Design and evaluation of network coding in a P2P VoD system

Thibault Gouala

Design and Evaluation of Network Coding in a P2P-VoD System

Research work report

Thibault Gouala

Master 2 Computing Research 2009-2010
Internship at the State Key Networking & Switching Lab of the Beijing University of Posts and Telecommunications – China
Under the responsibility of Shiduan Cheng and Hongbo Wang

Abstract:

To measure the impact of Network Coding on classical P2P systems, we have designed and added a Network Coding module in an existing P2P-VoD system called PPlayer. As we worked in a micro environment we got more focused on the influence of Network Coding on individual peer rather than on the overall system throughput. However, the tests performed in our laboratory have highlighted some advantages. With equal performances, less data are sent on the network and slow peers can help to provide data. But Network Coding implies a heavy computing complexity which is an obstacle in the deployment of Network Coding at a large scale. To deal with it we propose two solutions: the generation of half decoded blocks and a buffering system allowing a peer to provide more peer at the same time.

Summary

1 INTRODUCTION

Over the past few years there has been a large expand of video streaming on internet. In 2005 there was a strong belief among companies that this market will expand exponentially in the next few years and this is happening nowadays. A lot of events, in particular sport events, are broadcasting and then retransmitted on internet. The Olympic Game of 2008 in China and the Football World Cup of 2010 in South Africa are two examples of events widely watched on the internet.

These media contents can be accessed through VoD services which allows a client to watch the video while he is downloading it. The systems offering this kind of services have to deal with hundred thousands of potential clients. Centralized server topologies have shown their limits to support it as the server's bandwidth was quickly overwhelmed by clients. Services as YouTube propose to users to share and watch videos through a content distribution network (CDN) which consists of a set of servers exchanging information to deal efficiently with clients' requests, taking into account the client localization, servers load and content replication. But this kind of system requires expensive infrastructures and is not really scalable.

Therefore, research changed to another direction and looked toward peer-to-peer (p2p). Peer-to-peer systems do not require special infrastructure and work well on the best-effort Internet. Furthermore, the data are not only downloaded from a source server, but are also shared between clients' computers (called peers) which alleviate the server load and the bandwidth usage. This is in this context that the *State Key Networking & Switching Lab of the Beijing University of Posts and Telecommunications* has developed a p2p-based Video-on-Demand system, *PPlayer*. This software based on an open source project (1) has been enhanced to permit an efficient collaboration between different peers and the source server. The different protocols and entities involved in *PPlayer* are similar to existing systems as PPlive (2) or GridCast (3). In order for the peer-to-peer VoD system (P2P-VoD system) to maintain the classical server-client streaming performances, the video files have to be divided into many segments that are then sent from clients to other clients. The management of these up and down segments streams is called the segment scheduling.

This report is the result of an internship which takes place in the context of a project called "**Design and Evaluation of Streaming-Media Segmentation Scheduling Strategy in a P2P-VoD System"**. The main goal of this project is to propose an approach for segmentation of streaming-media files at different bit-rate and to design efficient scheduling strategies for playback, storage, transmission and advertising of segments between peers. My teacher Hongbo Wang orientated me towards a quite new approach of data sharing in networks: *Network Coding*. *Network Coding* basic principle is optimizing the use of bandwidth resources of a network by mixing streamed data according to a specific code scheme. This technique was first stated in 2000 in the seminal paper by R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network Information Flow" (4).

Under the highlight of articles dealing with *Network Coding*, my mission was to identify the different aspects and modifications to take into consideration in order to integrate

Network Coding in the existing system *PPlayer*. I worked with three of the people that have implemented *PPlayer*, to see in which way the system had to be modified in order to integrate *Network Coding* in an efficient fashion. Most of the modifications are related to the segmentation scheduling. This prototype allowed us to measure the performances of *Network Coding* in a micro environment. Indeed, under the internship time constraint, we worked without simulator and were unable to carry out measurements comparable with the results obtained in a real-world application as UUSee did. Nevertheless, this research work develops some elements that have just been presented in surface in articles on *Network Coding* applied to P2P-VoD system. The test we carried out shows that PPlayer can work with *Network Coding* with much less data redundancy that in the original system without altering the users viewing experience. Some ideas are also proposed to tackle *Network Coding main* weakness, the encoding and decoding complexity.

To well understand the topic of this research and the different achievements, the paper is organized as follow: in Section 0 we describe the original system *PPlayer*, in Section 0 the *Network Coding* concept is introduced and a basic algorithm and an optimized one are proposed to perform it, in Section 4 we explain how the original system *PPlayer* is modified in order to integrate *Network Coding*, in section 5 we discuss about the different aspects of the system that could be improved.

2 THE ORIGINAL SYSTEM

PPlayer is derived from an open source project initiated by Kevin (1). This open source implements the basic architecture of the P2P system. A group of students in BUPT have enhanced the system by adding a more evolved gossip protocol between peers and some functionality that were not in the original source code. *PPlayer* is a program that allows users to watch any movies proposed in a database. The interface displays the available movies that can be watched by the user. After starting a movie, the user can operate VCR operations which as a video cassette recorder includes forward, rewind, pause and stop operation. The forward and rewind operations are not continuous and can be described as a jump of the play position from a point in the movie to another one. Some research has been interested in integrated a continuous function in P2P-VoD systems but it is not our topic here. To understand how the video-on-demand service is provided let have a look to the system architecture.

As many P2P file sharing or streaming systems(2), (3) *PPlayer* has the following major components: (a) a source server which is the original media source, (b) a tracker to help the peers to connect each other in function of their ability to share data, (c) a web portal which allows the clients to check out the available movie on the source server, and (d) a log server which retrieves logged events from the different peers to perform data measurements. All this components and their interactions are illustrated in Figure 1.

When a peer connects to the system, it first registers on the tracker. Once the peer has started to watch a movie, it regularly reports its position to the tracker. Thus, when another peer will start the same movie, the tracker will send him a list of randomly picked up peers watching this movie. The peer will then exchange data with the peers of the list to check their aptitude to provide segments. Then, while downloading segments, the peer advertises the segments it already owns by gossiping a buffer map to the peers it knows (that we call neighbors) which will forward the message to the peers they know themselves.

When a peer has identified the peers to which to send a segment request, the scheduling algorithm works as follow:

- It checks if the number of peers likely to provide the segment has reached an arbitrary number. This number is 3 for non urgent segment and 5 for urgent segments and the request cannot be sent to more peers than this arbitrary number.
- Then it sends the request to the 3 or 5 peers. If this number cannot be reach, it sends an additional request to the source server.
- Once the segment is received from a peer, it sends a CANCEL SEGMENT message to the other peers.

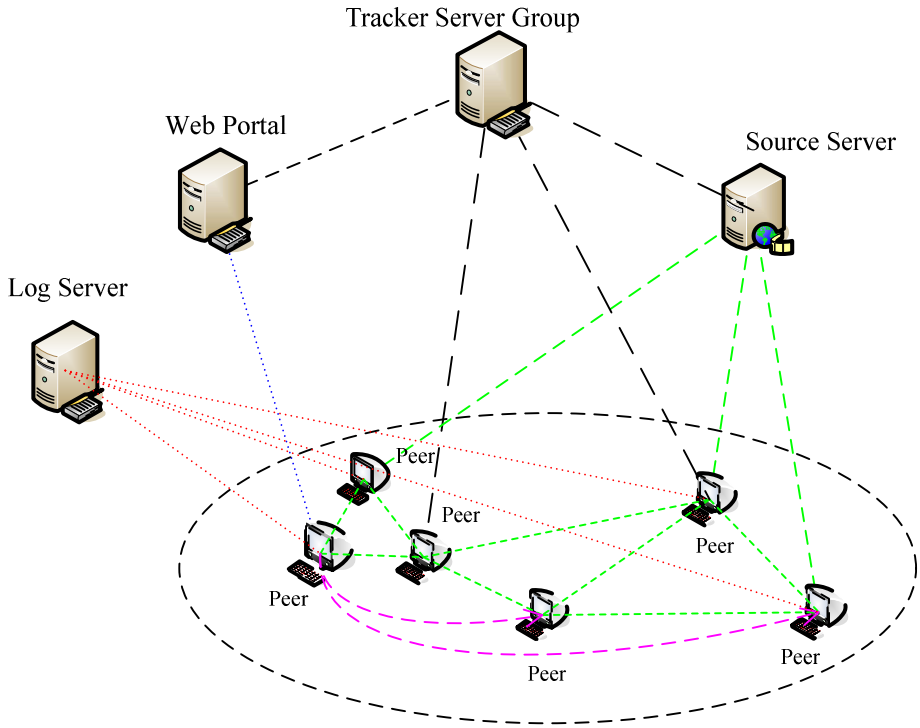

Figure 1 : *PPlayer* architecture. The dotted links shows the possible data exchanges between entities. The links between peers are of two types : the media data transfers (in green) and the gossip messages (in purple).

This algorithm has one evident weakness. To ensure the segment delivery with high probability, the request is sent to more than one peer. After downloading the requested segment, the CANCEL SEGMENT messages sent are not ensured to reach peers before they have already sent the segment. In the worst case, every peer has already sent a segment that is not useful any more, wasting network and peer resources. A solution is to send only one request to one peer and cancel it after a certain amount of time has passed before asking to another peer. This implies a meticulous management of time to synchronize peers which is a real challenge in a distributed streaming application. To avoid such kind of implementation, we looked forward *Network Coding* which is described in Section 0.

3 NETWORK CODING

PPlayer scheduler has shown incapacity to make peers get synchronized together. A new concept introduced in 2000 in (4) has revolutionized the way of considering network information flow. Information flow is not seen as an ordered sequence of messages any more but as a whole packet of information pushed into the network which shape can be decomposed and recomposed in order to reach the network capacity. This new breakthrough in network theory has found applications in multicast systems which aim to achieve the maximal flow capacity of a network between a source and one or several sinks. It has also found application in P2P systems with a random coding fashion where each peer is able from one segment and random coefficients to generate and transmit encoded blocks of information.

In 2005, a team from Microsoft Research proposed an implementation of a P2P large file content distribution system called Avalanche that uses *Network Coding* (5). In 2006, Lava (6) was the first experimental testbed for P2P live streaming content using *Network Coding*. In 2008, UUSee Inc. carried out the first large-scale deployment of random *Network Coding* adapted to a P2P-VoD system, collecting 200 Gigabytes traces throughout the 17-day Summer Olympic Games in August 2008. The analyses of these traces are presented in (7).

The randomization introduced by the coding process eases the scheduling of blocks propagation as every block may contains innovative information with high probability. Thus each coded block generated from a segment and random coefficients is valuable whatever from which peer it is received. This is particularly important in large unstructured overlay networks, where the peers need to make block scheduling decisions based on local information only. Thus, the peers can achieve a perfect synchronization independently of time constraints.

In a first time we present the *Network Coding* theory. Then, we observe the advantages and challenges raised by Network Coding. At last, we present our implementation of Network Coding through two algorithms, a basic one and an enhanced one that runs faster. The two algorithms are tested and results are discussed.

3.1 The Network Coding theory

3.1.1 The original concept

In (4), R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung showed that it was possible to maximize the flow of information from one source to several sinks in an network. This maximum is characterized by the minimum of the individual max-flows between the source and each sink. This minimum is called the maximal flow capacity of the network. Note that the maximal flow between the source and a sink can be described by the Max-Flow Min-Cut theorem explained in (8).

To achieve the maximal flow capacity of the network, even if it seems counter-intuitive, the information flow not has to be seen as a stream of sequential messages that can be routed and replicated at nodes. It has to be seen as raw information that we can mix (code) at different nodes and spread into the network to the sink which reorders the information

(decode) to get the original message. This concept is well illustrated in the Figure 2. The left part of Figure 2 shows a network with its capacity. The goal is to transmit 2 bits, b_1 and b_2, from the source s to the two sinks t_1 and t_2. The max flow capacity is 2 for both t_1 and t_2, so based on the *Network Coding* principle, the flow of information reaching the two think can be maximize to two. The graph in the middle shows how the bits can be propagated in the network without using *Network Coding*. We can observe that at node 3, b_2 will wait b_1 to be sent before to be sent itself. Through this pattern, the max flow capacity from the source s to the sink t_1 is not fully used. The graph on the right shows the same network but this time using *Network Coding*. The information is coded at node 3 resulting in one bit $b_1 \oplus b_2$ forwarded to node 4. Sink t_1 receiving b_1 and $b_1 \oplus b_2$ can recover b_2, sink t_2 receiving b_2 and $b_1 \oplus b_2$ can recover b_1. Thus, every sink receives information with flow equal to the maximal flow capacity of the network.

The question remaining is which coding scheme allows reaching the maximal flow capacity. In (4) the authors propose a coding scheme called *α-code* that can be generalized in many different coding schemes. Based on this *α-code*, (9) proposes a linear *Network Coding* and (10) proposes a random fashion of the same coding scheme called *Random Network Coding*. This is this coding scheme that uses most of the VoD-P2P systems based on *Network Coding* and that we develop in the next part.

