and 3 is for the hybrid broadcast plus NC schema.

Packet length is fixed for all the packets, actually it is not a real problem, as the streaming server also generates fixed length packets. Finally it contains the number of nodes, and also the IP address of the streaming server because the node should login to get the desired packets.

Parsing and loading of the configuration file is done using libxml2 [6] within the function *confload()*. This recursive function is called during initialization and iterates through the XML tree, loading all values and setting them into a parameters structure.

5.1.3 Main application behaviour

When the program is started, the *main()* function initializes all relevant parts of the program as described below. The order of these actions is important as many of them have relations with the others.

This function is designed according to the activity diagrams showed in chapter 4.3. Some parts of the program run in separate threads to enable different actions to be performed concurrently using the pthreads.h library. Also mutexes are used to ensure that more than one thread access the same memory position at the same time.

Next list show all the activities related with the main application

1. Initializes all variables at default values.

2. Creates the mode variable using a mutex and sets the value at NOT STARTED.
3. Sets the nodeid value from the command line arguments.
4. Loads the configuration from the XML file and sets the data structure.
5. Initializes the log file.
6. Sets the threads priorities attributes.
7. Launch main close thread.
8. sets the Control-C signal handler to shutdown gracefully.
9. Launch sequentially all the threads.
10. Waits until all the thread activities has end.

Shutting down the whole program is performed only when all the running threads have finished. For this purpose the function *pthread_join()* is used, this function waits until the end of a given thread before continuing. When all the launched threads have finished, the dynamically allocated memory used is freed, the Log file is closed, and all the mutexes are destroyed.

5.1.3.1 Main close thread

The main close thread is one of the most important threads in our application since it is the thread that gives the logic of our application to the NC framework.

It's formed by a loop that always performs the same logic. It is shown in the following list and can be also seen in Figure 4.8(b), the activity diagram of VA Main.

```
1  if (!completed_exchange){
2      wait();
3  }
4  else{
5          if (video_has_been_exchanged){
6                  close_app();
7          }
8          else{
9                  restart_app();
10         }
11 }
```

As we see the behaviour of the application checks in each iteration if the last exchange of a data set has been completed. When the exchange of a set of packets is finished a second check is performed: if the whole video exchange has been performed it tries to shutdown the application, but before it is necessary to wait until the whole video has been played; otherwise it will restart the NC framework and inject a new set of packets. To end the loop execution the active mode should be STOPPED, if this happens, the loop and the thread will be finished freeing all the resources used.

5.1.3.2 NC receiving thread

This thread is responsible of listening the packets received from the NC on a socket, processes them and saves them in the correct position within the *videoBuffer* based on the packet numbers.

The logical of this thread can be seen in the pseudocode fragment next, and also in Figure 4.8(b), the activity diagram of VA Main.

```
1 init_receiving_socket();
2 while (!all_packets && mode != STOPPED){
3         // Wait until a packet arrives
4         if (packet_length > 0){
5                 parse_packet_and_save();
6         }
7 }
8 close_receiving_socket();
```

The thread opens a socket to listen all the packets that NC will send to it, once the socket is opened, the loop begins, waiting for packets to arrive. When some valid information arrives the packet is processed, avoiding the unnecessary headers, and saving it in the *videoBuffer* for future uses. This loop ends when all the packets have been exchanged or if the mode is set to STOPPED. Once the loop is finished the socket is deleted to free system resources.

5.1.3.3 NC thread

This thread launch the whole NC framework subsystem. The operation of this framework will be seen deeply in section 5.2.

Basically, it performs a basic configuration for the underlaying NC framework threads, setting the basic initialization parameters of NC framework, initializes the tunnel interface and opens a raw socket to send information through it. After that, it launches all the underlaying NC threads and waits until they have finished. Once they finish, it frees all the resources that have been used by NC framework and ends.

5.1.3.4 USB thread

The USB thread has to perform all the USB communication related tasks, such as login on the streaming server, ask for the desired video file, get all the properties or save in the *videoBuffer* the packets received from server through USB connection.

The behaviour of this thread is shown in pseudocode below:

```
1 init_USB_receive_socket();
2 login();
3 while (!all_packets && mode != STOPPED){
```

```
4           // Wait until a packet arrives
5           if (packet_length > 0){
6                   parse_packet_and_save();
7           }
8           if(packets == USB_THRESHOLD){
9                   start_injection = TRUE;
10          }
11  }
12  USB_close_socket
```

The thread opens a socket to communicate with the remote server. Once the socket is opened, it tries to login in the server, if the login process is done, the loop begins, waiting for packets to arrive. When a valid packet arrives it is processed and saved into the *videoBuffer*. When a defined threshold is surpassed the *start injection* variable is set to 1 , this is to advice the main thread that the injection of packets can begin. This loop could die for two reasons, the first one and the usual one is because all the needed packets have been received, the second one is because the mode has been set to STOPPED, so then all the threads should destroy themselves. After the loop ends and before the thread dies, some cleaning tasks as closing sockets are done.

5.1.3.5 PLAY thread

The function of the play thread is to iterate the *videoBuffer* structure, and write the available information to the Mplayer queue file. This loop is given in pseudocode below:

```
1  while (counter < PLAY_THR){
2           wait();
3  }
4  create("FIFO.avi");
5  launch_player();
6
7  while(current_pos < bufferLimit && mode != STOPPED){
8           if(videoBuffer[current_pos]){
9                   write(videoBuffer[current_pos]);
10                  current_pos++;
11          }
12          else{
13                  wait();
14          }
15 }
```

This thread waits since the beginning until a predefined threshold is reached (by default is set to 1200 packets). This is necessary because we need some buffering before playing the video to avoid unnecessary video freezings. Once this threshold has been passed, it setups the FIFO queue, and creates the dropping file for the mplayer. When the file is ready, the next step is to launch the player, and begin to write the available information

into the queue. It is important to note that if the *videoBuffer* is iterated faster than the NC coding can write new information on it, it will get out of information easily. If this happens, the loop will wait a second and then it will try again to read information from it. The loop ends when all the *videoBuffer* has been iterated, then is time to close the file and after that the thread ends.

5.2 Network Coding framework

As we said before, Network Coding framework is independent of the Video Application, so a common interface between them has to be designed to allow the communication between them.

This interface has to offer this basic functionalities to the VA.

- Allow to setup all the parameters of the NC framework.
- Inject the necessary packets from the VA to the NC framework.
- Return to the VA new packets achieved through the exchange.
- Start the exchange of packets within the NC framework.
- Restart the NC framework for next exchange round.
- Shutdown the NC framework.

This interface will be provided by an internal UDP socket, so all the commands and packets sent from the VA to the NC framework will be sent as an UDP datagram with some extra headers (this will be covered in detail in section 5.2.4).

Inside the NC framework several tasks are performed at the same time:

- Receive commands and packets from the VA.
- Send encoded packets through the medium.
- Received encoded packets through the medium and pass it to the upper VA.

That force us to use again several sockets and threads to achieve all the functionalities. Three new threads will be used, one per each independent functionality. As we done with the VA application is interesting to define an internal state machine, that will help us controlling the different running threads.

This state machine for the NC framework has 5 different states which are:

- READY, the NC framework is ready to perform any desired task.
- RUNNING, the NC framework is running and exchanging packets with other neighbors.

- COMPLETE, this node has completed the exchange, is sending packets to other nodes.
- ALLCOMPLETE, all the nodes have all the packets, no more exchanges will be done.
- STOPPED, the NC framework is completely stopped.

While the packets are sent to the medium, they need to be easily identified by the receiver as a NC packet, so some NC headers will be added. This will occur for the SYNC packets as well that will be used to maintain the synchronization among the devices involved in the video exchange.

5.2.1 General architecture overview

As we can see in Figure 5.2, the NC framework is designed to be an interface between the IP layer and the MAC layer, offering to the upper levels the service of encoding, broadcasting the NC packets and decoding the packets in a transparent way for the final application.

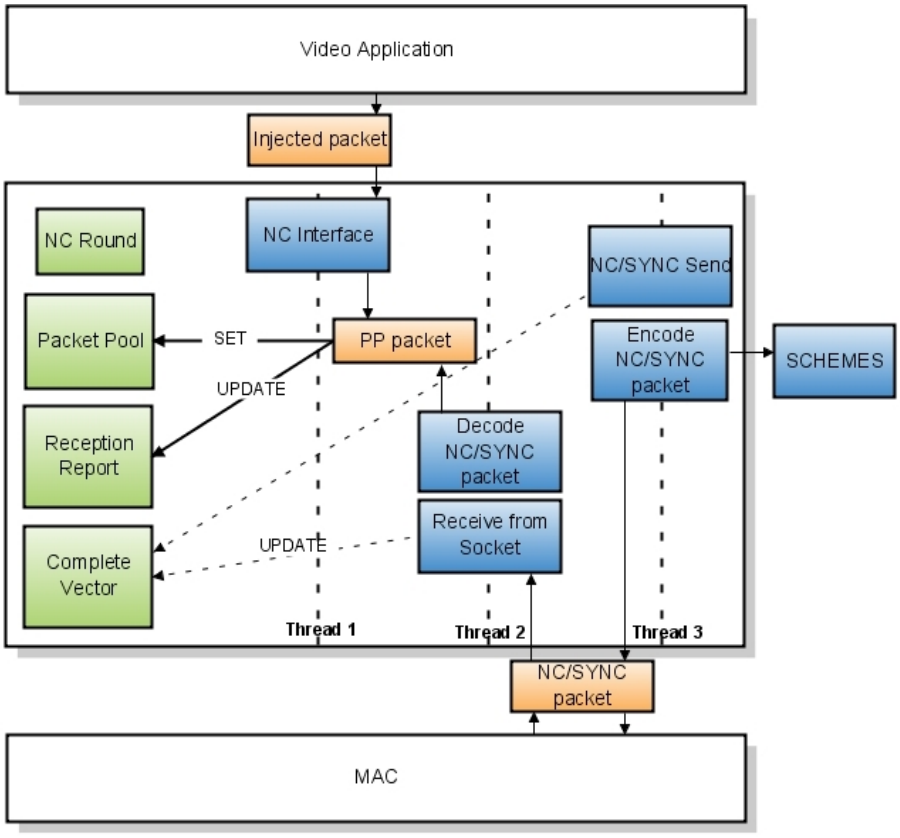


Figure 5.2: NC Framework's most important functions

As we can see in Figure 5.2 there are three basic modules: the data structures, the behaviour of the underlaying running threads, and the schemes that can be used for coding the packets on the framework.

The following sections will describe how NC is designed focusing on the main modules shown in Figure 5.2. This modular design is used to allow an easier development in the future. Also is important to comment that all the used modules are completely interchangeable, the only requisite is to maintain the proposed interfaces between them.

In the next sections the Data structures and implementation details will be described.

5.2.2 NC Data structures

One of the most important sections in the NC framework are the data structures created to save all the exchanged information, and also the states of this exchange. For this purpose two different structures are used: the packet pool, which is a place to save all the packets interchanged and the vector storage.

5.2.2.1 Packet Pool

To support NC encoding techniques, the framework has to be able to save all the packets that each node has, so the use of an indexed pool for the packets has to be designed to offer the needed functionalities for the NC framework.

Every time a packet is received at NC framework the packet is handled by the Packet Pool, which will save it inside the data structure in a proper way.

Because all the packets of the set will be managed, the Packet Pool must offer a capacity equal or larger of the exchanged set, to provide a correct delivery of all the packets. Otherwise information could be lost.

