Dublin City University

Final year project
Technical Documentation

Design and implementation of a Performance-Enhancing Proxy for connections over 3G networks

Author:
Gregoire Delannoy

Supervisor:
Darragh O'Brien

May 27, 2013

Abstract

The aim of this project is to improve the quality of experience for users connected to Internet via wireless networks. As for now, most of our Internet usage rely on a protocol designed nearly 40 years ago for wired networks : TCP. It has numerous drawbacks when used over a wireless networks, as segments can be lost randomly and the delay can vary significantly. Research teams have been working on alternatives or modifications to the TCP protocol in the past 15 years. Usually protocol modifications can only be simulated, as we need a wide acceptance in the Internet infrastructure for them to be really used. The goal with this project was to design and develop a proxy infrastructure – allowing a real-world use –, in conjunction with a new network protocol : Coded-TCP. This new protocol has been proposed in 2012 by a team at MIT to solve the low performances problem of TCP over packet-erasure networks. The protocol has been implemented and the system has been deployed to the proxy system. It can now be used to enhance performances of mobile devices such as phones or laptop connected to packet-erasure networks like some 3G networks or WiFi hotspots.

Contents

1 Motivation **3**

2 Research **3**
- 2.1 Investigating the problem . . . 3
  - 2.1.1 Academic research . . . 3
  - 2.1.2 Personal research . . . 4
- 2.2 Review of different possibilities . . . 5
- 2.3 Preferred solution : Splitted + Network Coding . . . 5

3 Design **6**
- 3.1 System Architecture . . . 6
- 3.2 Client $\Leftrightarrow$ Proxy Protocol . . . 7
  - 3.2.1 Block Systematic Network Coding . . . 7
    - Network Coding Theory . . . 7
    - Application over a finite field . . . 8
    - Block Network Coding . . . 9
    - Systematic Coding . . . 9
  - 3.2.2 Communication Protocol . . . 10
    - General format : multiplexer level . . . 10
    - Message types . . . 10
      - Data packet . . . 11
      - ACK packet . . . 11

4 Implementation and Problems solved **12**
- 4.1 Methodology . . . 12
- 4.2 Finite-Field operations . . . 13
- 4.3 Matrices . . . 14
- 4.4 Random Linear Network Coding . . . 15
  - 4.4.1 Coding . . . 15
  - 4.4.2 Decoding . . . 16
- 4.5 Communication Protocol . . . 17
- 4.6 Networking Issues . . . 18
- 4.7 Congestion Control . . . 19

5 Results **20**
- 5.1 Test results . . . 22
  - 5.1.1 Simulation . . . 22
  - 5.1.2 Real-World . . . 22

6 Future Work **23**

7 Bibliography and Links **24**

1 Motivation

There has been two main motives to start this project: At first the problem of adapting *old* network protocols, such as TCP and IP, to new types of links with completely different loss patterns and mobility came to my sight when I was designing a Wifi on board infrastructure for a train company. They were using 3G links to provide connectivity between the train and the ground, and we quickly became aware of the limitations of these connections when we used it : the effective TCP throughput was orders of magnitude lower than the advertised data rate. I started to look at potential solutions, but it did not fit into the frame of the internship.

Later on when moving to Dublin, the problem appeared again as I have been using a mobile data connection as my only Internet access. Given the freedom offered by the School of Computing regarding the choice for the final-year project, I decided to take it as a good occasion to tackle this problem down.

2 Research

As I started searching into the possible improvements for User Experience over mobile connections, I stumbled upon a wide range of research papers focusing on the problem. It seems to be a widely accepted fact that the resulting performances of TCP connections matter greatly, as it is the most widely used protocol out there.

2.1 Investigating the problem

As for now, most of my motives were only feelings, or what I thought to be general knowledge over TCP behavior and wireless links. It appeared that the issue is a lot more complicated and that the most widely used implementations of TCP already perform quite well

2.1.1 Academic research

A good summary of the problems faced by TCP connections over 3G links can be found in [6]. According to the author, the relevant differences between traditional wired and modern wireless/cellular networks come from the following :

- Bit-Error Rates (BER) are in the order of 10^{-10} to 10^{-12} in wireline networks, whereas the usual target in wireless networks is 10^{-2} which leads to much higher Frame-Error Rates (FER). It can cause timeouts, or reduction in the window size because the loss was interpreted as congestion
- Losses often occur in burst, which even worsen the previous point

- Handover (e.g. going from one cell to the other) is relatively slow compared to the data rates expected. This will lead to buffer overflows, re-ordering and sudden delay changes
- Rate and Delay vary widely over time, while most TCP implementations assume these parameters to be stable

As for the perceived effects of these differences, I will refer to two real-world studies : Goodput measurements performed over a 3G link [8], and Download completion times over public Wifi hotspots [5].

From these studies, we can see that the picture is not really as expected: in the 3G world, a lot of attention has been drawn to this problem and lower-layer (physical or link) take care of shielding the higher levels from the losses. Mechanisms such as Hybrid-ARQ (See [7] for details) are quite efficient and the main cause for performances degradation appeared to be the initial resource allocation performed at the Base Station.