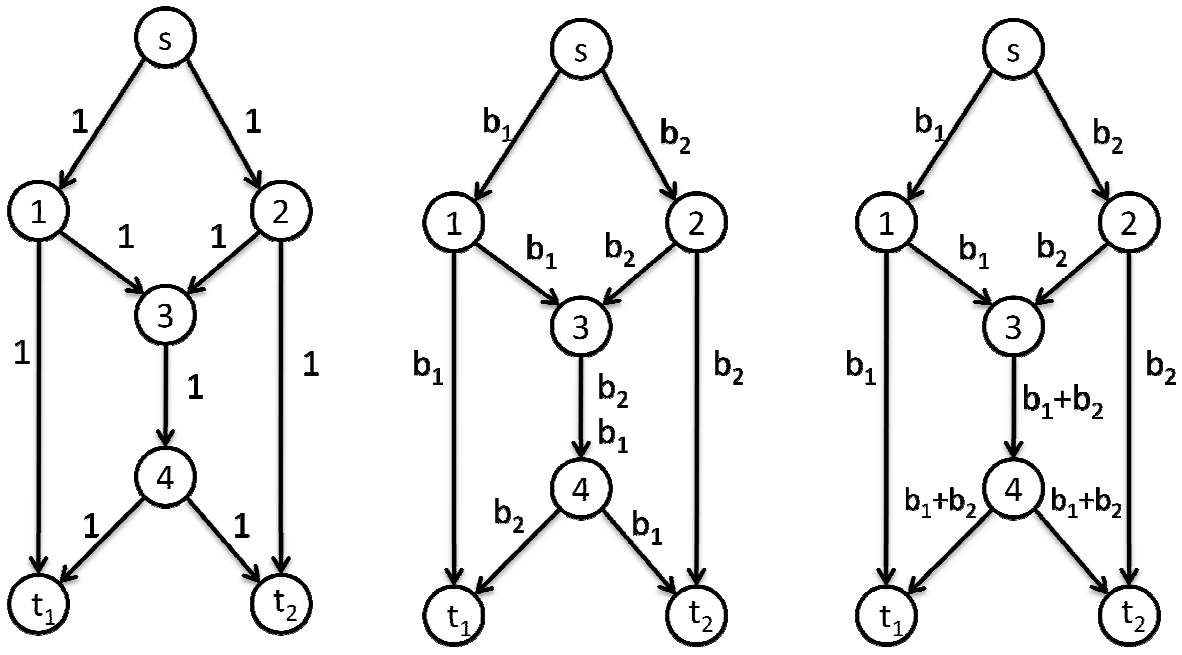

Figure 2 : a once-source two-sinks network with the network capacity (left) a uncoded transmission (middle) and a coded transmission (right) of two bits.

3.1.2 Random Linear Network Coding

In the *Linear Network Coding* designed in (9), the information at each node is encoded by computing a linear combination of the incoming data. The decoding process at sinks is based on the knowledge of the different linear transformations that have been applied to the original message.

In *Random Linear Network Coding* applied to a video stream, each segment S_i represents a message that need to be coded. For that, it is divided into n blocks $[b_1,b_2,\ldots, b_n]$ where b_i has a fix number of bytes $\boldsymbol{k}$ called the block size.

When a peer is set to provide a segment to one of its children, it first randomly chooses a set of n coding coefficients $C_i = [C_{i1}, C_{i2}, \ldots, C_{in}]$ in the Galois Field $GF(2^8)$. Then it multiplies $[C_1, C_2, \ldots, C_n]$ by $[b_1,b_2,\ldots, b_n]^T$ which produces one coded block x_i.

$$x_i = \sum_{k=1}^{n} C_{ik} \cdot b_k$$

The block and the coefficients used to produce it are sent to the downstream peer. The operation is repeated with a new set of coding coefficients for each new produced block. When the receiving peer receives $\boldsymbol{n}$ coded blocks produced by $\boldsymbol{n}$ linearly independent coefficient sets it can decoded the blocks by resolving the equation $\boldsymbol{C} \cdot \boldsymbol{b} = \boldsymbol{x}$ with $\boldsymbol{C}$ the matrix which rows are the coefficient sets of each coded block, $\boldsymbol{b} = [b_1,b_2,\ldots, b_n]^T$ and $\boldsymbol{x} = [x_1,x_2,\ldots,x_n]^T$. These $\boldsymbol{n}$ blocks are called *innovative* as their coefficient sets being independent they bring new information about the segment. To resolve this equation, the receiving peer has to compute $\boldsymbol{C}^{-1}$ to get $\boldsymbol{b} = \boldsymbol{C}^{-1} \cdot \boldsymbol{x}$. It is obvious that without n innovative blocks the matrix cannot be inversed, $\boldsymbol{C}$ not being full rank. This equation is illustrated in Figure 3.

Note that the inversion in a finite field as Galois Field is possible thanks to its particular arithmetic which provides it a ring structure. The cardinal of $GF(2^8)$ is 256 which is equal to the number of values that can hold one byte. Therefore, the different operations are applied on bytes. Each coefficient is coded on one byte and the multiplication of a block by a coefficient is processed by multiplying each byte of the block by the coefficient. For more information on Galois Field the reader is referred to (11) and (12).

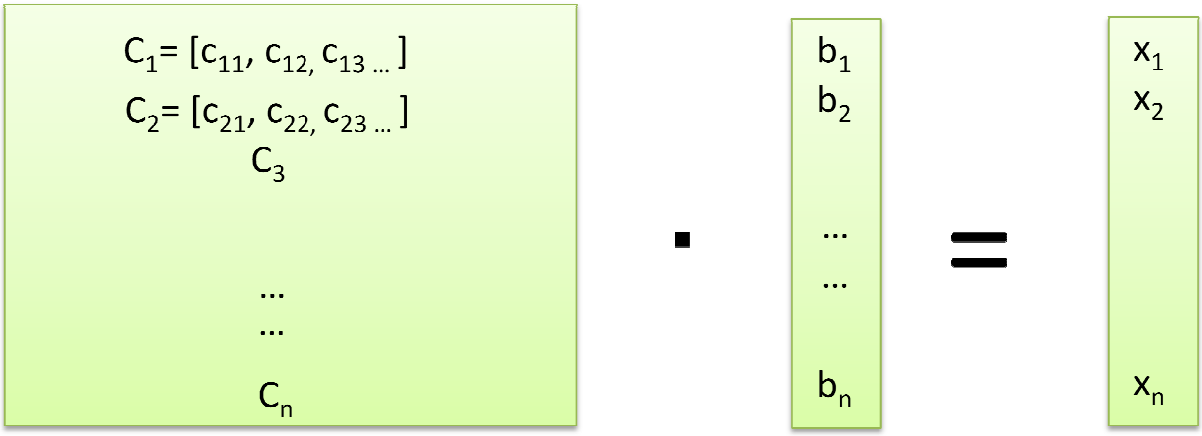

Figure 3 : Equation to resolve to decode the segment.

3.2 The advantages and challenges raised by Network Coding

Which consequences should have *Network Coding* on a system like *PPlayer*? In this part the advantages and challenges identified in (7) and (13) are described. They will be

observed through tests presented in this section and in Section 4. Some solutions are proposed to deal with the identified challenges in Section 5.

3.2.1 The Advantages

Rateless code and resource use optimization. The code rate of a coding scheme is the fraction between the number of symbols in an original message and the number of symbols needed to encode the message. In *Network Coding*, as ***n*** blocks are sufficient to recover the original segment with high probability, this coding scheme is called rateless. This propriety implies that the peers' upload and download bandwidth should be saved. Indeed an arbitrary number of different peers can be used to serve the same segment to a receiving peer. The receiving peer needs to receive only ***n*** innovative blocks to decode the whole segment. When these ***n*** blocks are received, the peer sends a message to its parents to ask them to stop sending blocks. Furthermore, a peer can send new linear combinations of blocks before receiving the whole segment. The coefficients being randomly chosen a received block is innovative with high probability. It results that different peers can collaborate to provide one segment to a common child. Thus, we get a perfect data synchronization much safer and easier to implement than time synchronization. A direct consequence of this first advantage is that even a peer with a low upload bandwidth can participate in providing the segment without harming the whole system throughput. Without using *Network Coding*, a slow peer could be asked to provide a segment but this segment would be worthlessly sent as the demanding peer would have already got it from a faster peer.

Near-optimal code. *Network Coding* can transform a message in practically an infinite encoded form. It means that a peer can generate blocks ad infinitum until ***n*** innovative blocks are received by its child. This propriety implies the use of UDP flows as we are not concerned by the lost of one block on the network. UDP is stateless and can be much faster than TCP on an unreliable channel. Indeed, for every dropped packet TCP will use a congestion control called AIMD (additive increase multiplicative decrease) which is an algorithm designed 30 years ago and prevents TCP to realize the optimal throughput (14), (15). Besides, the process to recover dropped packets implies delays. In the implementation of UUSee, it has been decided to use UDP packets not bigger than 1KB to permit even the slowest peers to provide one block. From our point of view it can be useful to avoid a lot of fragmentation due to MTU in particular on a network as Internet. Indeed, if one fragment is lost, the whole UDP packet becomes useless.

3.2.2 The Challenges

Encoding and decoding CPU overhead. The encoding and decoding process implied by random linear coding consume a lot of CPU resources. The complexity of these operations depends on the size of the matrix $\boldsymbol{C}$ to inverse and the size of the blocks. Every article dealing with *Network Coding* in P2P share content systems talks about the same algorithm to inverse the matrix but the implementation of this algorithm and the condition under which it is compiled can differ. In part 3.3 we propose our own implementation and compare the coding and decoding speed obtained with the results obtained in the articles on the same subject.

Block overhead. Each block sent contains a header composed of the coefficients and other metadata. Each coefficient is coded on 1 Byte. The number of coefficients is equal to the number of blocks in one segment. The larger is the number of blocks in one segment the

larger is the number of coefficients and block overhead. To avoid this overhead the seed used to randomly generate the coefficients can be sent rather than each coefficient. However, if a peer encodes a block from encoded blocks it will have to send two seeds, the one used to encode the original segment and the one used to re-encode from the coded received block.

Network Overhead. When a peer has received enough blocks to decode the segment, it informs its parents that it does not need blocks anymore by sending a CANCEL SEGMENT message. A delay is needed for the message to reach the parents and to be treated. During this delay, additional redundant blocks are sent on the network (this phenomenon is called breaking redundancy in (7)). As the number of blocks in a segment becomes smaller, the overhead of such redundancy becomes more significant. It comes from the fact that for a smaller number of sent blocks, the delay is not changed (only depending on the network conditions and the end-user computers' reactivity) which means that the number of redundant block is the same. So we may prefer a larger number of blocks. But more blocks means more set of coefficients and more complexity to inverse the matrix $\boldsymbol{C}$. This tradeoff has to be taken into account.

3.3 Network coding implementation

The algorithm used to resolve $\boldsymbol{C} \cdot \boldsymbol{b} = \boldsymbol{x}$ is the Jordan-Gauss Elimination (16). This algorithm allows performing the resolving process before getting all the segments. Thus, the matrix can be inversed while the UDP packets are received.

3.3.1 The original implementation

3.3.1.1 The Jordan-Gauss Elimination algorithm

When a coefficient row is received, we operate linear combinations between the new coefficient row and the rows already received in order to obtain **the reduced row-echelon form (RREF)**, in which each row contains only zeros until the first non-zero element which has to be 1, the elements above and below this 1 are 0. The transformation $\mathcal{T}$ operated to get the RREF matrix are also applied to the block vector x. When $\boldsymbol{n}$ independent coefficient rows have been received the RREF matrix has the identity matrix form and the blocks are decoded (Figure 4).

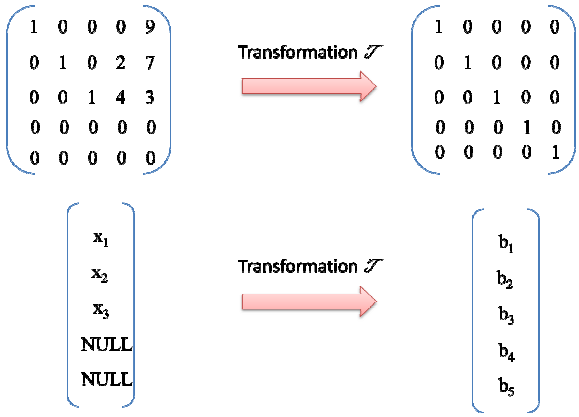

Figure 4 : On the left the RREF matrix and the partially decoded blocks after receiving 3 coded blocks. On the right, the identity matrix and the decoded blocks after the transformation $\mathcal{T}$

The algorithm to get the RREF matrix is illustrated in the Figure 5 to 8. It is divided in 3 steps:

- **Step 1:** reduce the leading coefficients to 0.
- **Step 2:** divide the row by the leading coefficient.
- **Step 3:** reduce the coefficient matrix to the RREF matrix.