The Packet Pool consists of data structures called PP Packets, this PP packets are formed by the original IP Packet plus a 16 bits identifier which will be the *packet_id* given for to the packet during this exchange (basically a number between 1 and 255). As we can see this data structure could hold until 65536 packets, but the actual implementation will not hold more than 255 packets simultaneously (due the processing limitations of the development platform). This structure also offers to the NC framework some basic functionalities as set a new *pp_packet*, get a *pp_packet* from the id, get an *ip_packet* from the id, and more.

5.2.2.2 Vector storage

NC framework also needs to maintain a control of which packets has each node, for this purpose the vector storage is used, this vector storage contains one packet status vector for each node.

As we can see in Figure 5.3 inside the data structure the actual state of each node is being saved as a bitmap, using one bit per each packet. This information is crucial,

Node ID	Vector
1	11111011011110.....10001000100010001
2	11011110011110.....01000100010001000
3	11111010111011.....00100010001000100
4	11111011110011.....00010001000100010

Figure 5.3: Vector storage data structure

cause the vector storage will be constantly interrogated by the scheduler as the choice of packages to send is closely related to the available and remaining packets in the nodes.

Also this data structure offers the NC framework some management functionalities, as init, shutdown, some getters and setters for the packets status, get the whole bitmap of a node, update the whole bitmap of a node, free the vector, and so on.

5.2.2.3 NC Packet headers

Once a packet has been sent through the medium some special headers have been added to it, this are the NC headers that will be commented during this section.

In the next Figure 5.4 the headers could be easily seen:

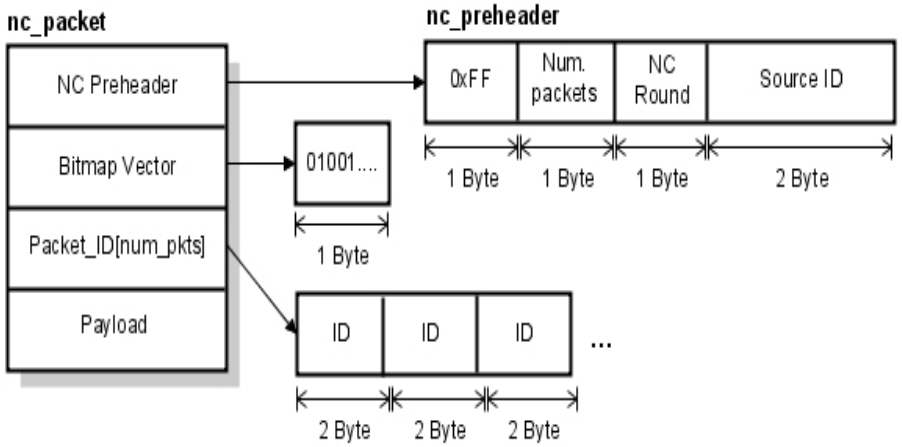


Figure 5.4: NC packet data structure

- NC pre header is one the NC mandatory header. As can be seen in Figure 5.4 the first byte is used as a key to identify the NC packets. The second one is a byte to indicate how many packets are coded within. The third indicates the NC exchange round of the packet, to avoid interferences between different data sets. And the last one contains the information about the source node that has send the packet.

- Reception vector contains the information about the packet that a node holds in the local packet pool, this header is also mandatory for any NC transmission. Only formed by a byte, it contains the bitmap of the local vector to allow the remote nodes to know which packets this node have or not.

- List of packet_id that are contained in the payload. This header is no mandatory, since in some cases empty NC packets are sent to update the reception vectors of all the nodes. If some IP packets are coded together in a NC packet, 2 bytes per packet should be added as a header to allow the other nodes know which packets are contained.

- Payload, three possible payloads of a NC packet are admitted, a normal IP packet (with IP header and UDP header), an XOR'ed packet, or empty if no packet has been sent.

5.2.3 Synchronization protocol

As we explained in chapter 4.1.2 the NC framework will only work with data sets of less than 255 packets, if we want to send sets with more than 255 packets, some restarts on the NC framework are needed. As the NC community works as a distributed system, some synchronization have to be realized to avoid desynchronization between the different nodes.

The typical desynchronization problem on the system is that one of the nodes finish before the others, and without waiting for the other nodes to finish, VA injects the new packets and NC begin the transmission of a new set of data. As the other are not in the same round that this node, the new broadcasted packets affects the throughput of the system.

To avoid this kind of problems a synchronization protocol has been implemented, forcing all the devices to wait until all the nodes in the cluster have completed the actual exchange round before restarting the NC framework. In the current implementation only the COMPLETE message has been implemented in the protocol, but if in the future is needed, more messages can be easily added to perform more synchronization or control tasks.

5.2.3.1 Status vector

As we need to save the status of all the nodes that take part in this exchange, we need to create a vector to hold the information about the status of all nodes. For this purpose the complete vector storage is used (see Figure 5.5), this vector contains one integer value per node, allowing the application to see if all the devices have already finish or not. The use of an integer value as status per node is chosen to allow future development to this functionalities, a future improvement could be hold in realtime the ncmode status of each node during the exchange.

Figure 5.5: Complete Vector data structure

Like the other data structures this also offers the NC framework some management functionalities, as init, shutdown, getters and setters for the node status, free a vector, and so on.

5.2.3.2 SYNC Packet headers

When a SYNC packet needs to be send through the medium a special header has to be added to it, basically a short header is attached, with some mandatory parameters, in Figure 5.6 a packet diagram is shown.

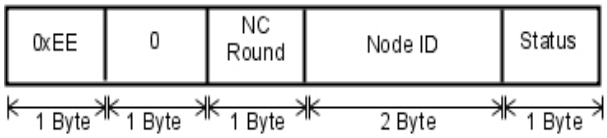


Figure 5.6: Synchronization packet data structure

As we can see the first byte is used as a key to identify the NC packets, the second parameter is an empty byte, the third one indicates in which NC exchange round the packet was sent and the last parameter in the header contains the information about the source node that has sent the packet.

5.2.4 VA to NC protocol

As we commented in the previous section the NC framework is remotely controlled by the VA, so a control protocol should be designed for this purpose. At this moment five different messages have been defined, all of them share the same header structure, modifying some attributes to identify each other. This headers are always attached to an UDP packet.

- Start packet (4 bytes)

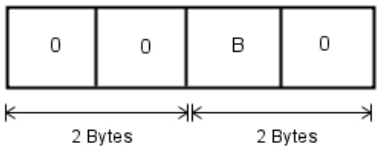


Figure 5.7: START packet data structure

This packet is used to start the exchange between the nodes, when the NC framework gets this message, it change the ncmode from READY to RUNNING. As we see the headers are quite simple, only 4 bytes are used, and only the third byte contains valid information, a 'B' which identifies the message as a START.

- Stop packet (4 bytes)

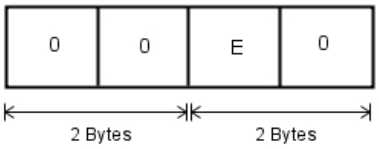


Figure 5.8: STOP packet data structure

This packet is used to stop the exchange between the nodes, when the NC framework gets this message, it change the ncmode from any state to STOPPED. As we see the headers are exactly as the start packets, only 4 bytes are used, and only the third byte contains valid information, a 'E' which identifies the message as a STOP.

- Restart packet (4 bytes)

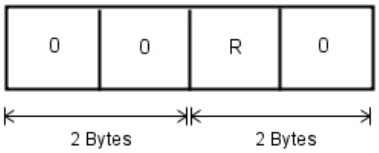


Figure 5.9: RESTART packet data structure

This packet is used to stop the exchange between the nodes, when the NC framework gets this message, it change the ncmode from any state to STOPPED. As we see the headers are exactly as the start and stop packets, only 4 bytes are used, and only the third byte contains valid information, an 'R' which identifies the message as a RESTART.

- Initialization packet (10 bytes)

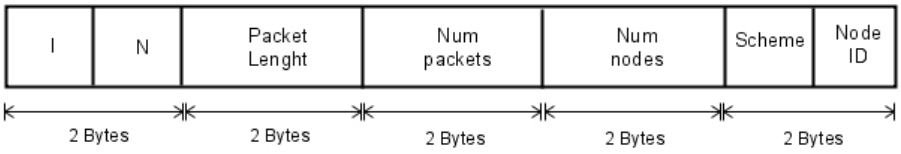


Figure 5.10: Initialization packet data structure

This packet is sent each time that the NC has to be initialized, which commonly happens at the beginning of each exchange round.

As we can see the header begin with two bytes that indicate the kind of packet, in this case the value is 'IN', two bytes more are used to specify the length of the packets exchanged, next 2 bytes for the number of packets, 2 bytes for the number of nodes that are taking part in the exchange, 1 byte for the scheme used, and another one specifying the node identifier.

- Inject packet (4 bytes + payload)

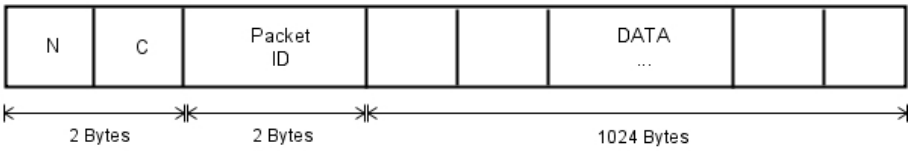


Figure 5.11: Injected packet data structure

This packet is received any time that the VA wants to inject information to the NC framework. As we see the first two bytes are used to identify the kind of packet. In this case the identifier is 'NC'. Two next bytes are used to give the packet an id number and after this the data payload is attached.

As we commented in section 5.2.1 the NC framework is externally called by a thread from VA, now we will explain with more detail the functionalities that each thread offer in the NC framework.

5.2.5 Handle from tun thread

This thread is the responsible to get and process all the messages that were shown in the previous section, depending on the message received it will perform one activity or another.

This function enters in a while loop that only finish when the ncmode is set to STOPPED. Inside the loop the thread has to wait until some information is received, to achieve this blocking opertation the socket is managed through the *select()* function. Once information has been received the data is directly sent to the *receive_from_tun* function, which parses the content like the next diagram:

```
1  tx_size = tun_receive(tx_buffer, BUFSIZE);
2  if (tx_size < 0) {
3          error();
4  }
5  if (check_for_start(tx_buffer)) {
6          if (initialized)
7                  setncmode(RUNNING);
8          return;
9  }
10 if (check_for_restart(tx_buffer)) {
```

```
11          reset_variables();
12          setncmode(READY);
13          nc_round++;
14          return;
15  }
16  if (check_for_stop(tx_buffer)) {
17          setncmode(STOPPED);
18          nc_shutdown();
19          return;
20  }
21  if (check_for_init_packet(tx_buffer)) {
22          return;
23  }
24  if (check_for_injection(tx_buffer)) {
25          if (!initialized)
26                  return;
27          pp_packet_t *pp_packet = new_empty_pp_packet();
28          memcpy(&(pp_packet->id), &(tx_buffer[28]), sizeof(uint16_t
               ));
29          pp_packet->ip_packet = buffer2ip(tx_buffer);
30          add_pp_packet_to_pool(pp_packet);
31          nc_put_packet_status(getncnodeip(), pp_packet->id, 1);
32          free_pp_packet(&pp_packet);
33          return;
34  }
35  return;
```

The function process the data from the received packet, and tries to determine the type of packet received, depending on the packet, different actions are taken.

For INIT packets the NC framework is setup with the given parameters, START packets modifies the ncmode to RUNNING and the exchange begins, for RESTART packets all the NC framework is restarted setting the default parameters, freeing the data structures and setting the ncmode to READY. The last packet is the INJECTION one, the data received is saved as a *pp_packet*, as we commented in section 5.2.2.1. This *pp_packet* is saved in the packet pool, and after that is marked as available in the status vector.