However it is important to note that this study was performed in 2005, while unlimited data plan were completely nonexistent and use was quite low, over a single equipment, in a single network.

The other study has been performed in the framework of the Coded-TCP testing: they compared their new protocol with the standard Linux implementation of TCP (i.e. CUBIC with SACK enabled). According to their experiments, the new CTCP performs roughly 2.5 times better in this particular loss-heavy environment.

2.1.2 Personal research

As explained in more details over the blog post, I captured my network traffic over multiple kind of Internet connections:

- Wired WAN network : from an ADSL broadband link, and from the school of computing wired network to Internet servers. Average FER = 1.45%
- Wired LAN : on my home LAN. Average FER = 0.0%
- Wireless 3G link : Lycamobile from Glasnevin, with good coverage. Average FER = 3.2%

We can see easily that loss is quite common compared to TCP's design considerations. We should be careful about these results, as the measurement method (at the client side) is not absolutely accurate as there is no way to know if a packet has been lost and already re transmitted, out-of-order delivery was neglected, etc...

Another interesting finding is the omnipresence of selective acknowledgment (SACK), which improves over the baseline TCP where the sender was forced to re transmit all the data from the loss event.

2.2 Review of different possibilities

The most useful and comprehensive review of all the potential approaches can be found in [4]. Here is a summary of the taxonomy :

- Link-Layer approaches
  - Automatic Repeat reQuest (ARQ): Thanks to an acknowledgment scheme, losses can be compensated for, to the desired FER
  - Forward-Error Correction (FEC) : Redundancy bits are added in order to reduce the FER while the BER stays high. It can be made adaptive to the channel
  - Hybrid-ARQ : Combines the best of both, re transmitting an encoded version of the lost packet
  - *Additionally these schemes can be made Transport-Layer aware, in order to block TCP retransmissions of an already re transmitted (via ARQ) segment for example*
- Transport-Layer approaches
  - End-To-End
    - SACK
    - Changes in the congestion-control algorithm: newReno, Vegas, westwood...
    - Explicit Congestion Notification (ECN): Instead of relying on losses to infer congestion, the sender could rely on a message from intermediate routers that announce a congestion event
  - Splitted : The TCP session is split around an intermediate node, which communicates with a new protocol over the wireless link, and with normal TCP on the wired Internet network.

2.3 Preferred solution : Splitted + Network Coding

In March 2011, a MIT team led by Muriel Médard proposed a new mechanism called TCP/NC [9] (for Network Coding, which will be explained later on) aimed to address the poor performances of TCP in wireless networks. Their results hit the press, as they claimed performances 10 times higher than standard TCP in real-world challenged network environments.

This mechanism could be implemented as End-To-End but it would imply changes on the server TCP stack, which was out of question. Therefore, I decided to go for a splitted scheme, with the client communicating with TCP/NC to the proxy, that would decode it and relay in standard TCP to the server.

The architecture proved to be a good idea, but TCP/NC was a poor candidate, because it was not designed to accommodate for losses outside of its control (i.e. on the proxy $\Leftrightarrow$ server link).

Thankfully, another protocol has been developed in 2012 by a PhD student in the same team : Coded-TCP (CTCP) [10]. It is a complete Transport-Layer protocol which use a block-oriented, as opposed to sliding-window, systematic network coding. I ended up implementing it, with some slight modifications, and using it for the communications between the client and the proxy over the lossy link.

3 Design

3.1 System Architecture

Most of the design for the system architecture has already been explained in the Functional Specification document, that you can refer to if needed. Here is a quick summary.

At first, let's define the terminology that we are going to use through the rest of the documentation :

- **Client**: The actual device used by the user. Typically, a laptop
- **Client Application**: The (unmodified !) TCP application that is running on the client machine
- **Client System**: The part of my system that lives on the client machine
- **Firewall**: The part of my system responsible for the interception of outgoing TCP connections. It consists of a simple set of iptables/netfilter rules.
- **Proxy**: The device that handles the translation from one protocol to the other. In my case, it will be a virtual machine hosted by RedBrick
- **Proxy System**: The part of my system that lives on the proxy machine
- **Server**: Some host/entity which hosts information or resources. Youtube for example
- **Server Application**: The TCP application that is hosted on the server, a Web Server for instance

The data flow within the different components is illustrated in the following diagram :

The communication between the Client Application and the Client System is standard TCP, as well as the one between the Proxy System and the Server Application.