In our example, at start, we have a matrix filled with two coefficient rows already reduced, a new coefficient row waiting to be inserted in the matrix, a new block, and the already received blocks (Figure 5). In step 1, the different linear combinations with the existing rows permit to insert the new coefficient row in the right empty row in the matrix (Figure 6). In step 2, the row is divided by its first non-zero value in order to get 1 in the diagonal (Figure 7). In step 3, the rows above the new one are reduced to insure that all the elements above the 1 in the diagonal are equal to 0 (Figure 8).

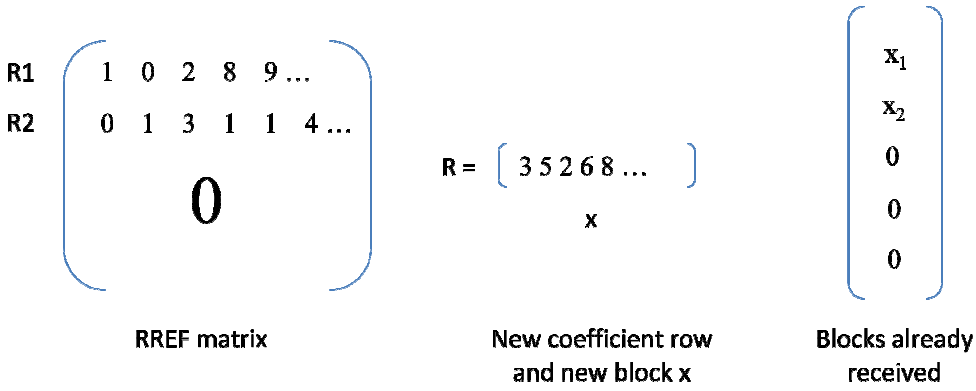

Figure 5 : Initial state.

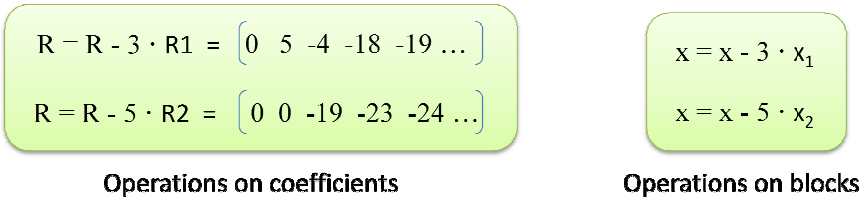

Figure 6 : Step 1.

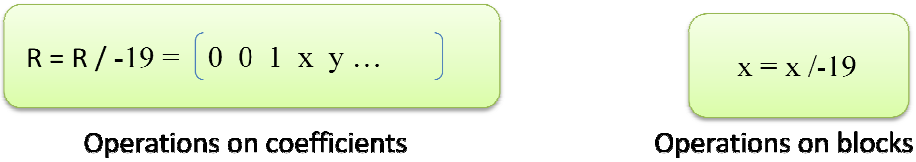

Figure 7 : Step 2.

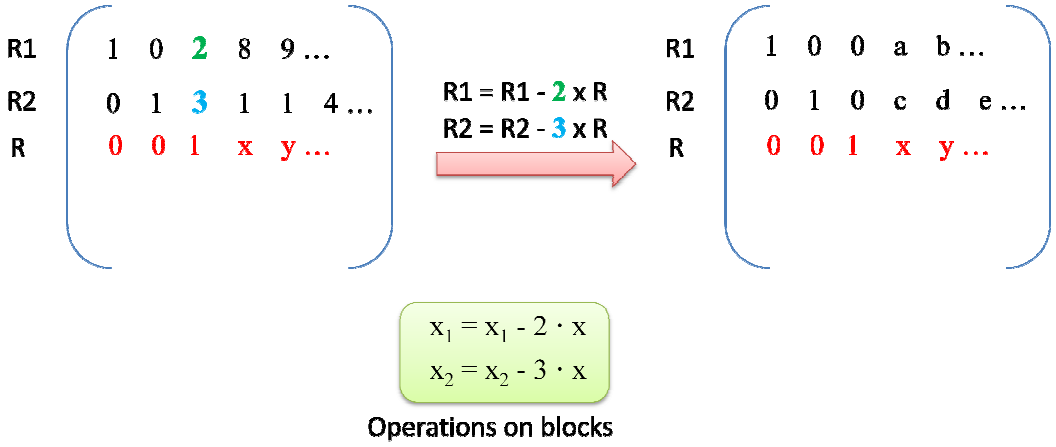

Figure 8 : Step 3.

One particular case is when after step 1 the new coefficient row does not fit the empty row just below the already inserted coefficient rows because its value at the diagonal is equal to 0. In this case, the new row is inserted at the right line below, step 2 is performed and step 3 will be performed only when all the empty rows above the newly inserted row will be filled. Thus we do not uselessly perform step 3 which save time. This case is illustrated in Figure 9.

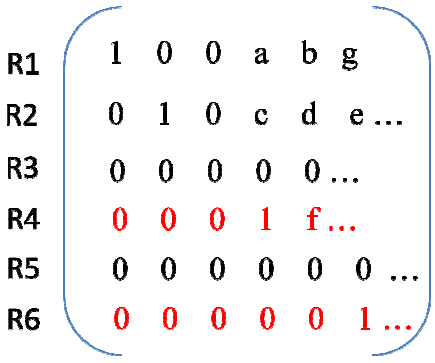

Figure 9 : the step 3 will be applied to R4 when R3 will be filled and will be applied to R6 when R3 and R5 will be filled.

3.3.1.2 Algorithm complexity

The operations in GF(2^8) are multiplications and additions intensively used in different loops. The addition is simply a XOR operation. The multiplication is much more complicated and need to be optimized. The multiplication can be operated faster with 256-entries exponential and logarithmic tables in GF(2^8). As an example, the multiplication between $\boldsymbol{a}$ and $\boldsymbol{b}$ is equivalent to computing exp(log($\boldsymbol{a}$) + log($\boldsymbol{b}$)). This implementation needs one addition and three memory-reads for each multiplication (6).

The complexity of our algorithm is measured by the number of multiplications and divisions performed to obtain the identity matrix. The case pointed above stating that gaps can possibly appear in the coefficient rows is ignored because rarely occurs. For each linear combination we have $\boldsymbol{n}$ multiplication. A new coefficient row insertion at row number l in the matrix will run step 1 in $\boldsymbol{l} \times \boldsymbol{n}$ multiplications. Step 2 is run in $\boldsymbol{n}$ divisions. Step 3 is run in $\boldsymbol{l} \times \boldsymbol{n}$ multiplications. To get the identity matrix $\boldsymbol{n} \times \boldsymbol{n}(\boldsymbol{n}+\boldsymbol{1}) + \boldsymbol{n}^2 = \boldsymbol{n}^3 + 2\boldsymbol{n}^2$

multiplications are performed. For a $\boldsymbol{k}$ bytes long block $\boldsymbol{n}$ is replaced by $\boldsymbol{k}$. To fully decode the segment $\boldsymbol{k} \times \boldsymbol{n}(\boldsymbol{n}+1) + \boldsymbol{k}^2 = \boldsymbol{k}\boldsymbol{n}^2 + \boldsymbol{k}\boldsymbol{n} + \boldsymbol{k}^2$ multiplications are performed. In Section 5, we will see how the complexity can be decreased. The encoding complexity is $\boldsymbol{k}\boldsymbol{n}^2$. The encoding process should be slightly faster than the decoding process.

3.3.1.3 Measurements

We have implemented the algorithm in C++ using two open libraries on internet. One provides an efficient random generator. It mixed two random generator algorithms *Mother-of-All* and *Mersenne Twister* to get the more uniform random function. Furthermore, the generation process is accelerated by using the SSE2 set of instruction which permit to applied operation on 16 bytes data in one instruction (17). The other library performs operation in GF(2^8) using exponential and logarithmic tables (18).

We have measured the encoding and decoding speed of our algorithm with segments of different size, each composed of 1KB blocks. The compiler is Microsoft Visual C++ Toolkit 2003 and the option *maximize speed* (-O2) is activated. The compiler automatic SSE2 optimizations are allowed. The tests are realized on an Intel Core 2.10 GHz and we get mean results out of 100 segments for each size. The results are shown in Figure 10. First, we can observe that the encoding speed is slightly faster than the decoding speed as predicted by our assessment of the algorithm complexity. The decoding speed should decrease faster than the encoding speed as the number of blocks increases but it is not obvious on our graph. Secondly, the speed decreases quickly with the number of blocks increasing linearly. It is implies by the complexity in $\boldsymbol{n}^3$.

Encoding and decoding measures without particular acceleration have been carried out in (19). Using a 1.83 GHz Intel Core Duo processor the speed achieved for 256 blocks of 1KB is respectively 36KB/s and 26.9KB/s for encoding and decoding speed. We measured respectively 173.4KB/s and 148.3KB/s. This gap between the values is first due to the processor characteristics (speed, caches, …). The speed difference also depends on the memory management and the number of loops. Indeed, in the current implementation we have limited the number of dynamic variable declarations, and eliminated the most intermediate variables. The reader can refer to the appendix 7.1 to get the C++ source implementation.

We also observe that by multiplying the size of the blocks and dividing the number of blocks by the same factor we can encode the same amount of data much faster. This is due to the fact that the operations operated on blocks complexity increases in k^2 and the coefficient row operations complexity decrease in n^3. For 500 blocks of 1KB the encoding and decoding speeds are respectively 82.9KB/s and 66.5KB/s whereas for 250 blocks of 2KB they are respectively 162.3KB/s and 158.6KB/s.

The highest movie rate that *PPlayer* source server stores is 171KB/s. *Network Coding* cannot sustain such a rate for segment larger than 200KB. Moreover, during these measures a minimum of applications were open on the computer. If *Network Coding* were integrated in *PPlayer*, it would share the CPU resources with other threads. The measures show us that we can only deal with segments smaller than about 150KB. It is obvious that we cannot afford the rate as such small segments would imply substantial overhead as explained in Section 3.2.2. To integrate *Network Coding* to *PPlayer* we need a faster implementation. This can be

provided by a SIMD (Simple Instruction Multiple Data) implementation that allows operating on range of 16 bytes in one instruction time. This implementation is described in the next part.

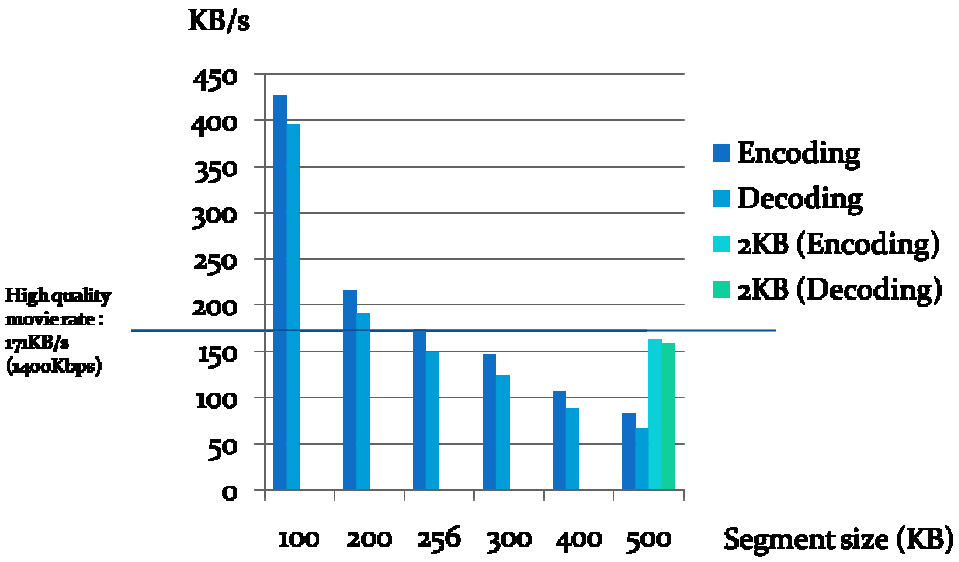

Figure 10 : the basic implementation encoding and decoding rate measured on a Intel Core 2.10 GHz. Blocks are 1KB long except for one test where blocks are 2KB long.