5.2.6 Handle socket receive thread

This thread has to process the received packets from the MAC layer. Basically, this function get a packet, decode it, and if is necessary the ip packet is directly sent to the upper layer VA. As we said in section 5.2.3, the NC framework also uses a synchronization protocol to provide better synchronization during the multiple exchange rounds, so this function has to be capable of manage two different kinds of packets, the NC_PACKETS and the SYNC_PACKETS, and each kind of packet will have its own behaviour.

As the previous thread, it enters in a while loop that will finish when the ncmode has been set to STOPPED. Inside the loop the thread waits until information is received in the

socket. Once the information is received the data is passed to the *receive_from_socket*, function which will behave like this:

This is the logic for a NC packet:

```
1  if(NC_PACKET){
2          nc_packet_t *nc_p = buffer2nc(rx_buffer);
3          // If is from this round
4          if (nc_packet->nc_preheader.nc_round == nc_round) {
5                  // If has payload
6                  if (nc_packet->nc_preheader.num_pkts > 0) {
7                           // Decodes the packet
8                          int num_packets = decode_nc_packet(
                              nc_packet, &tx_buffer_ptr);
9
10                         If(num_packets == 1){
11                                 // Sends the packet to an upper
                                       layer
12                                 parse_packet_and_save(&(
                                       tx_buffer_ptr[28]));
13                         }
14                 }
15                 // Update the packet vector
16                 handle_header(nc_packet);
17                  // No packets remaining and not COMPLETE
18                 if (!nc_is_complete(getnodeid()) && getncmode() !=
                       COMPLETE) {
19                         // Update the status vector
20                         nc_put_node_status(getnodeid(), 1);
21                         if (nc_get_vector_status())
22                                 // Set the ncmode to ALLCOMPLETE
23                                 setncmode(ALLCOMPLETE);
24                         else
25                                 // Set the ncmode to COMPLETE
26                                 setncmode(COMPLETE);
27                 }
28         }
29 }
```

The logical of this function has been simplified to allow a better understanding. The first check is to know if the basic NC headers are correctly set, if this check is passed, the function processes the received buffer and creates a new data structure with all the contained information. If there is some payload and the nc_round is valid, then the function will try to extract a the payload information using the decoding function. If there's no valid information for us (because we cannot decode it or because it was an update vector packet) the NC framework always tries to update the packet status vector, and the complete vector, to have them as much updated as possible. If a unknown packet have been received, it will be send to the VA.

For the case of a SYNC packet the function will behave like this:

```
if(SYNC_PACKET){
        sync_packet_t *sync_p = buffer2sync(rx_buffer);
        // If is from this round
        if (sync_packet->sync_preheader.nc_round == nc_round) {
                handle_complete_header(nc_packet);
        if (nc_get_vector_status())
                // Set the ncmode to ALLCOMPLETE
                setncmode(ALLCOMPLETE);
        else
                // Set the ncmode to COMPLETE
                setncmode(COMPLETE);
}
```

The case of the SYNC packet is much simpler since we only decode the message if the headers are correctly set. Also the NC round is checked and if everything is correct the NC framework updates the complete vector for the node that has send the message.

5.2.7 Handle socket send thread

This thread sends all the packets exchanged through the NC framework. That, includes the NC packets and also de SYNC packets. Therefore it have to create the packet structure with all the headers, choose the desired packet using the corresponding scheme (schemes will be shown in the next section), encode the packet, and send it through the medium.

As usual in the NC framework threads the it enters in a while loop that only finish when the ncmode is set to STOPPED, so it will never end until the whole application has been ended.

```
while (getncmode() == READY) {
        wait();
}
while (getncmode() == RUNNING) {
        nc_send();
        usleep(rand()%100);
}
while (getncmode() == COMPLETE) {
        if (count >= 1) {
                usleep(rand()%100);
                nc_send_complete();
                usleep(100);
                count--;
        }
        usleep(rand()%100);
        nc_send();
        usleep(1000);
```

```
18 }
19 while (getncmode() == ALLCOMPLETE) {
20         if (count >= 1) {
21                 usleep(rand()%100);
22                 nc_send_complete();
23                 count--;
24         }
25         usleep(5000);
```

As we can see in the schema the loop always perform the same logic and follows the usual behaviour of the ncmode state machine. During the READY status there are not tasks assigned so this thread so it waits until a new mode is set. Once the state has change to RUNNING the thread tries to send packets to all the nodes, once each node has got all the packets, they will change the ncmode to COMPLETE, and apart from the exchange of data packets, the SYNC packets, the number of this packets is limited to avoid unnecessary congestion on the network, they should be send during this phase, to advice the other neighbors that this node already has all the packets and is waiting for the other nodes to restart. When all the nodes have all the packets, then the state machine of them is changed to ALLCOMPLETE, and if still any SYNC packet has to be send will be send during this last state.

5.3 Schemas

An schema is the algorithm that choices which packet has to be sent in each moment, so depending on the final application, the equipment or the needed power consumption it's important to choice one or another schema. During this section we will speak about the designed schema and the benefits of each of them.

To estimate which is the packet that has to be send, we need to know which packets do we have and which packets have our neighbors, this could be easily achieve using the data structures showed in section 5.2.2, with all the information available the schema selects which packet has to be send and it is encoded if it is needed.

5.3.1 Broadcast first then NC Schema

The broadcast first then NC schema tries to be a hybrid schema between the broadcast schema and the fully NC schema. Ideally gets the best of each schema, so is possible to reduce the power consumption and also a better throughput using XOR coding techniques to retransmit packets.

The behaviour of this schema is quite simple, it consists of two different phases, during the first phase the schema behaves as a simply broadcast schema, just gets the first packet not send before from the packet pool and broadcast it without using any NC encoding technique. During the second phase (retransmission phase), the behaviour changes and uses the NC techniques to do specific retransmission to the end users.

With this schema we try to achieve two different objectives, the first one is avoid the unnecessary use of coding techniques on an earlier phase. The use of this techniques causes a processing overhead that affects directly to the power consumption, avoiding this some energy could be saved. After the broadcast phase, the retransmission phase begins, right now the use of NC techniques is interesting, while doing packets retransmission over broadcast is not a good solution since probably not all the users will be interested in all the packets broadcasted. Then using NC to broadcast specified packets to the interested users could be a better solution as more than one packet could be send over the same transmission, reducing the number of retransmission that are send on the network. Therefore the power consumption, and also the network congestion will decrease.

5.3.1.1 Implementation

Any designed schema basically must obey the interface defined by the NC framework, this interface compels us to design two basic functions to encode a decode packets with some given parameters and return some specified values.

- int nc_scheme_snc_sbc_first_encode(unsigned char **buffer, uint16_st **pkt_ids)
- ip_packet_st* nc_scheme_nc_bc_first_decode(nc_packet_st *nc_spacket, int *pkt_sid)

There is also a third function that we've implemented for an external use but it was not specified in the schemas interface

- void reset_spackets_send(), that only reinitializes some values in each exchange round.

Encoding function

This function writes a buffer with the packet to be send, this packet could be an XOR´ed packet or a simple *ip_packet*, also this function returns an integer value with the number of packets coded within the returned buffer.

```
1  if(next_packet < 0){
2          next_packet = getnodeid();
3  }
4  // First phase: Broadcast directly the packets
5  if(packets_send < PHASE\_THRESHOLD){
6          get_ip_packet_from_pool(next_packet);
7          next_packet =  next_packet + getnodeid();
8          packets_send++;
9          return 1
10 }
```

In this first phase, just looks for the next available packet and broadcast it without checking if it is held by other nodes or not. When all the packets that were held at the beginning by this node has been send is the time to begin the retransmission for lost packets using the NC techniques.

```
1  //Second phase: Retransmission using XOR
2  //No more packets can be coded with or no packet can be found to
       be coded
3  while(ackets_to_code_with > 0 && max_number_of_iterations <
       getncpacketno()+1) {
4          //Loop searching the best packet to code with (The one
               that least user has)
5          best_packet = bf_find_next_packet(codingvector);
6          //Loop searching packets to code with the previous one
7          for (packetno=1; packetno<=getncpacketno(); packetno++) {
8                  packets_to_code_with +=bf_is_packet_in_vector(
                       codingvector,packetno);
9          }
10 }
11 // No packets to send
12 if(num_packets == 0){
13         //Last_packet variable to a random value
14         last_packet = random_packet();
15 }
16 // only one packet has been found / no coding posiblilities
17 if(num_packets == 1){
18         packet = get_ip_packet_from_pool(pkt_id[0]);
19 }
20 if(num_packets > 1){
21         packet = bf_XOR_found_packets(buffer, *pkt_ids,
               num_packets);
22 }
23 return num_packets;
```

In this second phase, the function tries to find the packet that less users have, and then tries to code it with other packets that other users will be interested in.

Decoding function

This function return all the known packets that can be decoded from the received packet, checks the pool for all the packets that are contained, determine the unknown packets and known, then XORs the received buffer with the known ones, to obtain the valid information from it. If the extracted packet is an unknown packet, it is returned to the NC framework, otherwise no packets have been decoded so a NULL value is returned to the NC framework.

```
1  // For all the packets coded inside the received packet
2  for (i=0; i<num_packets; i++) {
3          //if we have this packet_id
```

```
4           if(nc_get_packet_status(getncnodeip(), pkt_ids[i])) {
5                   dummy_p=(unsigned char*)get_ip_packet_from_pool(
                        pkt_ids[i]);
6           }
7           else {
8                   unknown_packet++;
9                   //save the ID of the unknow packet
10                  *pkt_id=pkt_ids[i];
11                  continue;
12          }
13  //XOR dummy with the received buffer
14  for (j=0; j<payloadsize; j++) {
15          buffer[j] = buffer[j] ^ dummy[j];
16  }
17  //free dummy packet
18  free(dummy_p);
19  // a packet has been found
20  if(unknown_packet == 1){
21          // return the XORed buffer
22          return (ip_packet_t*)buffer;
23  }
24  else{
25          return NULL;
26  }
```

As we see this function tries to decode a received *NC_packet*, iterating all the contained packets in the coded packet and elects the unknown and known packets, and tris to decode the packet, if is possible to decode returns the decoded *ip_packet*, if not returns a NULL pointer to avoid errors in upper level functions.

Chapter 6

Results

In this chapter the results obtained from the different tests performed to our application will be presented.

The main objective of our project was the creation of a video application demonstrator using the NC framework and implementing a new hybrid schema using broadcast and network coding techniques. Now is time to see the results achieved with this new schema and compare with the previous ones, and find out if the new schema developed performs as in the theoretical chapter it was either supposed or not.

The tests are performed using the architecture commented in section 4.1.2, and the equipment used for the test is the following:

- 4, 6, 8 Nokia N800 Internet Tablet Devices, depending on the needs of each test.
- 3 USB 2.0 HUB to connect all the devices with the streaming Server.
- 1 LG S1PRO Notebook to act as a media server with:
  - 2,5 GB DDR2 667Mhz Memory
  - Intel Core Duo T7200 2.0Ghz
  - Ubuntu Linux 8.04 LTS

The results obtained are mainly focused in general aspects for a multimedia transmission over a wireless link, in particular three basic aspects will be discussed:

- Throughput
- Time necessary to achieve a satisfactory playback
- Energy

6.1 Throughput Results

In this test, throughput has been measured as the total number of packets available on the nodes as a function of time since the start of the test. Furthermore this can be used to measure how long it takes to distribute the data among the nodes.

As it was explained before, broadcast schema is effective at the beginning since all nodes are interested in packets belonging to other nodes. This is because the server splits the video in as much fractions as nodes in the network and gives a different one to each one, so correlation between available packets on the nodes is null. But when retransmitting packets that were lost in the first transmission NC is much more efficient, coding packets together before sending them and diminishing thus the number of necessary transmissions. Therefore, what we are theorically expecting is to improve the throughput using the combination of broadcast and network coding.