3.3.2 The optimized implementation

3.3.2.1 Implementation

The optimized implementation employs hardware acceleration using a set of instructions called SIMD implemented on all modern processors. These SIMD instruction set allows a single operation to be performed on multiple data in a parallel fashion. Intel x86's implementation of SIMD is called SSE (Streaming SMID Extension) which has matured in SSE2 introduced in the Pentium 4 family in 2001 and on AMD processor since 2003. To our knowledge, the first attempt to implement *Network Coding* using SIMD is related in (19). Yet, the way to use SIMD instructions set is not described and we had to refer to (20) and (21) to achieve the implementation.

```
byte gf256::loop_gf_multiply(byte x, byte y)
{
    byte result = 0;
    bool overflowing;
    while (x != 0) {
        if ((x & 1) != 0)
            result = result ^ y;
        overflowing = y & 0x80;
        y = y << 1;
          // irreducible poly: x^8+x^4+x^3+x^2+1
        if (overflowing == true)
            y = y ^ 0x1d;
        x = x >> 1;
    }
    return result;
}
```

Figure 11 : basic implementation of the multiplication in GF(2^8) between two bytes

To take advantage of SIMD instructions we have to give up the exponential and logarithmic tables and go back to a basic implementation of the multiplication in GF(2^8) described in (19) (Figure 11). The basic implementation is composed of a loop which has to be modified to work with 16 bytes long data. To implement it we use functions in the library

emmintrin.h of msdn which embeds inline functions written in assembly language and operating on data of type __m128i which represents 16 bytes array in the computer.

As this paper is not a programming manual we only detail how we implemented the instruction: if ((x&1) != 0) result = result ^ y. The whole implementation is available in the appendix 7.2. The trick to apply condition on __m128i data is to use a mask in order to operate only on particular bytes of the array. The functions we use are listed below (a = [a1 a2 a3 a4… a16] and b = [b1 b2 b3 b4… b16] are two 16 bytes arrays):

- **_mm_and_128(a, b)**: return the array r = [a1&b1, a2&b2, … a16&b16].
- **_mm_cmpeq_epi8(a, b)**: return the array r = [a_1==b_1 ?**1**:**0**, a_2==b_2 ?**1**:**0**, … a_{16}==b_{16} ?**1**:**0**] with **1** the byte with all bits set to 1 and **0** the byte with all bits set to 0.
- **_mm_xor_si128(a,b)**: return the array r=[a1 ^ b1, a2 ^ b2, … a16 ^ b16] (where ^ is the XOR operation).

The vectors used to compute the mask are:

- _m128i x = 1001 0001 1010 0010 0100 1000 1010 0011 … x16
- _m128i un = 0000 0001 0000 0001 0000 0001 0000 0001 … x16

We have **mask = _mm_cmpeq_epi8(_mm_and_si128(un, x), un);**
Once we have obtained a mask, an AND operation with the mask permits to select the bytes we want to operate on. Let take as an example the instruction *result = result ^ y*:

If $(x_i\&1)!=0$ we will compute $result_i = result_i \wedge y_i$. if we have $(x_i\&1) == 0$ $result_i$ should be unchanged. For that, we first compute **maskedY = _mm_and_si128(mask, y);** then we compute **result = _mm_xor_si128(result, maskedY);**

These operations are clearly summarized in Table 1.

Instructions	16 bytes array variables				
_m128i y	1011 0101	0010 0110	1100 1000	1010 0011	… x16
_m128i result	0010 0001	0000 1010	0001 0011	0000 1100	… x16
mask	1111 1111	0000 0000	0000 0000	1111 1111	… x16
maskedY = y & mask	1011 0101	0000 0000	0000 0000	1010 0011	… x16
result = result ^ maskedY	10010100	0000 1010	0001 0011	1010 1111	… x16

Table 1 : instruction result = result ^ y implemented through a mask.

3.3.2.2 Measurements

The measurements have been performed in the same condition that the non-optimized implementation. As we operate on 16 bytes array the algorithm only works on segment which size is a multiple of 16. We get a considerable speed improvement as shown Figure 12. For each segment size, the speed is improved by a factor 10 (1000%). The step 2 (division by pivot) has not been optimized because we don't have any practical implementation of the division in GF(2^8). Further research may found out the right algorithm allowing implementing SIMD optimization for the division.

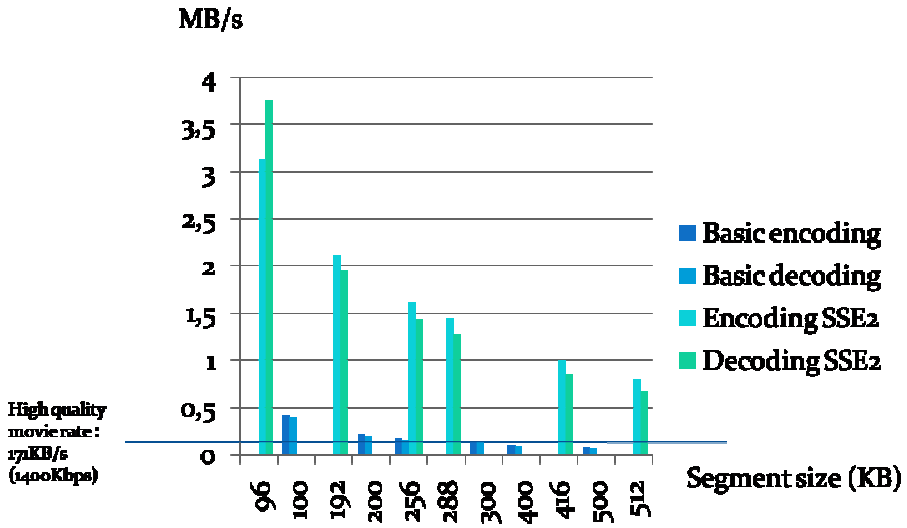

Figure 12 : the SIMD optimized implementation encoding and decoding rate measured on a Intel Core 2.10 GHz compared to the basic implementation speed. Blocks are 1KB long.

Figure 13 shows the results obtained in (6) and (19) compared to our own results. In (19) the SMID implementation is enhance by a multithreaded encoding and decoding process. In *PPlayer*, peers encode and decode segments at the same time. It involves already two different threads. Most of the clients' computers integrating at most 2 CPU cores, using multithreaded encoding and decoding processes would only add multithread overhead which instead of improving speed would make lower the encoding and decoding processes.

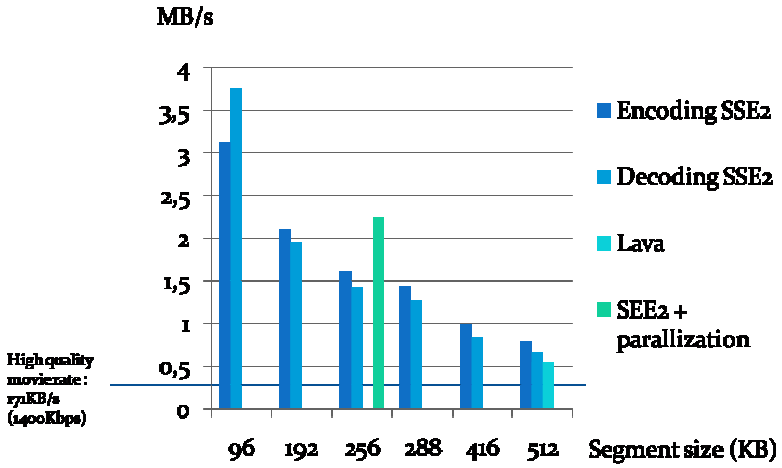

Figure 13: the SIMD optimized implementation encoding and decoding rate measured on a Intel Core 2.10 GHz compared to the result achieved by Lava (SEE2 on Dual core Pentium 4 Xeon 3.6 GHz) and the algorithm implemented in **(19)** (SEE2 + parallization on Dual P4 Xeon 3,6Ghz). Blocks are 1KB long.

4 INTEGRATION OF NETWORK CODING IN *PPLAYER*

We have in the previous part validated our *Network Coding* implementation. The different measurement results that we have obtained are indicators that will help us to design a robust P2P-VoD system based on *Network Coding*. To achieve this integration we planned the research as follows: first identify the dilemmas we have to deal with. Then, propose a first implementation and perform some measurements. Under the light of these measurements propose a second implementation.

4.1 Design dilemmas

How many children can provide a peer? Since the CPU load induced by encoding process is heavy and time consuming how a peer can deal with different children. Indeed, *PPlayer* allowed a peer to have a maximum of five children. If we consider the *Network Coding* encoding speed it appears obvious that a peer cannot deal with more than two or three children by its own. A child supplied by slow parents will need more parents to get the segment on time. The parents' efficiency is highly empirical and mainly depends on the environment the peers evolve in. Yet *PPlayer* provides safeguard mechanisms to assure that a peer will get a segment on time, in the last case requesting it from the server. So it could be interesting to tune the maximum number of parents and observe the effects on the system throughput by recording the server load.

The experiment has not been carried out as the laboratory environment is very small. Yet, we suppose that the more children a peer can provide the better will be the system throughput. The bottleneck in providing a peer is the encoding speed. If different children ask for different segments at the same time the same number of encoding process will be triggered on the same peer. In part 5.2 we propose a solution that could improves the number of children supplied by one peer but it is not an absolute necessity to make the system work. Because of time constraint and coding complexity it has not been implemented. However, we will explain which considerations have lead to this solution.

From how many peers a segment should be requested? In UUSee (7) peers download one segment from 10 to 30 parents. In PPLive (2) peers download segment from 8 to 20 neighbors. These values are highly empirical as explained by the authors of the two articles. We first do not change the maximal number of parent from which a peer can request segment in *PPlayer* (3 parents). One research direction could be to design a system which can determine the optimal number of neighbors after probing an environment. Unfortunately, because of time constraints we did not have the time to explore this problem.

How to deal with block dependency? This question has been partially answered in the first design dilemma. However, let consider a more basic implementation where only decoded segments are stored on the peer memory and encoding process is triggered each time a segment is requested. Should the block linear independence be check for each block sent? The answer to this question is negative as the process to check it is more time consuming than sending not more than 1 or 2 blocks more than the n initial block. The question becomes relevant when it is related to the server. Indeed, the server has to deal with many clients and cannot afford with encoding complexity. If we want the server to collaborate with the peers to

provide block we have to provide him with blocks already ready to be sent. If the server provides decoded blocks (decoded blocks are as useful as encoding blocks and does not imply decoding process) that it sends using TCP connection, the peers only connected to the server are ensured to get the right blocks. Then the server does not need to send more than *n* blocks which would load the server with computing process. But the size of each block being small, the CPU overhead use to send a TCP packet could be very heavy for the server. Measures need to be performs. This problem does not seem to have been pointed out in the articles related to *Network Coding* in P2P-VoD systems (7),(13),(6). Unfortunately, we did not implement *Network Coding* on the server because of time constraints. However, we performed tests showing that using unencoded blocks with encoded blocks does not increase the number of non-innovative blocks.

Segment size. The more suitable size is not easy to determinate. This is a trade-off between decoding speed and overhead implied by the delay to receive the CANCEL message after a peer has received enough blocks. UUSee proposes to divide segments in 300 to 500 segments in function of the movie rate. This number is empirical and aims to reduce the breaking redundancy. We choose 320 blocks for our first tests. With test carried out in the same situation as previously we get a decoding speed of 1.05MB/s.

How many segments can be requested at the same time ? In *PPlayer*, a request buffer store all the segment requests that have to be sent. This buffer is divided into two areas, the *urgent area* and the *normal area*. If the requested segment position in the movie is very close to the media player position in the movie, the request will be considered as urgent and put in the *urgent area*. If the distance between the media player position and the requested segment is large enough, the request will be considered as normal. Every 2 seconds, all the segment requests in the urgent area and some of them in the normal area are sent. If *Network Coding* is implemented in this system, it means that blocks from different segments can be received at the same time triggering different decoding process.

The question is how many decoding processes can be sustained by a decoder. Figure 14 shows the approximate encoding and decoding speed of an Intel duo core 2 Ghz using 25% of the processor (embedded two cores). It is approximate result we got by dividing by 4 the results obtained in 3.3.2.2. We consider that the overhead produced by the CPU to deal with different threads has minor incident on the encoding and decoding speed. This assumption need to be verified by implementing *Network Coding* in *PPlayer*. Under these conditions, we can decode 320 blocks of 1KB (3.74 s of a 700kbps movie and 1.87 s of a 1400kbps movie) in 1240 ms. A computer downloading the data at a speed of 100kB/s will get one segment in 3.2 s. The time overhead needed to decode the block is difficult to assess as the decoding process and the receiving process are simultaneous. But for sure, the time will be inferior to 3.2 + 1.24 = 4.44 s which is already much more than 3.74. If we do not want to push the computer to its limits we can hardly deal with more than one or two segments at the same time.

The different dilemmas we have highlighted will help us to analyze the results we will get in our implementation of *Network Coding*. Indeed, the assumptions we make are based on parameters that we do not control (CPU speed, multithread management) and the real behavior of the system is hardly predictable.