To have a record of available packets on the nodes at every moment a log file is updated each time that a packet is received. Also information about sent packets is stored.

6.1.1 Packet logger

The Log is initialized during the TA initialization and closed during shut-down of the program. The Log is saved to a file named nc.log, which can be retrieved for later analysis.

The interface for writing to the Log is the function *write_log_packet(int type, int packet)*, which takes the type of event and a number, e.g. packet number, as arguments. Each event is timestamped to allow examination of the test results as a function of time. The timestamp is generated from functions in the standard library sys/time.h, which is available in most Linux distributions. Each timestamp signifies the number of seconds since January 1st 1970.

An excerpt from a log file is shown next:

```
1        1229923034.216125       1       89
2        1229923034.219085       1       37
3        1229923034.220123       1       6
4        1229923034.222717       2       0
5        1229923034.246765       2       0
6        1229923034.267974       1       11
```

The first column is a time stamp, followed by a number specifying the type of event. The following types are used:

- 1 - Log received packet, the second number specifies the packet id of the received packet,
- 2 - Log sent packet, the second number specifies the packet id of the sent packet.

From this file we are able to analyze the throughput and extract some graphs commented below.

6.1.2 Results

To reduce random differences among the nodes, the results are averaged. The result is normalized to show the number of packets in percent of the total set.

Next table shows the elapsed time until the nodes get all the packets and the corresponding throughput.

Num. Devices	NC		BC		BC+NC	
	Time (s.)	Throughput (kbps)	Time (s.)	Throughput (kbps)	Time (s.)	Throughput (kbps)
4	191.56	502.57	124.33	774.32	116.55	826.01
6	269.09	357.77	129.04	746.06	117.17	821.64
8	275.61	349.3	150.44	639.93	118.63	811.53

Table 6.1: Elapsed time to transmit the video and Throughput obtained for the different scenarios

As it can be seen in Table 6.1 throughput is improved using the new schema (broadcast combined with NC), as we expected. On the other hand, it can be seen how the transmission time increases when the number of nodes in the cluster is increased. The reason for having this performance could be the number of packets available to the nodes in the different scenarios. As it can be seen in Figure 6.1 in the four nodes scenario each one gets 1/4 of the total set of packets, in the six nodes scenario each one gets 1/6 of these packets and in the 8 nodes scenario each one gets 1/8 of the packets. So, the number of packets that the nodes have to get from their neighbors in the cluster increases with the number of nodes, being this slower than getting these packets from the server. Also having more nodes on the network suppose more packets being sent and, therefore, more collisions and consequently more packets lost.

Figure 6.1: Amount of data available from the beginning on the nodes in 4, 6 and 8 nodes scenarios

The time data in Table 6.1 was measured since the first received packet was recorded on the log until the last reception. As it was explained before, the video is exchanged in sets of data, needing to restart the Network coding framework between transmissions. Therefore, time used for restart is included in these data, and it can not be ignored because it is quite long. It can be seen in Figure 6.2, which shows the percentage of available packets on nodes as a funtion of time during a complete distribution of the whole video and shows the fact that these periods of time, while NC is restarting,

represents a long part of the total transmission time. This is one of the points that as future work should be improved trying to diminish the restarting time.

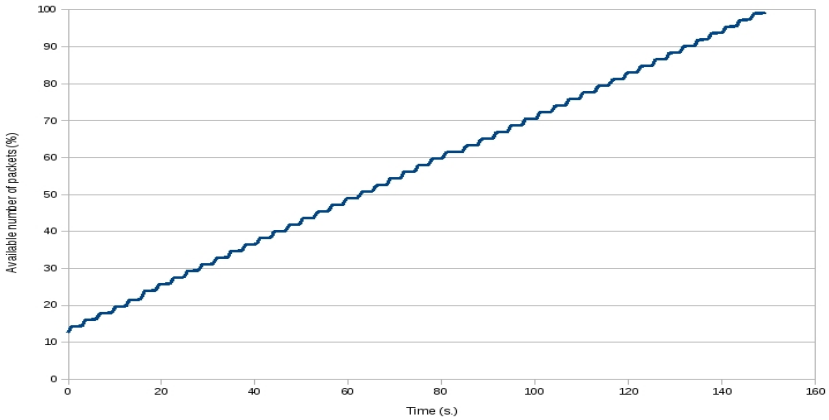


Figure 6.2: Average number of packets available on the nodes as a function of time during the distribution of the whole video.

These scenarios favour BC schema because each packet is only available in only one node and, therefore, one broadcast should satisfy all the other nodes in the network (assuming no transmission errors). On the other hand, coding opportunities are scarce, especially in the beginning, again because the data is disjointed.

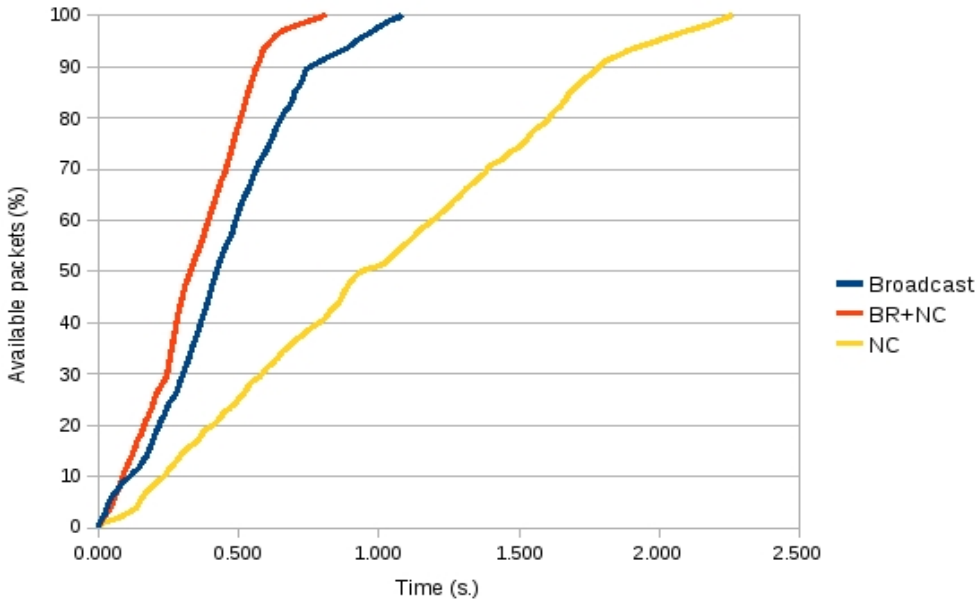


Figure 6.3: Average number of packets available on the nodes as a function of time when using Broadcast, NC or the combination of both in a four nodes scenario

After broadcasting all the packets, there is a lot of overlap between he packets on each node. Because of this overlap, many opportunities for coding packets together exist, which should give NC an advantage. Because of that, we use NC to retransmit packets. Figure 6.3 and Figure 6.4 show the number of packets available on the nodes as a function

of time since the start of the test and until a complete set of data (240 packets) is distributed on the four and eight nodes scenario respectively, since these are the most representatives.

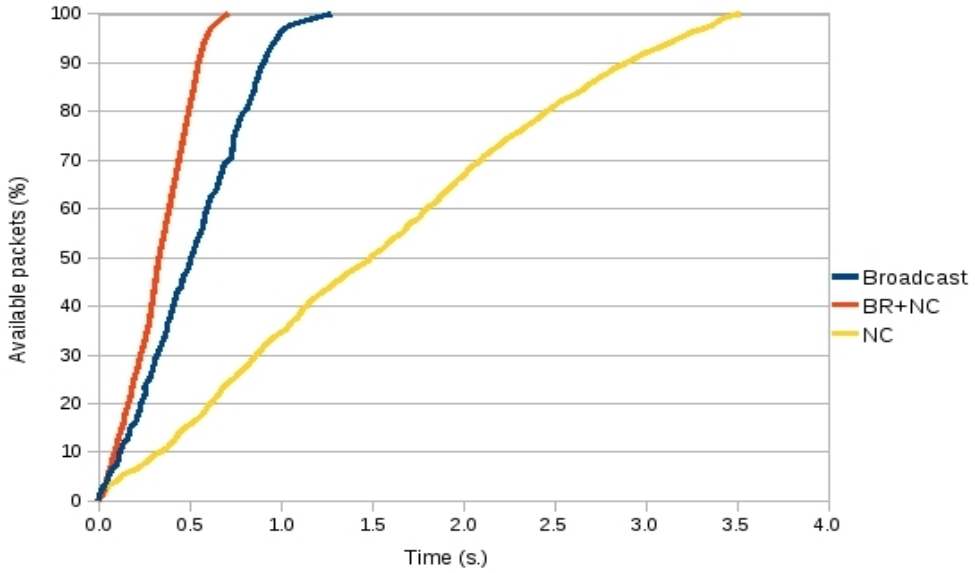


Figure 6.4: Average number of packets available on the nodes as a function of time when using Broadcast, NC or the combination of both in a eight nodes scenario

First of all it can be seen in Figure 6.3 and Figure 6.4 that broadcast is considerably faster than NC. Even thought, as explained before, this scenario favours BC, NC should have a better performance. The reason for not having this performance is that coding opportunities are limited by the CPU of the devices, since no more than two packets are coded together to simplify the process.

Furthermore, it can be seen, that the schema studied in this project combining broadcast and network coding is the faster one and the increase in the available number of packets is almost linear. From the server the packets are distributed following a predefined pattern and thus the algorithm to chose which packet has to be send is more efficient than in the other schemas.

Towards the end of the test, the reception rate drops somewhat in both, four and eight nodes scenario, being it more noticeable in the first one. The reason, in this case, could be that at the end of the test is more difficult to find packets to be coded together, also less nodes in the network means less coding opportunities. Broadcast curve drops too, and it is more noticeable in this case. This could be due to at the end of the test more than one node is able to transmit a given packet to the nodes needing it. This leads to increased lead on the medium and contention could pay a role. Furthermore each transmission can only satisfy one receiver, since only one node need each packet.

Therefore, using network coding at the end has the advantage that less transmissions are done, falling the load on the medium and the number of collisions between packets.

6.2 Energy Results

As we commented in introduction chapter, when new characteristics are added to mobile devices the power consumption increases. So this is a huge problem for mobile devices manufacturers that should be solved using different not fully covering solutions. Using a cooperative grid supported by Network Coding techniques could reduce seriously the power needed to obtain multimedia content from the operator network.

But also another of the goals was the design of a more energy efficient schema, which should reduce as much as possible the energy consumption during the cooperative exchange over the wireless network.

During this section we will show some practical results obtained during the tests, and with these results we will be able to compare if the theoretical benefits of the new implemented hybrid schema, explained in previous chapters, have been achieved.

The results of the new schema will be compared with a full broadcast (avoiding coding techniques), and also with a full network coding schema, and also using different number of devices, from 4 to 8 incrementing by 2 devices.

To get the results of energy consumption we have used the tool Nokia Energy Profiler that will be commented in next section.

6.2.1 Nokia Energy Profiler

Nokia Energy Profiler is a stand-alone test and measurement application for Maemo powered devices. The application allows developers to test and monitor their application's energy usage in real time in the target device.