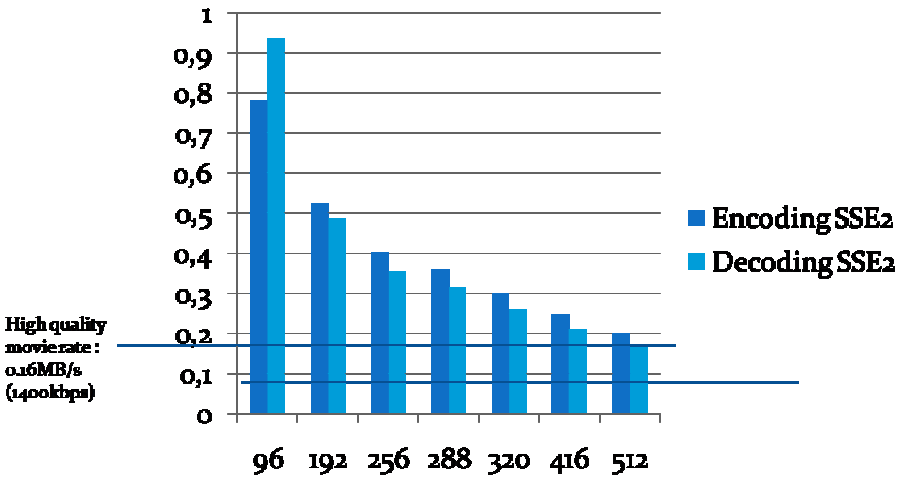

Figure 14 : encoding and decoding speed using 25% of the CPU.

4.2 First implementation

4.2.1 Design

PPlayer is written under the object paradigm which provides a modular implementation. Therefore, *Network Coding* can be integrated as a new module with an adapted interface to communicate with the already implemented modules. The two modules with which *Network Coding* module interacts are two objects: CP2PRequestManager and CP2PNetwork. CP2PRequestManager is responsible for the segment requests that it sends to CP2PNetwork. It can be consider as the segments scheduler. CP2PNetwork can be considered as a low layer module in charge of sending the requests on the network and receiving the segments sent by the parent peers. The received segments are then sent to CP2PRequestManager which will delete the requests and provide the segments to the media player (Figure 15 (a)).

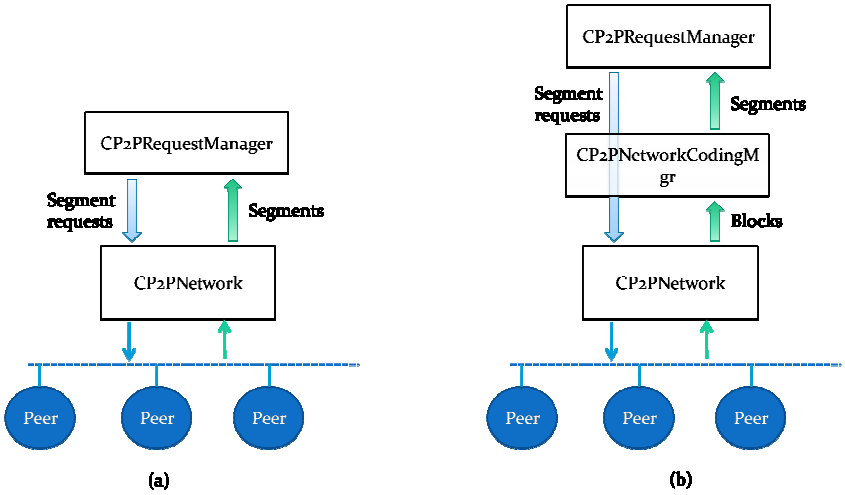

Figure 15 : *PPlayer* scheduler modules (a) and integration of *Network Coding* module in *PPlayer* (b).

The *Network Coding* module is an object called CP2PNetworkCodingMgr. It is integrated between CP2PRequestManager and CP2PNetwork (Figure 15 (b)). Blocks are UDP packets composed of (a) the encoded data, (b) the channel ID (in *PPlayer* each movie is considered as a channel to which the peers connect), (c) the segment ID and (d) the random generator seed. Each block is 960B long, and the UDP packet is 1KB long (with the packet header included). One segment is composed of 320 blocks which results in 300KB long segments.

In our first implementation, a thread running in CP2PNetwork is in charge of sending new blocks to CP2PNetworkCodingMgr. CP2PNetworkCodingMgr checks if it belongs to the right channel. If the peer has connected to a new channel and blocks from the former channel are still received they are discarded. If the blocks belong to the right channel they are put in a list. CP2PNetworkCodingMgr hold a thread pool. This pool contains threads that are already initialized and will be triggered faster than not initialized threads. A main thread regularly checks if there are new blocks in the list and called a thread in the pool to deal with it. All the blocks sharing the same segment ID are treated in an object called decode unit characterized by its segment ID. The thread called from the pool will determine which decode unit to use to treat the block in function of its segment ID. After treating the block the thread check if the decode unit has finished decoding the whole segment. In that case, the segment is sent to CP2PRequestManager. Figure 16 summarized these different steps and the objects involved in the decoding process.

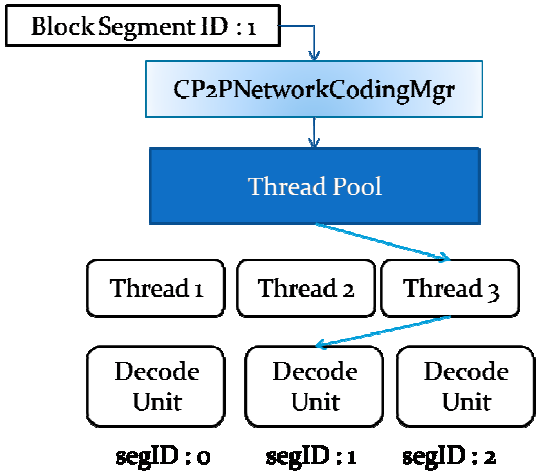

Figure 16 : decoding process in CP2PNetworkCodingMgr.

4.2.2 Test results

To test the performance of our implementation we first run two peers. Peer A first connects to the system. Being the only one it requests segments from the source server. After a few time, peer B connects to the system and ask A to send blocks to him. The objects of our interest are CP2PNetworkCodingMgr and CP2PNetwork. These two objects write log information in a file each time they perform an action. We assess the *Network Coding* implementation through the study of this log file.

The performances of the first implementation are not very conclusive. Our most important concern is the receiving block speed. Indeed, peer A encoding units produces 294 blocks/s. The thread in charge of sending blocks through UDP packets treats 70 blocks/s. P2PNetwork on peer B receives the blocks at a speed of only 10 blocks/s (Figure 17). A deeper analyze of our implementation reveals some weaknesses that could be at the origin of such a gap between the sending speed and the receiving speed. We do the test once more time but this time each block is assigned a block ID. Thus, we can detect if the blocks are lost between peer A and peer B or if they are just treated slowly by peer B. The log on peer B shows that some blocks are missing on B.

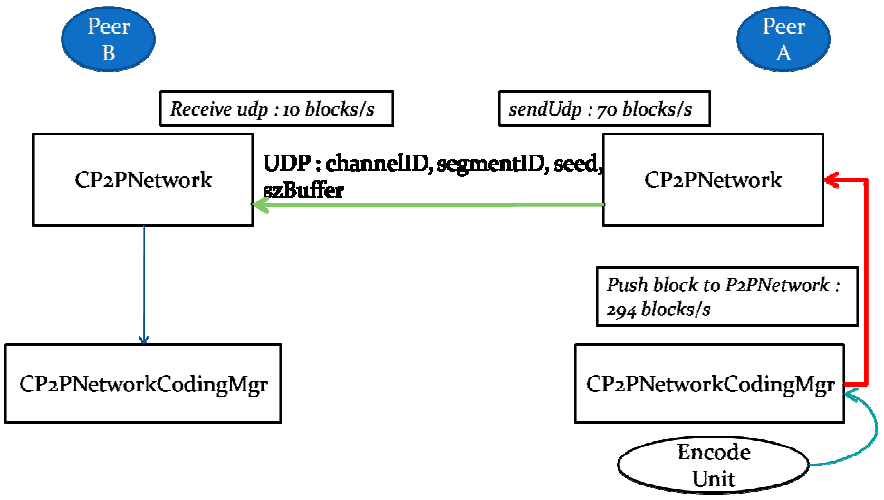

Figure 17 : block flow at different key points of a block transfer between peer A and peer B.

By using the sniffer Wireshark we probe the network to see where the sent packets have disappeared between peer A and peer B. There are three possibilities: 1) The packets have been pushed to the socket by CP2PNetwork but have not really be sent on the network 2) Part of the packets have been dropped by the switch installed between peer A and peer B 3) The packets have reached peer B but has not been treated by B. Wireshark reveals that all the packets sent by peer A have reached peer B network interface card. Hence, this is the third possibility which is validated. It seems that some of the blocks arriving at the socket on peer B are dropped. To each socket is assigned a buffer which will discard UDP packets if this buffer is full *i.e.* if the sending speed and the speed at which the packets are treated are different. The socket buffer size is automatically set to 8KB when it is created. We change this value to 1MB. It result that most of the packet sent by A are treated in peer B. But, as expected, it has no influence on the receiving speed.

We think that these low speed results may be implied by a heavy multithread management. Furthermore, some threads need to run faster than other and hence should have an easier access to the CPU resources. We decide to design a more simple system which will include a CPU resource management adapted to each process needs.

4.3 Second implementation

4.3.1 Design

In the first implementation, only one thread deals with the TCP and the UDP packets. In this configuration peer A cannot receive a segment from the server (TCP connection) and at the same time send block to peer B. Indeed, it will have to wait for receiving the whole segment (300KB) before to send one block (960B). Therefore, one new thread is created to receive blocks. The thread which generates encoded block sent it at the same time with a synchronous call to the UDP socket in CP2PNetwork. Thus, the number of generated blocks and the number of sent block are the same.

The thread pool was considered as a good idea as different threads are already ready to deal with new blocks. But it appears that the management of such a pool is very heavy for an application that needs to be fast on standard computer. Such a thread pool may have better results on powerful server dealing with several clients. Furthermore, CP2PRequestManager asks for at most two segments every two seconds, the use of multithread to decode segment is not really justified. Hence, we decide to abandon the thread pool and to substitute it by a single thread which each round check for a new block in the blocks list and treat it in the right decode unit.

To permit thread to use more resources than others, we force some of them to sleep after each round. The encode unit sleep 1ms after sending one block. The thread which receives the blocks through the UDP socket does not sleep as long as blocks are in the socket buffer. The thread treating each received block in a decode unit sleep 4ms after decoding one block. In Figure 18, these three threads are called *EncodeUnit*, *UdpReceiverMgr* and *MainProcThread*. Each thread can be assigned a priority value ranging from 0 to 31. The CPU time slices assigned to the thread will be more important if the priority is higher. The priority default value is 0. We set *MainProcThread* priority to 15 which allows it to operate more operations than the other threads when it is running.

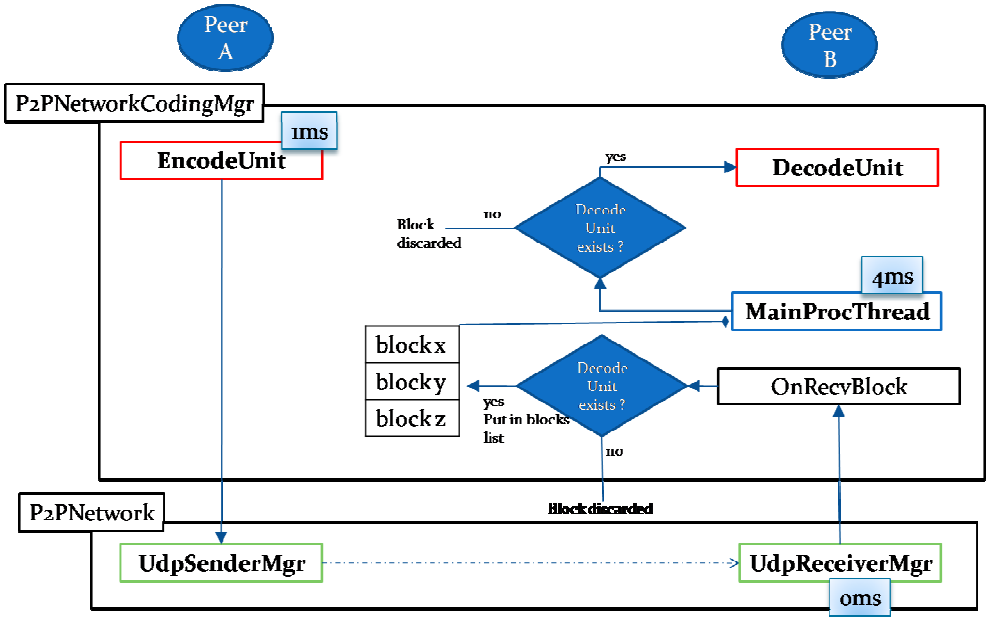

Figure 18 : new implementation design. The filled arrows indicate synchronous calls. The times in ms indicate the sleeping time after a thread has performed one round.

When CP2PRequestManager requests a segment, it asks CP2PNetworkCodingMgr to create a new decode unit. When CP2PRequestManager has received the segment, it asks CP2PNetworkCodingMgr to destroy it. The blocks corresponding to the same segment ID are discarded from the block list. At each step the blocks that correspond to a decode unit that has been destroyed are immediately discarded (diamond-shaped condition in Figure 18).