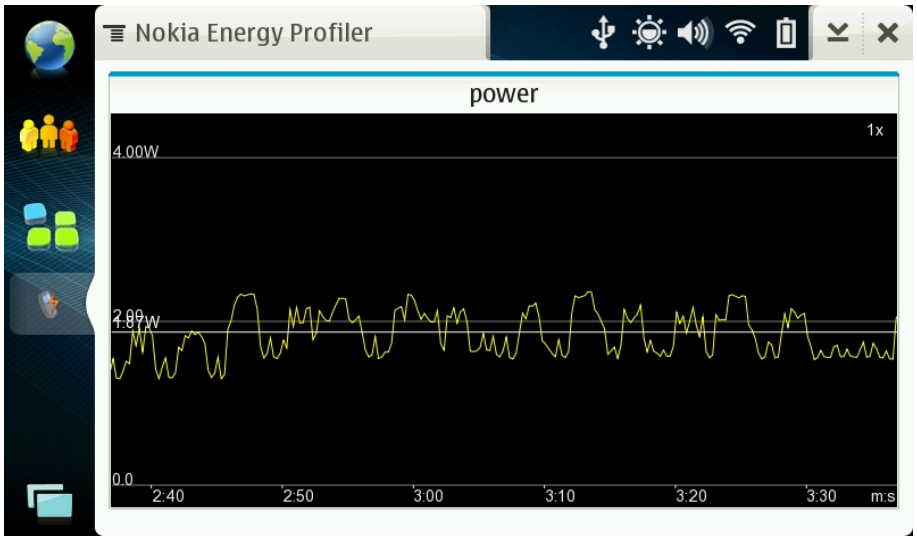


Figure 6.5: Nokia energy profiler

The application is available as several ".deb" packages for Maemo devices. It is possible to install with the package manager, which makes the installation process very simple.

Nokia Energy Profiler supports several measurement views. The version used for these tests has the following views:

- Instant Power (W)
- Instant current (A)
- Instant Voltage (V)
- Instant Temperature (°C)

Also offers some statistics values for each measurement performed. All the information logged by the Nokia Energy Profiler could be exported as a CSV file to be post processed by other applications like Microsoft Excel or similar.

6.2.2 Four devices scenario

During this section the results of the tests have been performed for a video exchange with 4 devices, which is the minimum number of devices that have been tested together.

For this exchange, each device receives from the streaming server approximately the 25% of the whole video, following the pattern defined in section 4.6, as the whole video is divided in 11752 packets of 1024 bytes, each device got directly 2938 different packets, and the 8814 remainder packets will be exchanged over the wireless network.

After the tests, results are obtained from the Nokia Energy Profiler; the data is exported to a CSV file and is post-processed with Microsoft Excel, obtaining different graphs per studied parameter.

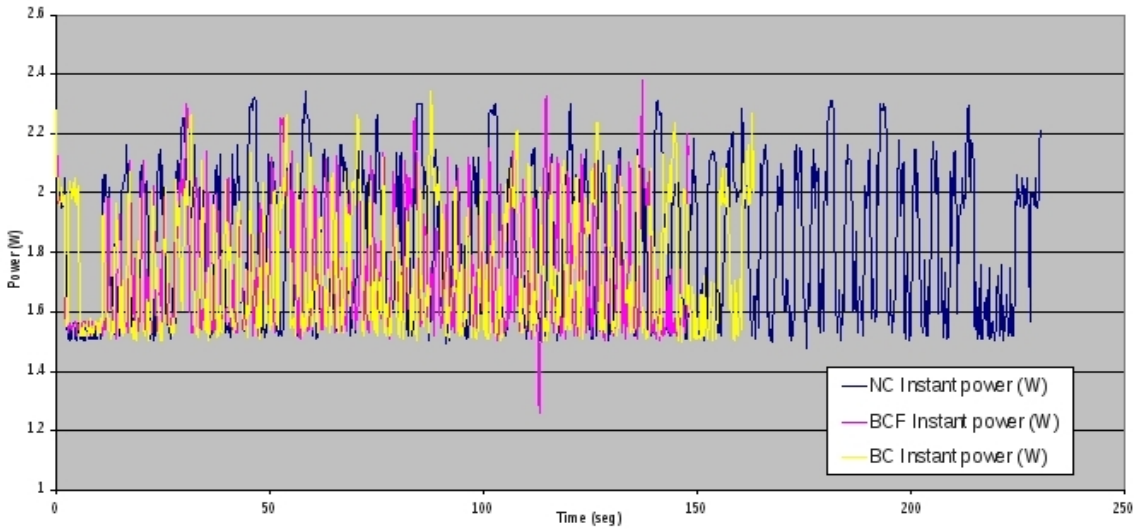


Figure 6.6: Instant power consumption in 4 devices exchange

In Figure 6.6 we can see the instant power consumption in Watts (W) of each mode, the NC (Network Coding schema) is represented by the blue color, pink represents BCF

(Broadcast First then Network Coding schema) and finally yellow is for the BC (Broadcast schema). At a first seen we can notice that the time used to perform a whole video transmission with the different schemas is different, this difference in time will affect seriously to the energy consumption results.

This differences could be cause of the higher processing rate demanded to the device, as the processing capacity is limited, the time needed to perform the demanded operations cause a delay, also another possibility could be the difference in the packet lose rate, but it does not seem a rational reason, as all the tests have been performed in the same environment conditions. The time employed by each schema can be seen in Table 6.2.

Schema	Time (s.)
NC	230
BCF	147
BC	163

Table 6.2: Time employed for a whole video playback

Another important parameter to be able to calculate the energy spent by a transmission is the average instant power consumption, to calculate this result is necessary to integrate the bottom area of each schema graph. The results obtained can be seen in Figure 6.7.Where we can be seen that the NC schema is the most power demanding with a value of 1.84W, next comes the BC with 1.73W, and the least instant consuming is the BCF schema with 1.72W.

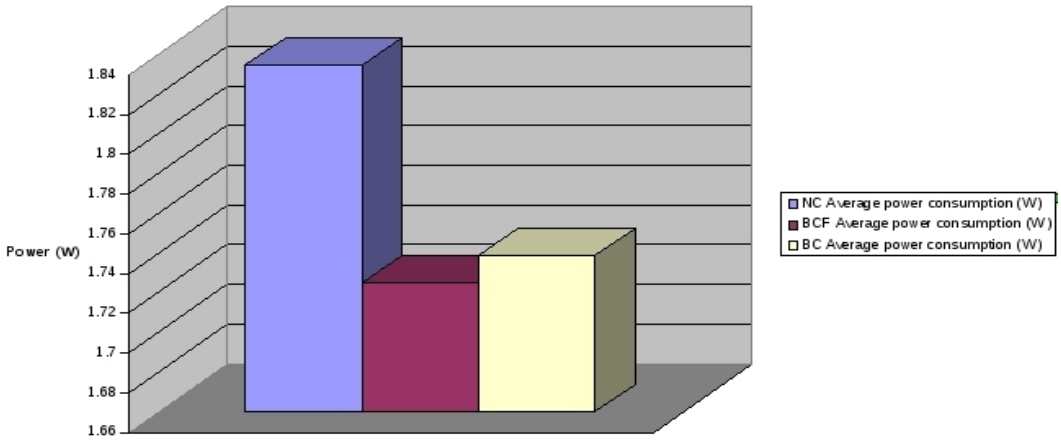


Figure 6.7: Average power consumption for 4 devices

Comparing the values between them we can observe that the BCF is a 6% more efficient than the NC schema, and 1% more efficient than BC. Comparing the BC with the NC, the BC is 5% more efficient than NC; this strange result is due the high requirements to code and decode a packet with the NC scheme.

With all this commented parameters (time spent and instant power) is possible to calculate the energy consumption per device which can be seen in Figure 6.8.

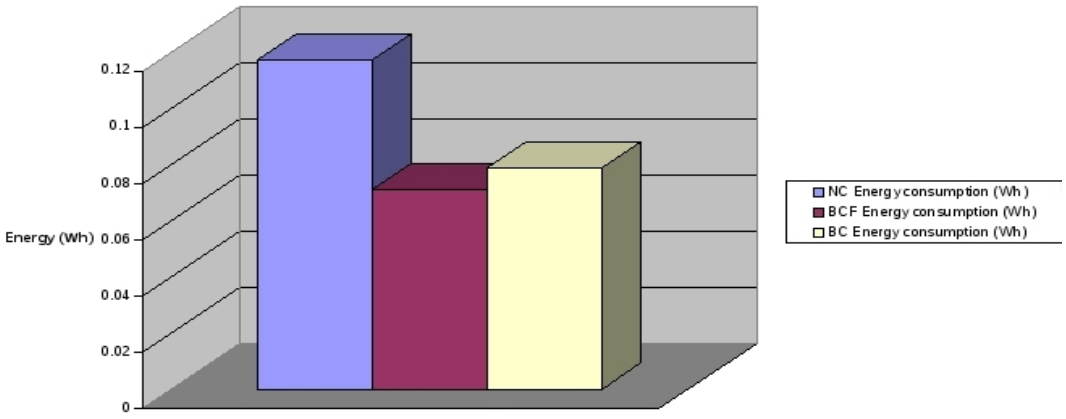


Figure 6.8: Energy consumption for 4 devices

These energy results are more significant, as we take in account the duration of the transmission and not only the instant power consumed. So the BCF schema has a benefit of 39% in front of NC and a 10% with BC. If we compare BC and NC the energy saved is approximately a 32% in favour of BC schema.

6.2.3 Six devices scenario

During this section the results for the 6 devices tests will be explained.

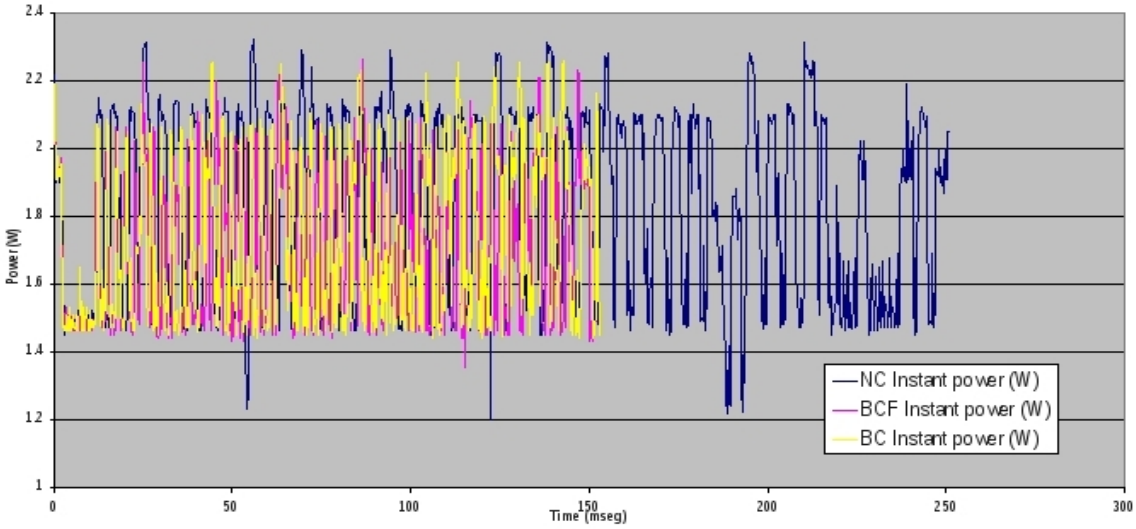


Figure 6.9: Instant power consumption in 6 devices exchange

For this exchange each device receives from the streaming server approximately the 16% of the whole video, as the whole video is divided in 11752 packets of 1024 bytes, each

device got directly 1958 different packets, and 9794 packets will be exchanged over the ad-hoc wireless network.

Figure 6.9 shows the instant power consumption of all the schemas for a 6 devices test.

The NC schema increments seriously the time to process all the packets compared with the 4 devices tests, and the BC and BCF schemas also increase but not in the same proportion of the NC schema. Table 6.3 shows the time used for a whole exchange by each schema.

Schema	Time (s.)
NC	251
BCF	153
BC	169

Table 6.3: Time used for a whole video playback

The average power consumption can be seen in Figure 6.10. Where we can be seen that the NC schema is still the most power demanding with a value of 1.82W, next comes the BC with 1.73W and the least instant consuming is the BCF schema with 1.70W.