4.3.2 Test results

The results obtained with the second implementation are more conclusive. However the block flow from peer A to peer B is not uniform and we will try to comment these variations.

The thread *EncodeUnit* generates and sends blocks at a speed of 384 blocks/s. *UdpReceiverMgr* receives blocks at a speed of 340 blocks/s and puts it in the block list at a speed of 336 blocks/s. This difference is due to the blocks that have been received while the decode unit was already destroyed (brake redundancy). Indeed when *UdpReceiverMgr* receives a block it checks if the decode unit is still existing (which means that the segment has not been entirely received yet). If it does not exist the block is discarded which incurs speed loss between the received blocks and the blocks accepted in the block list. *MainProcThread* picks up a block in the block list at a speed of 116 blocks/s and treats it in a decode unit at a speed of 115 blocks/s. This difference is only due to a design fault. Indeed, after a segment is received, the block list is cleaned of the useless blocks before the decode unit is destroyed. Between the time the list is clean and the decode unit is destroyed new useless blocks can arrived. They are stored in the list since the decode unit still exists. Thus, *MainProcThread* has to filter these redundant blocks. In the further tests, the cleaning will be performed after the decode unit has been destroyed. This way, no redundant blocks will be accepted in the block list after the segment has been decoded and the list has been cleaned. All these results are illustrated in Figure 19.

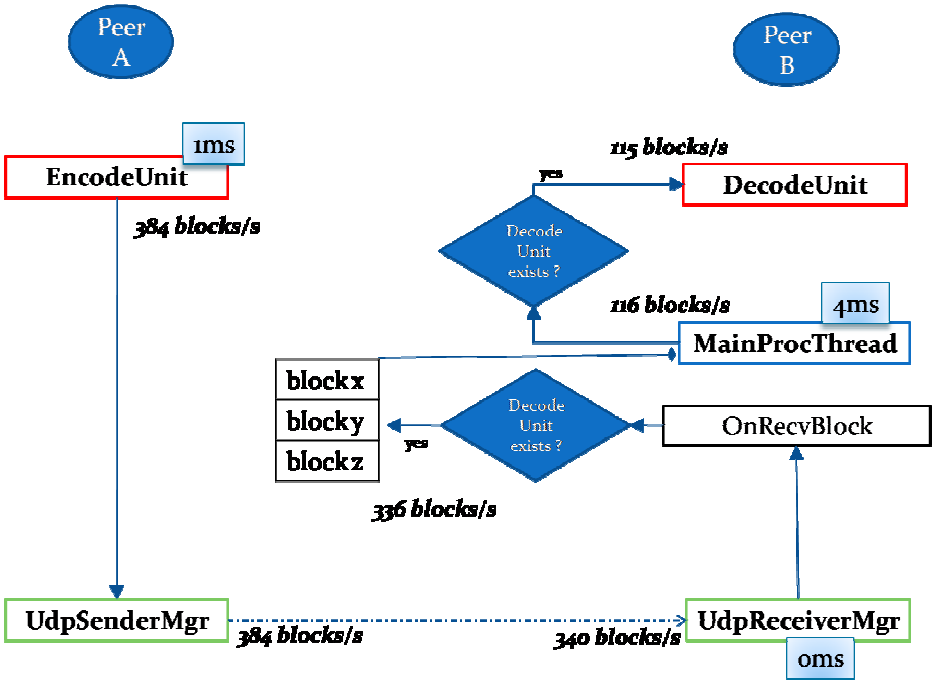

Figure 19 : block flow at different key points of a block transfer between peer A and peer B (second implementation).

How to explain that on the 384 blocks sent every second, only 340 will be received and 115 will be decoded? One hypothetic reason is that there is not synchronization between the sending peer and the receiving peer. To validate this hypothesis we change the value of the threads sleeping time. In the first test *MainProcThread* sleeps 4ms more. The results in Figure 20 show that the sending speed is increased by 24 blocks/s and the receiving speed is increased by 43 blocks/s. On the other hand, the decoding speed decreases by 36 blocks/s.

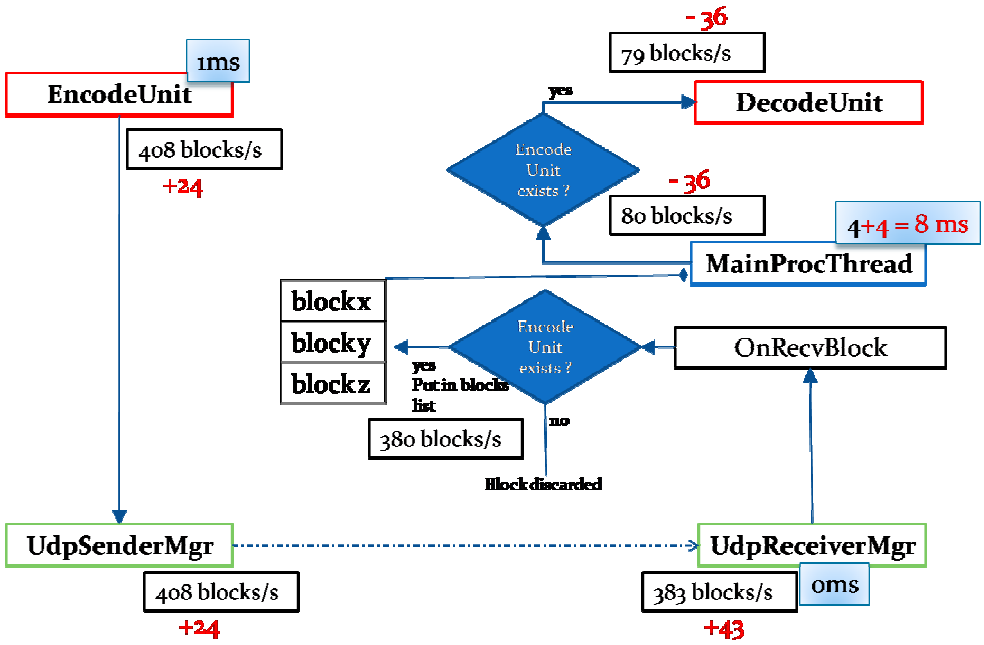

Figure 20 : block flow at different key points of a block transfer when the decoding process is slowed down for the profit of the sending and receiving speed. The numbers in red indicates the difference with the results obtained in Figure 19.

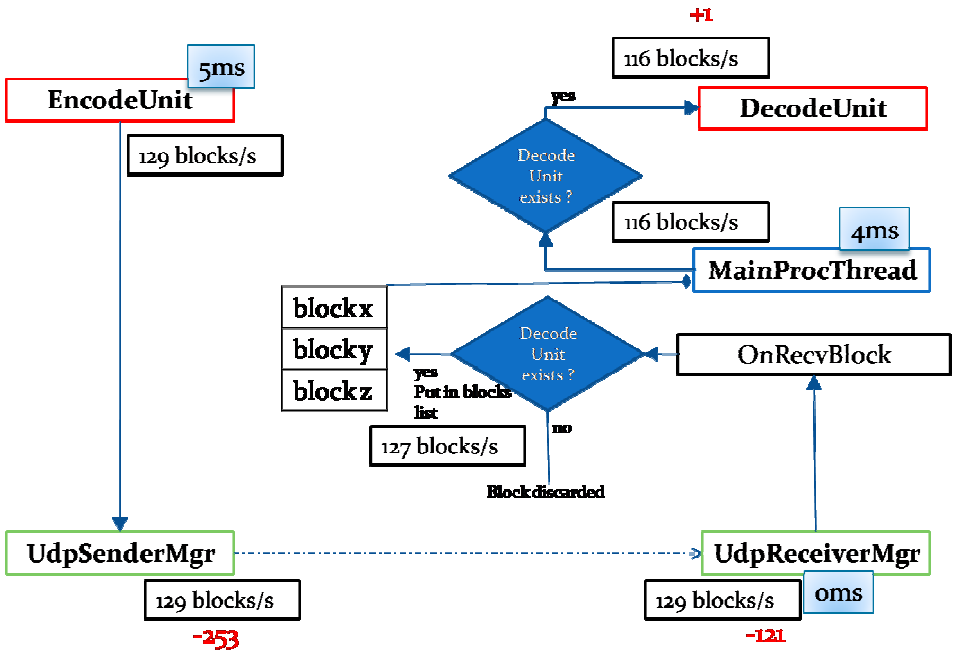

Figure 21 : block flow at different key points of a block transfer when the sending and decoding speed is slowed down at the profit of the decoding speed. The numbers in red indicates the difference with the results obtained in Figure 19.

In the second test the *EncodeUnit* sleeping time is set to 5ms (Figure 21). Slowing down the sending speed results in a synchronization between peer A and peer B which both send and receive blocks at the same speed. The decoding speed is improved only by 1. This improvement is minor but the number of blocks wasted on the network is largely reduced while the decoding performances are not harmed.

These tests show that *UdpReceiverMgr* uses resources that could be used by *MainProcThread* to decode faster. Even if it seems paradoxical, we need to decrease the sending and the receiving speed in order to decode faster while saving network resources.

Now we have an operable system that works between two computers. It is time to test it in an environment involving more than two peers where peers can connect to the system and leave it at any time.

4.4 Test in a P2P environment

This test involves a computer running on an Intel Core Duo T8100 2 GHz and 8 other identical computers running on an E2180 Intel Pentium dual 2 GHz. The test is realized in the laboratory LAN. The movie rate is 700 kbps which is equivalent to 85KB/s (one segment is 320*960 = 300KB long). The source server provides non-coded segments through TCP connexions. A peer will not required a segment from the server if it finds 3 other peers able to provide him. In the other case, the segment will be downloaded from the server. The maximum number of children that a peer can provide is set to 5. *EncodeUnit* sleeps 3 ms after one block is sent, *MainProcThread* sleeps 4 ms and *UdpReceiverMgr* sleeps 0 ms while blocks are in the UDP stack.

At t_1=09:58:05, 3 computers enter in the system and watch the same movie from the beginning. As no other peer can provides segments they download segments from the source server. The six other computers enter in the system at different time. Figure 22 shows a snapshot of the overlay at time 10:07:52.

The good news is that no peer observes buffering latency. Except the two first segments that are considered as urgent since the peers need to play them as soon as they enter the system, the other segments are downloaded one by one. The average speed to download a segment and decode it is 1642 ms (194 blocks/s). We cannot measure the decoding speed apart of the sending speed because of measure constraints due to the precision of the function we use to measure the time. However, the block flows we measured in the former tests include the decoding and the sending time. So we can compare the result with the previous tests and claim that it is quicker than in our previous test. It may be due to the *EncodeUnit* sleeping time which is shorter. Furthermore, the previous test was performed with two different computers (the Intel Core Duo T8100 2 GHz and one E2180 Intel Pentium dual 2 GHz). In the current test, much interacting computers are E2180 Intel Pentium dual 2 GHz which may alter the decoding speed. The CPU resources used by PPlayer on each computer is 50% (25% when *Network Coding* is not implemented). To evaluate the decoding speed, we do the same test without coding and decoding (the peers arrival and departure are different). We get an average time for sending one segment of 492 ms (650 blocks/s). The decoding speed

can be estimate to 1642 - 492 = 1050 ms for one segment which is much more than the tests realized outside of PPlayer.

Only one user, Li Yang Yang, receives two non-innovative blocks which is equivalent of a ratio of 0.001% (Table 2). In UUSee (7) the ratio of non-innovative block on the total number of sent blocks is 0.023% but the environment and the measuring time are much more important which prevents us from comparing the two results.

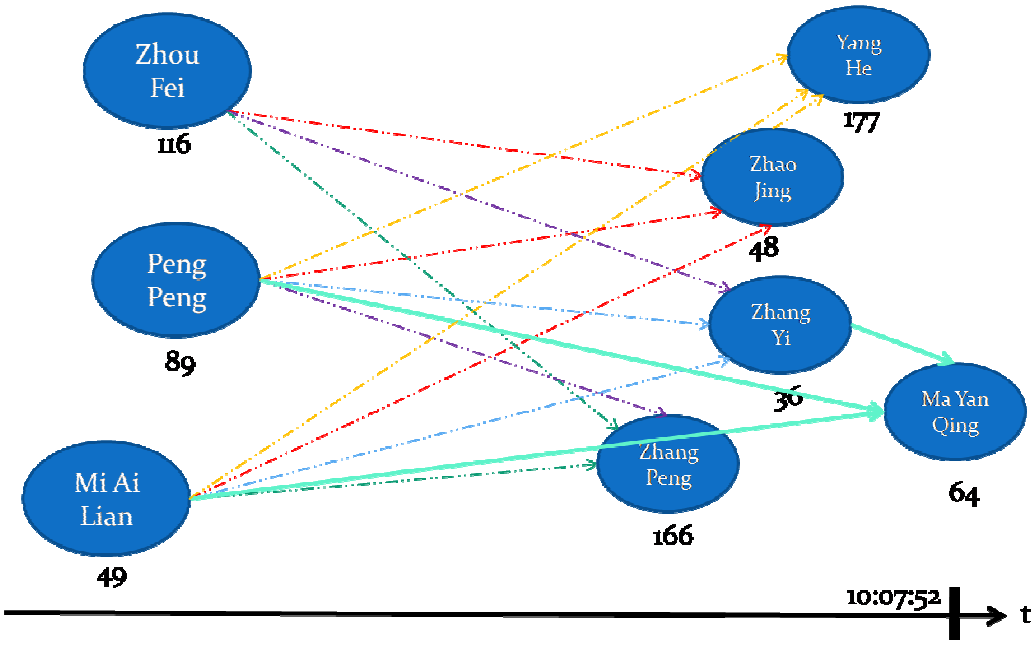

Figure 22 : Snapshot of peers overlay at time 10:07:52. The computer users and the last 8 bits of the IP address are indicated. The arrows show the block flows between peers.