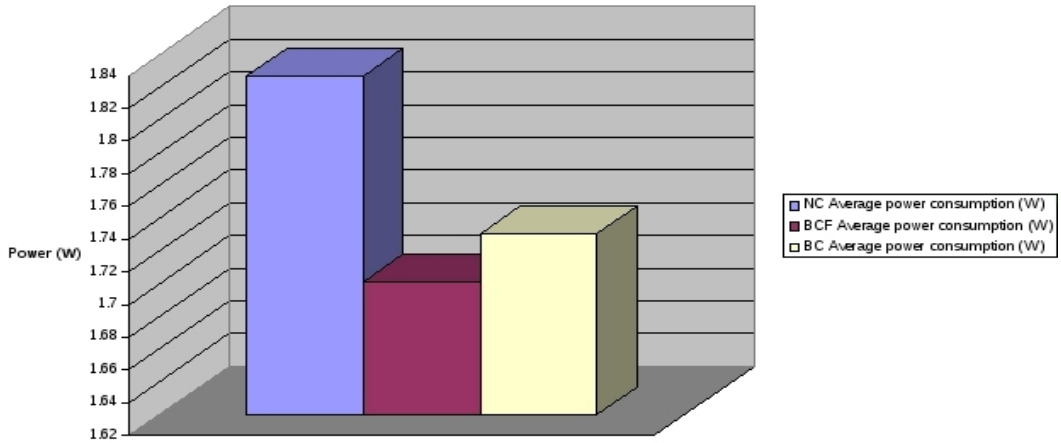


Figure 6.10: Average power consumption for 6 devices

Comparing the values between them we can observe that the BCF is a 6.9% more efficient than the NC schema, and 1.7% more efficient than BC. Comparing the BC with the NC, the BC is 5.2% more efficient than NC.

With all this parameters we will estimate the energy consumption per device which can be seen in Figure 6.11.

With this information it is possible to know that the BCF schema has a benefit of 43.4% in front of NC and a 11% with BC. If we compare BC and NC the energy saved is approximately a 36.4% in favour of BC schema.

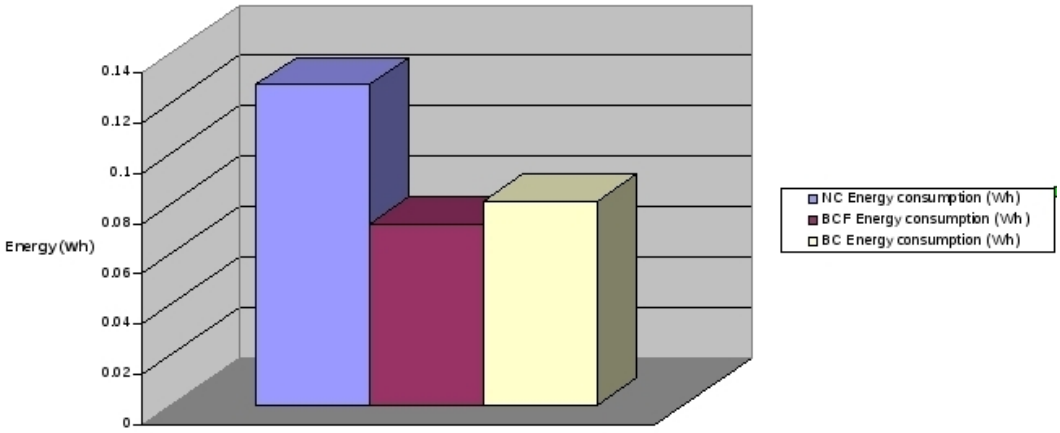


Figure 6.11: Energy consumption for 6 devices

6.2.4 Eight devices scenario

In this section the results for the 8 devices tests will be shown.

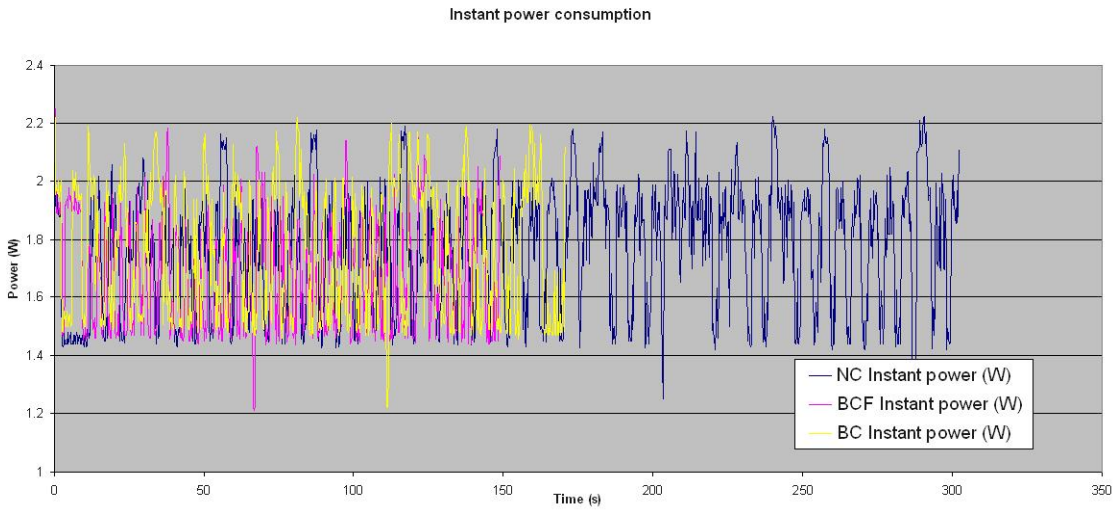


Figure 6.12: Instant power consumption in 8 devices exchange

For this exchange each device receives from the streaming server approximately the 12.5% of the whole video, each device got directly 1469 different packets, and 10283 packets will be exchanged over the air.

As we have seen in the previous test, Figure 6.12 shows the instant power consumption used by all the schemas during the test.

The pattern seen in the previous tests is followed, the NC schema increments seriously the time to process all the packets, and the BC and BCF schemas maintain approximately the values. The time spent by each schema can be seen in Table 6.4.

Schema	Time (s.)
NC	302
BCF	149
BC	171

Table 6.4: Time employed for a whole video playback

The power consumption results can be seen in Figure 6.13. Where we can see that the NC schema is the most power demanding with a value of 1.76W, next comes the BC with 1.66W, and the least instant consuming is the BCF schema with 1.75W.

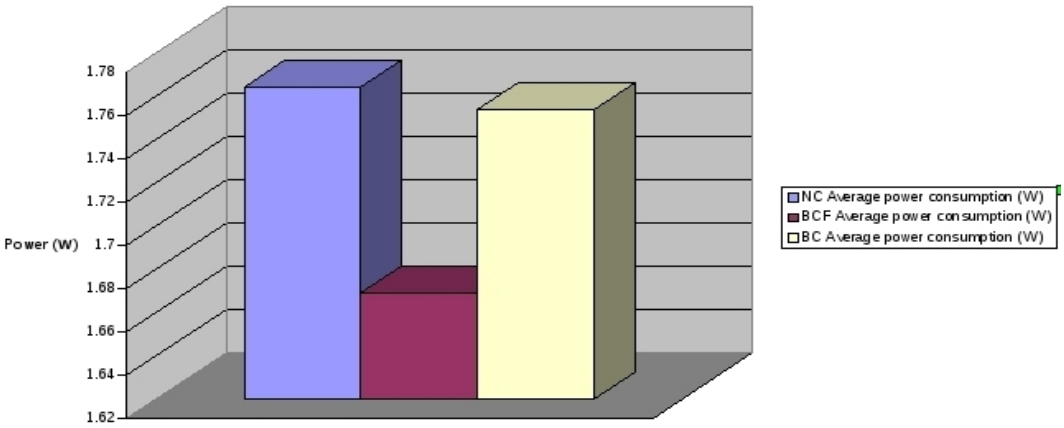


Figure 6.13: Average power consumption for 8 devices

Comparing the values between them we can observe that the BCF is a 5.3% more efficient than the NC schema, and 4.8% more efficient than BC. Comparing the BC with the NC, the BC is 0.5% more efficient than NC.

Figure 6.14 shows the energy consumption per device, where can be seen that the BCF schema has a benefit of 53.4% in front of NC and a 16.9% with BC. If we compare BC and NC the energy saved is approximately a 43.8% in favour of BC schema.

6.2.5 Results comparision

Once all the results of our tests have been shown, it's necessary to compare them and see the evolution of them while increasing the number of nodes.

After processing all the information obtained, we noticed than the most important parameters are the evolution of the time spent to play a video, and the overall energy spent are the significant parameters.

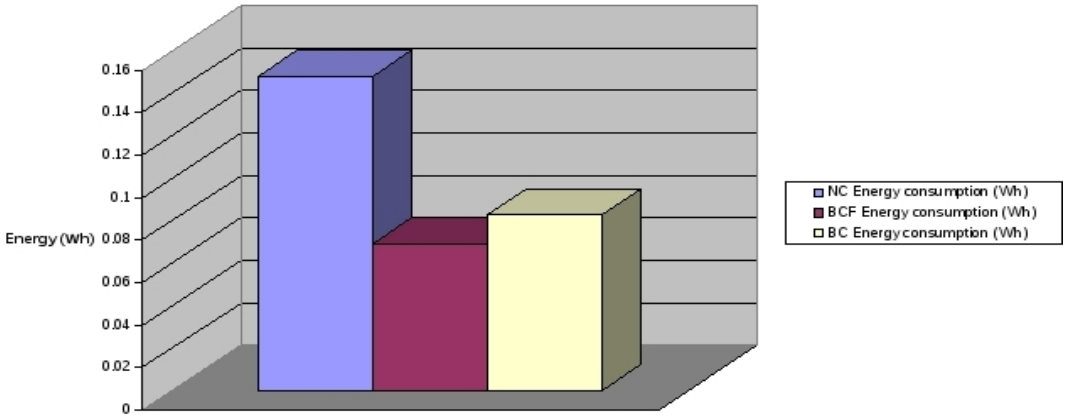


Figure 6.14: Energy consumption for 8 devices

6.2.5.1 Time evolution

This parameter is one of the most important for the energy consumption results, as an increase of the time to perform an exchange affects directly to the energy consumption results, in Figure 6.15 we can observe the evolution of the time, while increasing the number of nodes.