User	Number of segments downloaded from other peers	Downloading and decoding speed for 1 segment (ms)	Dependency (number of non-innovative block received)
Li Yang Yang	83	1559	2
Zhang Peng	84	1533	0
Zhao Jing	84	1608	0
Zhang Yi	84	1573	0
Ma Yan Qing	169	2077	0
Yang He	112	1504	0

Table 2 : the number of segments downloaded during the test session (left), the average time for downloaded and decoding one segment (middle), the number of segment which some block are non-innovative (right).

What draws our intention is the capability for a peer to provide different children. In Section 4.1, due to the CPU resources we have identified the number of children being served as a challenge. It appears that in a real environment where peers are collaborating, a peer does not need to sustain the movie rate while uploading blocks. For example, Mi Ai Lian computer sends blocks at a rate of 206 blocks/s. However, she can serve 5 children without altering their viewing experience. *Network Coding* appears more scalable that it seemed to be. Unfortunately, we did not have the time to test it in a large-scale environment.

The brake redundancy is similar to the brake latency in (7). Indeed, we get an average of 4 blocks on each channel for each required segment (2 blocks in UUSee). But we have another kind of redundancy much more important than the brake redundancy. As there is no synchronization between the decoding speed and the sending speed of two peers, the receiving peer will receive much more blocks that it actually needs before to send a CANCEL SEGMENT message. This redundancy can vary from 0 to more than 180 blocks by segments. It depends on the load on the parent peer. The more the parent has children, the slower it sends blocks and the smaller is this redundancy.

Even if Network Coding avoids sending redundant segments in the network, it implies huge overhead. Indeed, one UDP packet's overhead is composed of the channel ID (32B), the segment ID (2B), the user ID (4B), the seed (4B) and the block ID (4B). The overhead for each packet is 46B long which results in an overhead of 14720B for 320 blocks sent. One segment is 300KB long. Hence, the overhead ratio is 4.8%. In the original system we have an overhead of 39B for one segment sent. Segments being 100KB long we have an overhead ratio of 0.0381%. The overhead implied by Network Coding compared to the original system is very important.

Let determine which quantity of data would have been downloaded in the original system and let compare it to the system using network coding. In our test 616 segments have been downloaded from other peers through coded blocks. The number of providing peers being set to 3, 1848 segments would have been sent in the network which is equivalent to 541.4 MB. The overhead ratio being 0.0381% the whole data sent would have reached 541.6 MB. In the system using Network Coding, if we suppose that synchronization is implemented between the peer sending the blocks and the peer receiving and decoding them, we can ignore the redundancy implied by non synchronization. Considering only the brake redundancy of 4 blocks on each of the three channels for each segment transfer, 616 segments transferred implied 7392 redundant blocks which is equivalent to 7.1 MB. If we consider the overhead ratio of 4.8% the data amount needed to transmit 616 segments is 189.1MB. The overall data amount is 196.2MB which is only 36% of the data amount needed in the original system.

We have put in evidence the save of network bandwidth incurred by *Network Coding*. However, the CPU resources used being important, the rate at which media data is provide to the media player (which include the downloading and the decoding process) is slower than in the original system which do not need to decode segments. Indeed, the media data is downloaded then decoded at a speed of 182KB/s. In the original system, the downloading speed can reach 500KB/s. In (6), they make the same observation but notice that if there is a close match between the bandwidth supply (network capacity) and the bandwidth demand (movie rate) the system using *Network Coding* allows a peer to buffer media data faster than in the same system not using *Network Coding*. This is due to the fact that the redundancy is lower with *Network Coding* and several peers will provide the segment.

In a future time, we need to test our system in a larger environment involving more peers and network constraints to put more in evidence the advantages of using *Network Coding*. Nevertheless, this test has confirmed that the system could be as robust as the original one while reducing the segment redundancy. It has also raised some weaknesses of Network

Coding as the computing complexity or the large overhead ratio. In the next part we propose a solution to undertake the complexity dilemma.

5 FURTHER IDEAS FOR IMPROVEMENTS

In the last part we have tested PPlayer with an implementation of *Network Coding* in a micro environment. Even if extrapolating the results to a large-scale environment seems difficult, we got several results justifying further researches to improve our system. First of all, the speed for decoding segment is very slow. Our system works well with standard computer. But low-CPU devices as PDA, set-top-box or smart phone cannot stand for *Network Coding* decoding complexity. This can be improved by modifying the encoding scheme allowing a faster decoding process. Secondly, we can improve the number of children being served by one peer by reducing the number of simultaneous encoding processes. These improvements are aimed to make low-CPU devices sustain *Network Coding* computation complexity without harming *Network Coding* efficiency.

5.1 Fast decoding Network Coding for low-CPU devices

Different researches have tried to deal with the decoding complexity of *Network Coding*. In (22), the authors propose a sparse network coding approach with overlapped classes. The idea is to gather blocks into classes and authorized linear combination only between blocks into the same class. Furthermore the classes can overlap; it means that blocks can belong to different classes. Blocks from decoded classes can be substituted into still undecoded classes. The backward of such a coding technique is that it leads to more overhead to deal with the different classes. However, it could be worthy to test it in the future to weigh the benefits and disadvantages of such a technique in our system.

We propose another way to deal with the complexity. The idea is to send blocks that does not need to realize the step 1 in the decoding process. For that we send coefficients in an half decoded form (Figure 23).

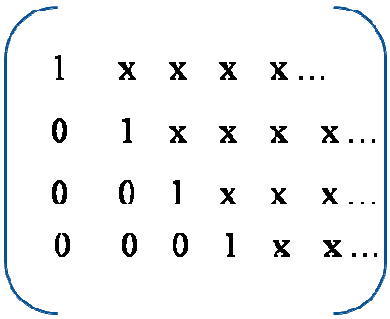

Figure 23 : coefficients in a half decoded form

We measured the encoding and decoding speed for different size segments in the same condition that in part 3.3.2.2 (Figure 24). The encoding speed is quite similar to the speed we had with fully random blocks. On the other hand, the decoding speed is two time faster for each segment size. The first problem we can identify before to test it in PPlayer is that the blocks are sent through UDP. If one packet is lost, the lost line can be recoverable only through the lines generated on the top (which can be reduced). If we send the blocks in a cycle way from the top to the bottom of the matrix (generating new coefficients each time a line is

re-sent to get innovative blocks), the lost of the lowest line in the matrix will be more quickly recoverable that the line on the top.

As our system is not tested on internet but on a LAN, we observed in the previous tests that the probability to lose a packet is very low. Hence, not paying attention to the drop of UDP packet on the network we test our solution in PPlayer with the same users that in the previous tests. Unfortunately, the decoding speed is not improved. It comes from the fact that different peers produce half decoded blocks without any coordination. Thus, the requesting peer get half decoded coefficients belonging to the same line in the matrix and need to perform a reduction (step 1 in the decoding process) which would have been unnecessary if only one peer was providing the blocks. A solution to this problem is to implement a protocol between the providing peers in order that they collaborate in a coordinated way. Each provider has to be assigned a set of lines in the matrix that it will provide in a cycle way. This solution is similar to the overlapped classes presented in (22) and can be assimilated to imbricated classes.

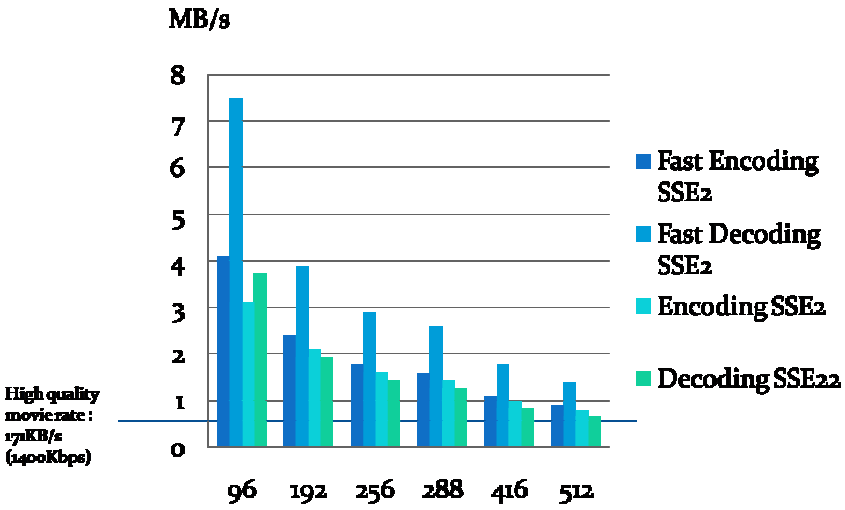

Figure 24 : encoding and decoding speeds of half decoded blocks compare to the encoding and decoding speeds of fully random blocks

5.2 Improvement of the number of children being served

When a peer is in charge of providing segments to different children the encoding load can quickly get very heavy. To avoid delay induced by encoding process one solution is to permit some slow peers to save the encoded blocks received by a peer. This way, the peer can hold the encoded blocks in its memory and send them when they are requested. The encoding process is longer as the peer that has originally encoded the segment had to check the independence between coefficient rows (which is equivalent to perform the step 1 in decoding process described in part 3.3.1.1). Yet, such a way to exchange blocks impoverishes the block diversity in the network growing the number of non-innovative blocks.

To deal with this problem, a peer can after decoding the segment re-encode it with its own random coefficients. On one hand, CPU resources are constantly used as blocks are received but on the other hand, the blocks being already encoded, the peer is more robust to a

flash crowd scenario where a lot of peers would ask for different segments at the same time. In such a design a peer is ensured to need to sustain not more than two *Network Coding* processes (even in flash crowd scenario): one decoding process to watch the segment it is downloading and one encoding process (with linearly dependency check) to store encoding blocks.

Such a design implied to double the memory use. Indeed, the data are stored under encoded and decoded forms. In order to avoid such memory waste, the freshly decoded segments are put in a buffer which provides the media player. After being played the segments are deleted from the buffer and encoded before to be stored in the memory (Figure 25). If a peer wants to replay a part of the movie recorded on its memory, it will have to decode it as the decoded segments are not stored anymore (Figure 26). The number of processes keeps unchanged as it is always no more than one decoding and one encoding process at the same time.

What could be seen as a weakness of such an implementation is that the number of blocks generated and sent is limited. On unreliable communication channels where blocks can be dropped a child that has not receive the n blocks can ask for more than the number of blocks stored on one parent. The parent does not know which blocks have been lost. Therefore it needs to generate new blocks from the segment. In the memory we only have coded blocks. Hopefully, one of the strong propriety of random *Network Coding* is that we do not need to encode from the original segment to get new innovative blocks. New linear combinations between blocks will produce innovative blocks. But it means that two seeds will be sent to the requesting peer in order for it to compute the product of the two linear combinations applied to the blocks.