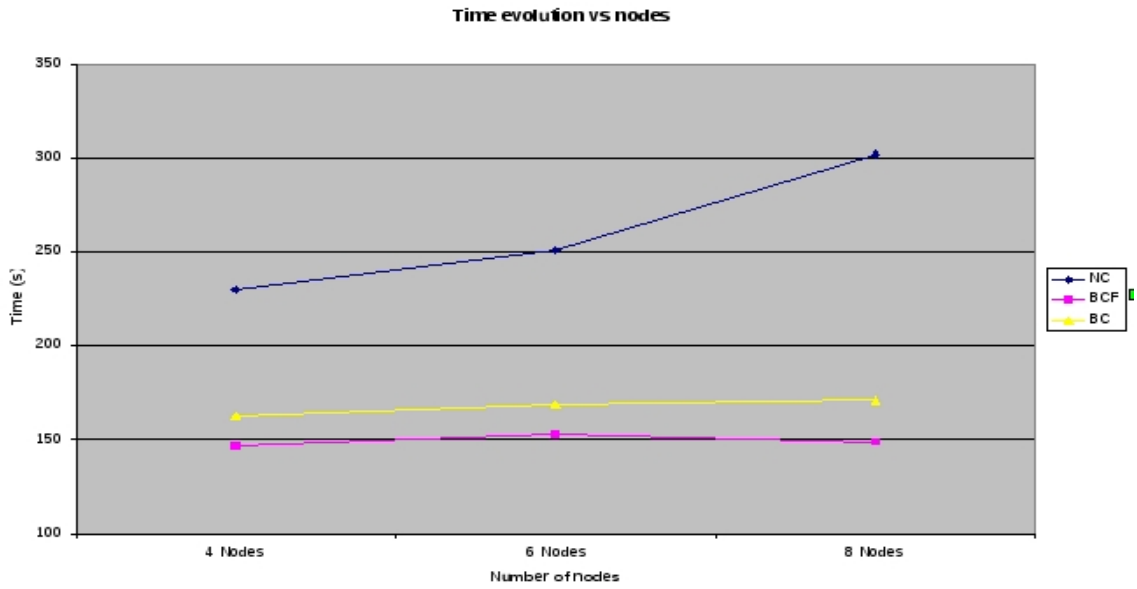


Figure 6.15: Time evolution vs nodes

As we can see in Fugure 6.15 the NC schema is the one that is more affected by the increase of the devices, as more devices are related in one exchange, the time to perform the whole transmission is increased in a non constant increment. That is due to the high processing requirements, as more devices are attached, more packets are shared over the

air, that also causes that more packets need to be coded and decoded; from the results obtained we know that the time to process a packet with NC is quite higher than a non coded packet, so it will be a bottleneck for the whole system, causing a higher packet loss rate.

As the NC framework uses UDP sockets, so no congestion avoidance algorithm is available, and also no reception packet queue is available on the application level, any packet received while decoding a previous one is lost, so the system at the end will perform more packet retransmissions, needing more time to perform the whole video exchange.

The BC schema follows a linear increment related with the increase of nodes that seems logical, as more devices are sharing the transmission medium, the probability of packet collision on the medium will be higher.

The BCF schema shows a constant behaviour, the time to perform a video playback maintains stable at a reference value of 150 seconds. As the playing time is kept constant while increasing the number of devices, this characteristic gives a very important advantage in front of the others schemas. As we will see it will affect directly on the energy consumption results that will be shown next.

6.2.5.2 Energy consumption evolution

One of the objectives was the evaluation of the energy consumption for the different schemas available; Figure 6.16 shows the results.

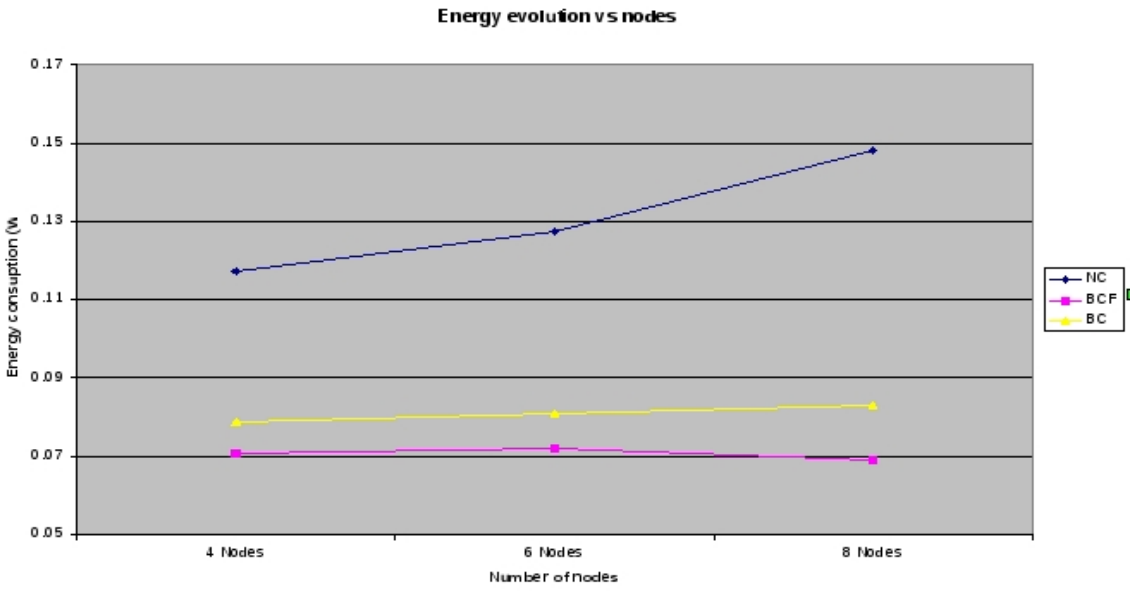


Figure 6.16: Energy consumption evolution vs nodes

As we can see the graph in Figure 6.16 is similar to the time evolution graph in Figure 6.15, showing the same non constant increment for the NC schema, the lineal behaviour

for the BC schema and the more or less constant behaviour of the BCF schema. With that result we can skip the average power consumption evolution as it is not very significant for the energy results.

6.2.5.3 Energy saving comparison

- BCF vs NC

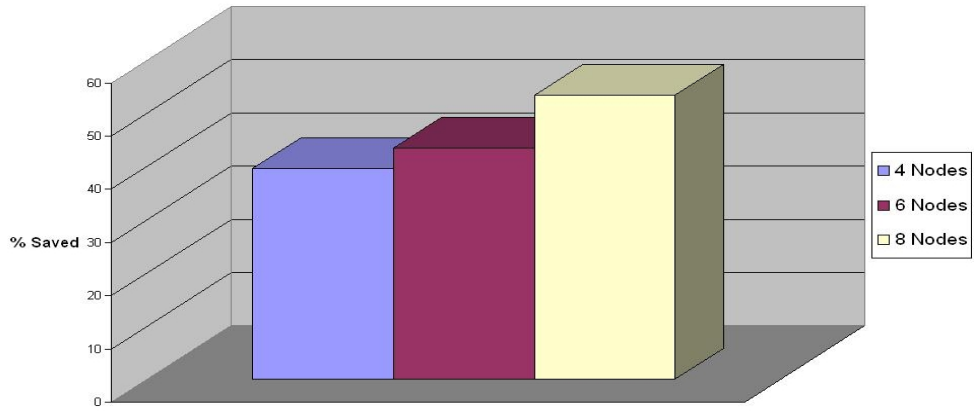


Figure 6.17: Energy saved BCF vs NC

Figure 6.17,comparison between BCF and NC, shows that increasing the number of devices in a video exchange causes also a direct energy saving increment, if the energy saved with four devices was about a 39%, for 6 devices is close to the 44%, and using 8 devices we are able to save a 53% more energy with the BCF schema. As more devices take part in the exchange the BCF is more efficient, saving more energy and improving the battery lifetime of the mobile device.

- BCF vs BC

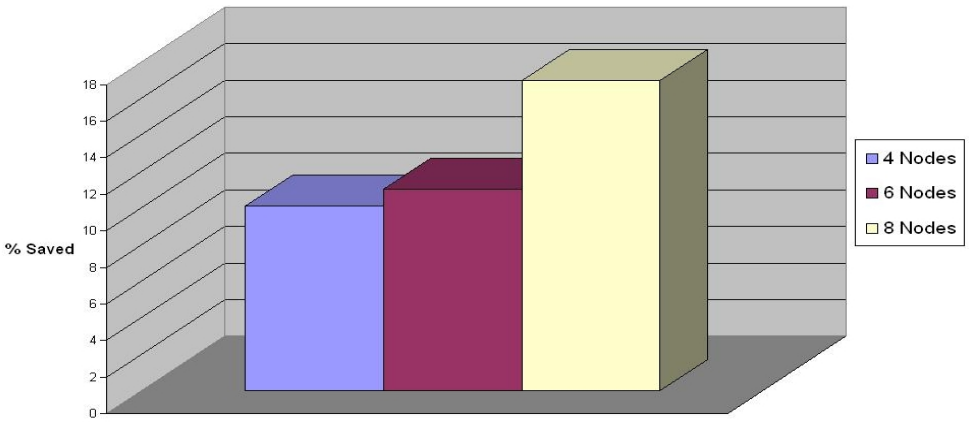


Figure 6.18: Energy saved BCF vs BC

In Figure 6.18 the BCF schema is compared with the the BC schema, as we can see the evolution from 4 to 8 devices is quite significant, improving from a 10% of energy saved using BCF for the 4 devices case, until a 17% for the 8 devices case. Using the BCF schema we achieve a better results as more devices are added to the grid, using the BCF schema packet retransmissions should be more efficient. With this results is possible to say that the new schema offers better scalability than the BC one.

- BC vs NC

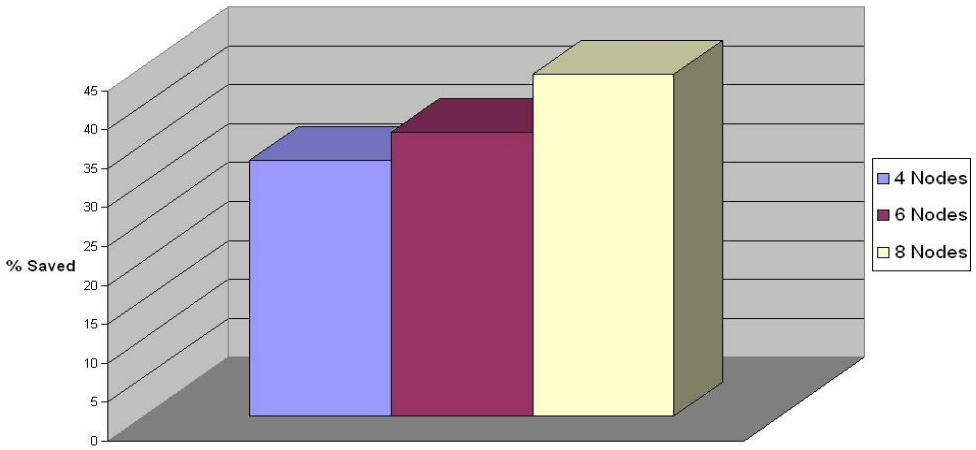


Figure 6.19: Energy saved BC vs NC

Finally Figure 6.19,comparison of the BC and NC schema, shows us one of the weaknesses of the network coding. As we can see the BC uses 34% less energy than the NC for 4 devices, and this increases until the 43% for the 8 devices test. With this results we can see that using a whole NC schema will affect seriously to the lifetime of the mobile device battery.

Chapter 7

Conclusions

This report documents the development of a demonstrator for a new hybrid schema combining broadcast and network coding, to compare it to both implementations separately. Network coding implementation is based on the work developed in [9] and Maemo is chosen as development platform for implementing the program.

Owing to how network coding was already implemented by [9] and the low processing capacity of the devices, it is not possible to exchange the whole video as a stream of data. So, the solution was to exchange the data in sets of a limited number of packets and restart the NC framekork after each exchange to be able to continue sending a new set of packets. This means that there is some time when the system is being restarted and no packets are sent. As we saw in the results chapter it represents a long part of the total transmission time, what have serious influence on the results obtained. This is one of the points that as future work should be improved trying to diminish the restarting time.

As we saw in the previous chapter the throughput obtained with the new hybrid schema is better than in the other cases. The worst result obtained is for the NC schema, when it should have at least as good performance as broadcast. It could be due to the fact that the implementation of NC used doesn't create enough coding opportinities to reduce the number of transmissions considerably. Broadcast schema gets almost as good results as the hybrid one, but it leads to increased lead on the medium. Therefore we can conclude that hybrid schema combining NC and broadcast has the best performance in this case.

As we observed in the energy results, the main factor for energy consumption evaluation is the duration of the video exchange. Instant consumption does not affect seriously to the final results since the average power consumption is similar in all the cases.

Newly developed hybrid schema has achieved its goal in terms of energy since the results are significantly better than using the BC and NC schemes. Moreover, increasing the number of nodes involved in the exchange, the results obtained improve. The new

scheme shows a constant energy consumption behaviour that does not vary depending on the number of nodes in the network.