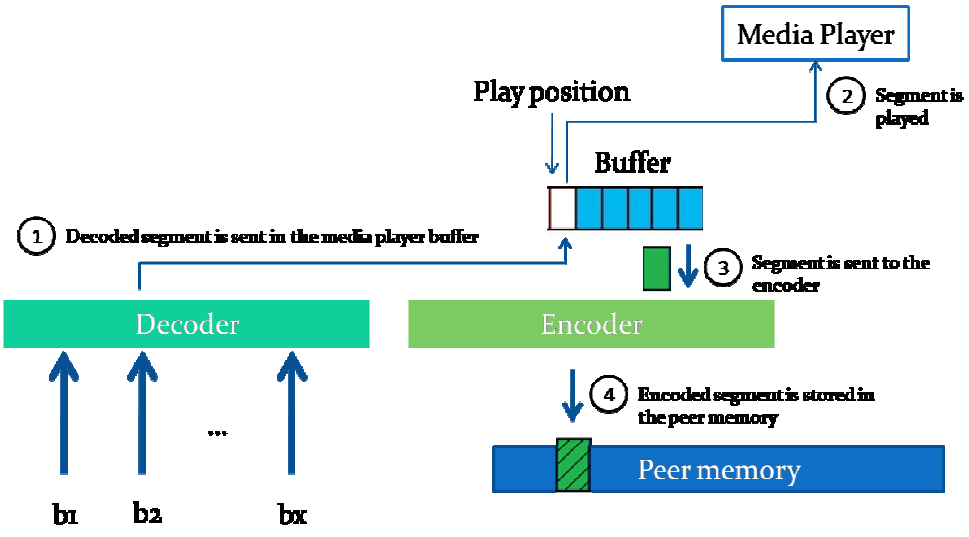

Figure 25 : model of a solution to alleviate the peer load due to children (download segment operation)

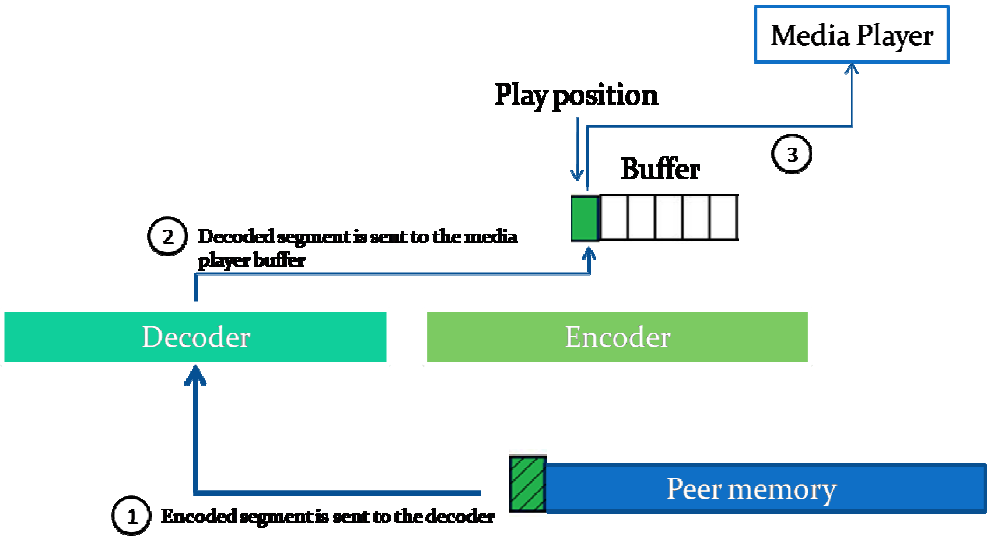

Figure 26 : model of a solution to alleviate the peer load due to children (VCR operation).

6 CONCLUSION

We have implemented a *Network Coding* module in an existing P2P-VoD system called PPlayer. It first permitted us to evaluate the different parameters to take into account when *Network Coding* is used in classical systems. Secondly, the tests we performed highlight the capacity of such a system to be scalable by permitting peers to collaborate into providing a child and by assuring robustness while limiting the data redundancy. However, the encoding and decoding complexity is a problem that still needs further investigations. The solutions proposed at the end of the report improve the decoding speed and the rate at which encoded segments are provided but still need to be tested in a real P2P environment. In the future, further test should be performed to validate them. Other weaknesses have not been dealt in the report as the brake redundancy or the synchronization between sending and receiving peers. The natural next step of our research would be to run PPlayer on the internet or in an environment simulating it and analyze the advantages of using *Network Coding* in such a context.

7 APPENDIX

7.1 The Jordan-Gauss implementation in C++

nblocks : number of blocks in one segment.
m_block_size : block size.
coeff[nblocks] : newly received coefficient row (generated from the seed).
block[m_block_size] : newly received block.
m_coeff[nblocks][nblocks]: matrix containing the coefficient already received.
m_blocks[nblocks][m_block_size]: matrix containing the blocks already received.

```
bool CDecoder::decodeSegment(char* block, char* coeff)
{
//element 1, 2, 3 are used to operate operation in GF(2^8)
galois::GaloisFieldElement element1(gf, 0);
galois::GaloisFieldElement element2(gf, 0);
galois::GaloisFieldElement element3(gf, 0);
static int n = 0;      //number of coefficient rows already inserted
int lv = 0;              //position where the new coefficient row has to be
inserted
bool dependent = true;  //flag : true indicates that coeff is linearly dependant
//with m_coeff matrix rows.

//Step 1: coeff is reduced until it can be inserted in the matrix
while(lv < nblocks)
{
    if(m_coeff[lv][lv] == 0 && coeff[lv]!=0)
    {
       m_coeff[lv] = coeff;
       m_blocks[lv] = block;
       dependent = false;
       gap[lv] = 1;
       break;
    }
    else if(coeff[lv] == 0)
    {
       lv++;
    }
    else
    {
       //linear combination between coefficient rows
       element1 = coeff[lv];
       for(int i=lv; i<nblocks; i++)
      {
          element2 = m_coeff[lv][i];
          element3 = coeff[i];
          coeff[i] = (element3 - (element1*element2)).poly();
       }
      //linear combination between blocks
      for(int j=0; j<m_block_size; j++)
      {
           element2 = m_blocks[lv][j];
           element3 = block[j];
           block[j] = (element3 - (element1*element2)).poly();
       }
```

```
       lv++;
    }

}

    if(dependent) //the coefficient row has not been inserted because not
innovative
    {
       return false;
     }

       //Step 2 : the new coefficient row is divided by its pivot
//Operation on coefficient rows
element1 = m_coeff[lv][lv];
for(int i = lv; i< nblocks; i++)
{
    element2 = m_coeff[lv][i];
    m_coeff[lv][i] = (element2/element1).poly();
}
//Operation on block rows
for(int j=0; j<m_block_size; j++)
{
    element2 = m_blocks[lv][j];
    m_blocks[lv][j] = ((element2/element1)).poly();
}

//Step 3 : Setting to 0 the whole column lv and the other columns that has not been
reduced because of gaps.
for(int i=lv; i<nblocks; i++)
{
//check if there is coefficient row missing on the top of the ith row
bool noGap = true;
 for(int k = i; k>-1; k--)
{
    if(gap[k] == 0)
    {
       noGap = false;
       break;
    }
}

//if no gap (missing row) we can operate the reduction
if(noGap)
{
    for(int k = i-1; k>-1;k--)
   {
        //linear combination between coefficient rows
       element1 = m_coeff[k][i];
       for(int j=lv; j<nblocks; j++)
      {
           element2 = m_coeff[i][j];
           element3 = m_coeff[k][j];
          m_coeff[k][j] = (element3 - (element1*element2)).poly();
       }
      //linear combination between blocks
        for(int j=0; j<m_block_size; j++)
       {
          element2 = m_blocks[i][j];
          element3 = m_blocks[k][j];
          m_blocks[k][j] = (element3 - (element1*element2)).poly();
       }
     }
```

```
  }
  else   //if coefficient row are still missing above the new inserted one we skip
step 3
  {
       break;
  }
}

n++;
if(n==nblocks)
{
   decoded = true;
}

return true;
}
```

7.2 The GF(2^8) multiplication SSE2 acceleration implementation

The function *linearCombination(row1, row2, factor, rowSize, result)* performs row1+row2*factor with row1 and row2 two char vector of size rowSize. The result is store in the row result. The comments in the code indicate the equivalent when working not on 16 bytes variables (type *__m128i*) but one byte variables (type *byte*).

```
void CDecodeUnit::linearCombination(char* row1, char* row2, char factor, int
rowSize, char* result)

{

       int nLoop = rowSize/16;

       __m128i* pResult = (__m128i*) result;
       __m128i* pRow1 = (__m128i*) row1;
       __m128i* pRow2 = (__m128i*) row2;
       __m128i row1cp;
       __m128i resultcp;

       __m128i un = _mm_set1_epi8(1);
       __m128i zero = _mm_setzero_si128();
       __m128i mask;
       __m128i overflowing;
       __m128i over = _mm_set1_epi8((char) 0x80);
       __m128i poly = _mm_set1_epi8((char) 0x1D);
       __m128i m1;
       __m128i m2;

       for(int i = 0; i<nLoop; i++)
       {
              __m128i mfactor = _mm_set1_epi8(factor);
              row1cp = _mm_loadu_si128(pRow1);
              resultcp = zero;

              while(_mm_movemask_epi8(_mm_cmpeq_epi8(zero, row1cp)) != 65535)
              {
                    //if x&1!=0 ->> 0xff
                     mask = _mm_cmpeq_epi8(_mm_and_si128(un, row1cp), un);
                    //mask on the factor array
                     m1 = _mm_and_si128(mask, mfactor);
                   //result = y ^ result
                    resultcp = _mm_xor_si128(resultcp, m1);
                   //bool overflowing = (y & 0x80)
                   overflowing = _mm_cmpeq_epi8(_mm_and_si128(mfactor, over), over);
```

```
                //  y = y << 1
                 mfactor = _mm_andnot_si128(un,_mm_slli_epi32(mfactor, 1));
                //if overflowing = true

                  m2 = _mm_and_si128(overflowing, poly);
                 //{y = y ^ poly}
                  mfactor = _mm_xor_si128(m2, mfactor);
                // x = x >> 1
                  row1cp = _mm_andnot_si128(over,_mm_srli_epi32(row1cp, 1));
            }

            //addition
            *pResult = _mm_xor_si128(resultcp, *pRow2);
            pRow1++;
            pRow2++;
            pResult++;
        }
}
```

8 REFERENCES

1. **Kevin.** [Online] [Cited: Jun. 28, 2010.] http://www.pudn.com/downloads97/sourcecode/p2p/detail394778.html.

2. **Yan Huang, Tom Z. J. Fu, Dah-Ming Chiu, John C. S. Lui and Cheng Huang.** Challenges, Design and Analysis of a Large-scale P2P-VoD System. *ACM SIGCOMM Computer Communication Review.* 2008. study of PPlive.

3. **B. Cheng, L. Stein, H. Jin, Z. Zhang.** Towards Cinematic Internet Video-on-Demand. *EuroSys'08.* 2008. Description of GridCast.

4. **R. Ahlswede, N. Cai, S.-Y. Li, and R. Yeung.** Network Information Flow. *IEEE Trans. Inf. Theory.* Jul. 2000, Vol. 46, 4.

5. **C. Gkantsidis, J. Miller, and P. Rodriguez.** Network Coding for Large Scale Content Distribution. *IEEE Infocom 2005.* 2005.

6. **Mea Wang, Baochun Li.** Lava: A Reality Check of Network Coding in Peer-to-Peer Live Streaming. 2006.

7. **Zimu Liu, Chuan Wu, Baochun Li, and Shuqiao Zhao.** UUSee: Large-Scale Operational On-Demand Streaming with Random Network Coding. 2008.

8. **Chinneck, John W.** Practical Optimization: a Gentle Introduction. [Online] 2000. http://www.sce.carleton.ca/faculty/chinneck/po/Chapter9.pdf.

9. **Shuo-Yen Robert Li, Senior Member, IEEE, Raymond W. Yeung, Fellow, IEEE, and Ning Cai.** Linear Network Coding. *IEEE TRANSACTIONS ON INFORMATION THEORY.* February 2003, Vol. 49, 2.

10. **T. Ho, M. Medard, J. Shi, M. Effros, and D. Karger.** On Randomized Network Coding. *Proc. of Allerton Conference on Communication,Control, and Computing.* 2003.

11. [Online] http://en.wikipedia.org/wiki/Finite_field.

12. [Online] http://en.wikipedia.org/wiki/Finite_field_arithmetic.

13. **Siddhartha Annapureddy, Saikat Guha, Christos Gkantsidis, Dinan Gunawardena, Pablo Rodriguez.** Is High-Quality VoD Feasible using P2P Swarming. *IW3C2.* 2007.

14. **Dah-Ming CHIU, Raj JAIN.** Analysis of the Increase and Decrease Algorithms for Congestion Avoidance. *Computer Networks and ISDN Systems.* 1989, Vol. 17.

15. **Gu, Yunhong.** [Online] Octobre 2005. http://udt.sourceforge.net/doc/udt-2009.ppt.

16. *Numerical Recipes in C: The Art of Scientific Computing.* s.l. : Cambridge University Press. sample pages. ISBN-10: 0521880688.

17. **Fog, Agner.** Pseudo random number generators. [Online] http://www.agner.org/random/.

18. **Partow, Arash.** Galois Field Arithmetic Library. [Online] http://www.partow.net/projects/galois/.

19. **Hassan Shojania, Baochun Li.** Parallelized Progressive Network Coding With Hardware Acceleration. 2007.

20. **Intel Corporation.** *IA-32 Intel® Architecture IA-32 Intel Architecture Optimization Reference Manual.* 2006.

21. **Microsoft.** msdn. [Online] http://msdn.microsoft.com/en-us/library/y0dh78ez%28VS.80%29.aspx.

22. **D. Silva, W. Zeng, and F. Kschischang.** Sparse Network Coding with Overlapping Classes. *Proc. NetCod'09.* June 2009.