These good results of the hybrid scheme are due to the lower computational requirements compared to the previous ones. The processing time is less than in other schemes, so the net throughput of the application improves and loss rate and energy consumption are reduced.

Broadcast schema seems a good solution for networks with a reduced number of nodes. The complexity of processing is lower and therefore the energy consumption as well, but on the other hand it has a worse performance during the retransmission of packets (with more transmissions than other schemes). It shows lower energy consumption than the Network Coding schema but still higher than the hybrid schema.

The NC schema does not seem a good outline for this kind of applications as in all cases observed consumption is much higher than the broadcast and that the hybrid scheme. Theoretically, this scheme should provide better results in networks with more nodes, because it should be more scalable. It should offer a more linear throughout during the transmission, but the results were otherwise noted, as more devices on the network the power consumption is higher. These bad results are due the low processing capacity of the devices, causing a distortion on the energy results. The fact is that the device is too slow to manage all the running processes, so it affects to the whole system causing the encoding and decoding of packets last longer. Also the video player lags during the playback because not enough resources are available, so finally the device needs more time to complete the exchange, and thus the device is active during more time the energy consumption is higher.

Bibliography

[1] The network coding home page, September 2003. http://http://www.ifp.uiuc.edu/~koetter/NWC/index.html.

[2] Esbox project page, December 2008. http://esbox.garage.maemo.org/.

[3] Extensible markup language (xml), October 2008. http://www.w3.org/XML/.

[4] Maemo.org: Maemo is the development platform for internet tablets, December 2008. http://www.maemo.org.

[5] Posix thread apis., December 2008. http://publib.boulder.ibm.com/iseries/v5r2/ic2924/index.htm?info/apis/rzah4mst.htm.

[6] The xml c parser and toolkit of gnome, December 2008. http://xmlsoft.org/.

[7] F.H.P Fitzek and editors M. Katz. *Cognitive Wireless Networks. Concepts, Methodologies and Visions Inspiring the Age of Enlightenment of Wireless Communications*, 2007.

[8] Marcos Katz Frank H.P. Fitzek and Qi Zhang. *Cellular Controlled Short-Range Communication for Cooperative P2P Networking*, January 2008.

[9] Morten Burchard Tychsen Jakob Sloth Nielsen, Karsten Fyhn Nielsen and Peter Ostergaard. *Performance Evaluation and Implementation of Network Coding on Mobile Devices using IEEE802.11 Technology*, June 2008.

[10] Shuo-Yen Robert Li Rudolf Ahlswede, N. Cai and Raymond Wai-Ho Yeung. *Network information flow*, July 2000.

[11] Wenjun Hu Dina Katabi Muriel M'edard Sachin Katti, Hariharan Rahul and Jon Crowcroft. *XORs in The Air: Practical Wireless Network Coding*, September 2006.

Appendix A

List of Abreviations

- NC – Network Coding Schema
- BC – Broadcast Schema
- BCF – Broadcast First Schema (Combination of BC and NC Schema)
- VA – Video Application
- FW – Framework

Appendix B

Demonstrator Setup Manual

B.1 List of Equipment

- Four Nokia N800 Internet Tablets
- One USB Hub (with at least 4 downlink ports, and 1 uplink port)
- A USB cable connecting the Hub and the PC server
- USB cables connecting the Internet Tablets to the Hub
- Power chargers for the Tablets

B.2 List of Peripherals

- A Personal Computer acting as the data server

B.3 Layout

Figure B.1 illustrates the layout of the devices. It is very important to connect the devices in the right order. Port numbers should be present on your USB hub, if not, then there are only 2 possibilites of the numbering. The green lights usually light up in the right order of numbering, when you connect the USB cable to the PC server. This could be a clue.

B.4 Prerequisites

Regarding the PC server:

- A Debian-based Linux

- kernel support for USB networking

Regarding the Nokia N810 devices: An internet connection (via an Access Point) is necessary to install the following packages.

- maemo-pc-connectivity
- rootsh
- wireless-tools
- mplayer

The following paragraphs will discuss how to install these packages.

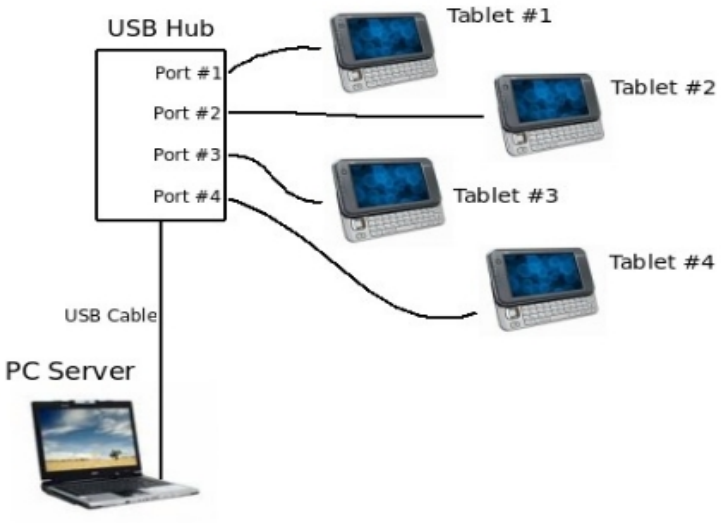


Figure B.1: Devices Layout

B.5 Installation Steps

- Switch On The Tablets

After switching on, you should choose a language, provide a device name, set the current date, time and location. Then you see the Hildon Desktop.

First of all, you should connect to the Internet using an Access Point. Click on the radio wave icon in the top right corner, and pick Connect. Then choose the network of your Access Point, and type in your Encryption Key when you are asked.

At this point, you should have Internet access on all of the Tablets. You could verify this by launching a Web Browser (Globe icon in the top left corner).

- Installing The Necessary Packages

The first package that you install will be the "maemo-pc-connectivity" package from maemo.org. It is really easy with a one-click installer. Just launch a Web Browser and type in the URL: http:// maemo.org/ development/documentation/pc_connectivity.

You will find the one-click installer of this package, just click on it. Answer yes-yes-yes to the questions popping up. It may take a while to complete the installation. You will be prompted to provide a root password for the SSH connection. It is wise to choose a short and simple password, because you will have to type it very often.At this point you should be able to SSH into your Tablets from your Linux box.

In the Application Manager click on: Title Bar -> Tools -> Application Catalogue... The list of repositories will pop up. There should a list item called "Maemo Extras", select it and click Edit. By default it is disabled, and it is not good for us. Please uncheck the Disabled checker, and hit OK.

Now we will add a new repository: click on the New button. Type in the following data to text-boxes:

```
1       Catalogue Name:              Diablo Tools
2       Web Address:                 http://repository.maemo.org/
3       Distribution:                diablo/tools
4       Components:                  free non-free
5       Disabled:                    Leave it unchecked!!!
```

Click the OK button. And close the Application Catalogue window. Now, you should see that it is refreshing the list of applications, it may take a while.

Click on the "Browse Installable Applications" button. First find rootsh, click Install, then find mplayer, click Install, then find wireless-tools, and Install.

B.6 Enabling USB Networking

After all the packages listed above are installed, the next step is to setup USB networking connecting the Tablets to the PC server.

First of all, the /etc/network/interfaces file has to be edited on the Tablets. You need root privileges to do this. So open a shell ono the tablets and type:

```
1         sudo gainroot
2         vi /etc/network/interfaces
```

The following section has to be inserted to the file on the tablets:

On Tablet No. 1:

```
1       auto usb0
2       iface usb0 inet static
3       address 192.168.2.10
4       netmask 255.255.255.0
5       gateway 192.168.2.1
```

On Tablet No. 2:

```
auto usb0
iface usb0 inet static
address 192.168.3.10
netmask 255.255.255.0
gateway 192.168.3.1
```

On Tablet No. 3:

```
auto usb0
iface usb0 inet static
address 192.168.4.10
netmask 255.255.255.0
gateway 192.168.4.1
```

On Tablet No. 4:

```
auto usb0
iface usb0 inet static
address 192.168.5.10
netmask 255.255.255.0
gateway 192.168.5.1
```

Please don't forget to save the files, after you have added the lines above!(type :wq to save and quit).

After this, you have to edit the /etc/network/interfaces file on the PC. You have to be root to do this. Please add the following lines to the end of the file:

```
    iface usb0 inet static
    address 192.168.2.1
    netmask 255.255.255.0
    network 192.168.2.0
    broadcast 192.168.2.255

    iface usb1 inet static
    address 192.168.3.1
    netmask 255.255.255.0
    network 192.168.3.0
    broadcast 192.168.3.255

    iface usb2 inet static
    address 192.168.4.1
    netmask 255.255.255.0
    network 192.168.4.0
    broadcast 192.168.4.255

```

```
19      iface usb3 inet static
20      address 192.168.5.1
21      netmask 255.255.255.0
22      network 192.168.5.0
23      broadcast 192.168.5.255
```

Now back to the Tablets click on Control panel -> USB Connectivity -> Insert USB Module.

Now, connect the USB Hub to any USB port of the PC server. After this, please connect Tablet #1 to the Slot #1 of the USB Hub. At this point, you should be able to ping 192.168.2.10 from the server with the following command using a normal terminal:

```
1       ping 192.168.2.10 -c 2
```

You should see the ping statistics almost instantaneously, but if it says: "Destination Host Unreachable" then something is not properly configured. The packet loss should be 0%, and the ping times should be below 20 ms.

If everything looks fine then:

- connect Tablet #2 to the Slot #2 of Hub, and run “ping 192.168.3.10” -c 2"
- connect Tablet #3 to the Slot #3 of Hub, and run “ping 192.168.4.10” -c 2"
- connect Tablet #4 to the Slot #4 of Hub, and run “ping 192.168.5.10” -c 2"

If the USB networking is up, then you may disconnect the Tablets from Access Point. This could be done by clicking on the radio wave icon in the top right corner, choosing Disconnect, and answering Yes.

From now on, you can SSH into the Tablets over USB. So, it is time to copy the executable file “nctest2“ and the configuration file ”conf.xml“ into the tablets. After that, edit the file conf.xml on all the tablets using the command:

```
1         vi /root/conf.xml
```

Be sure that ”number of devices“ is set to 4 and ”server address“ is set to 192.168.X.1 where X is the tablet number (1, 2, 3 or 4).

B.7 Setting Up the Ad-Hoc Wireless Network

On tablet #1 click on the wifi icon -> Connectivity settings -> Connections -> New, name the connection “N800” and select ad-hoc mode. Set the next configuration:

- IP: 10.1.1.1

- Subnet: 255.255.255.0
- Router: 10.1.1.100

Open the Main Menu -> Settings -> Connection Manager on Tablet #1. Click the Select Connection button, and a network named "N800" should be visible in the list. Connect to this network. Wait a few seconds until it says “Connected to N800”.

Then open up the Connection Manager on the other 3 Tablets, and connect manually to this “N800” network. Configure the IPs as in the previous case:

- IP: 10.1.1.X where X is the tablet number (2, 3 or 4)
- Subnet: 255.255.255.0
- Router: 10.1.1.100

B.8 Running the Application

You should launch the server first. It is located in the "Streamer" directory, so just navigate there in another terminal, and type:

```
1 ./streamer
```

To run the clients on the Tablets simultaneously, just open a SSH terminal for each tablet (you can use a SSH cluster), navigate to the /root directory and type:

```
1 sh nctest2 X //where X is the tablet number (1, 2, 3 or 4)
    .
```

You don’t have to launch mplayer separately, the application will automatically launch it, and instruct it to play the video as soon as enough data arrives from the server. The video playback should start in about 7 seconds on all the 4 Tablets almost simultaneously.