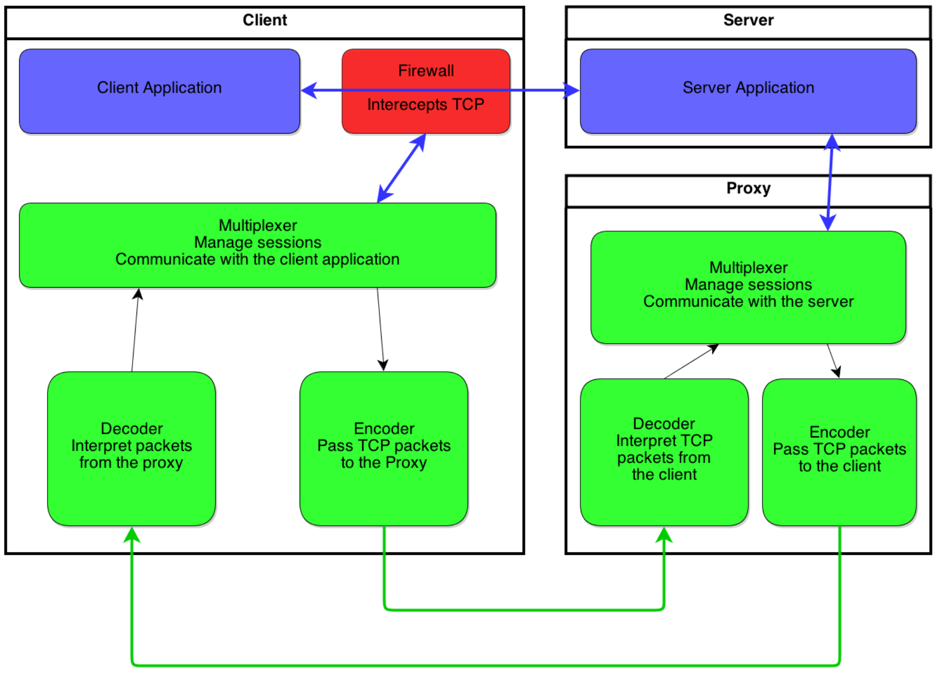

Figure 1: Data Flow within the different components of the application

3.2 Client ⇔ Proxy Protocol

3.2.1 Block Systematic Network Coding

In this section I will explain briefly Network Coding principles, from the underlying theory to the more complex block systematic scheme that is used in my system.

Network Coding Theory We want to send a set of packets $P_{i=0..N}$ from a sender to a receiver. In Linear Network Coding, the sender has to generate a set of coded packets C_j such that :

$$C_{j=0..M} = \sum_{i=0}^{N} P_i * a_{j,i}$$

Here, $a_{j,i}$ represents the coefficient of the corresponding original packet P_i in the current coded packet C_j. There are various ways of generating coefficients, to ensure the highest possible entropy in the set of C_j. However in practical cases, using random coefficients proved to be good enough.

The sender can generate any arbitrary number of packets, but if we want the receiver to retrieve the entire set of P_i, we need to have $M \geq N$ and to ensure sets of coefficients are linearly independents. The sender needs to send coded packets as well as coefficients used to encode them.

At this stage, matrix representation is more convenient :

$$\begin{pmatrix} C_0 \\ C_1 \\ \vdots \\ C_M \end{pmatrix} = \begin{pmatrix} a_{0,0} & \cdots & a_{0,N} \\ \vdots & \ddots & \vdots \\ a_{M,0} & \cdots & a_{M,N} \end{pmatrix} * \begin{pmatrix} P_0 \\ P_1 \\ \vdots \\ P_N \end{pmatrix}$$

or

$$C = A * P$$

On the receiver side, we got the coefficients matrix as well as the vector of encoded packets. To recover the vector of original packets, we need to solve the equation system. Basically, what we want to do is to find the invert of the coefficients matrix. This can be achieved using Gauss-Jordan elimination, which I am not going to detail here.

Assuming every sets of coefficients were independent, we can compute the invert matrix A^{-1} on the receiver side. Then :

We know that $C = A * P$
With the invert matrix, $A^{-1} * C = A^{-1} * A * P$
So, $P = A^{-1} * C$

Application over a finite field Now that we got valid operations for the infinite set integers, we need something for working with computers, i.e. use a finite set. We are going to use bytes as base symbol ; so we're working in the field $\mathbb{F}_{2^8}$.

Defining Operations We need to define our standard set of operations :

- **Addition** : bitwise XOR
- **Subtraction** : bitwise XOR
- **Multiplication** : two possibilities
  - Use polynomials : Choose an irreducible polynomial for this field (for $\mathbb{F}_{2^8}$, we can use Rijndael's : $x^8 + x^4 + x^3 + x + 1$).
    Then, you can multiply the two terms, represented as polynomials and perform a modulo(Rijndael's polynomial) on the result
  - Use discrete logarithm : as a faster alternative, we can compute tables for logarithm and exponentiation. *In $\mathbb{F}_{2^8}$, we end up with two 255 bytes tables.*
    Let α be a generator in our field (for example $\alpha = 0x03$). For each

non-zero number x in the field, we have $x = \alpha^{l(x)}$, with $l()$ our logarithm function.
Since $l(xz) = l(x) + l(y)$, we can compute xy very easily by simply looking up at our tables.

- **Division** : two possibilities
  - Euclidean algorithm
  - Discrete logarithms : as for the multiplication, we have $l(\frac{x}{y}) = l(x) - l(y)$. It's straightforward to use our pre-computed tables.

Block Network Coding As you might have concluded from the reading of the previous paragraph, one of the problem caused by Network Coding is the size of the matrices : the bigger the number of packets included into a linear combination, the bigger the matrices and the more complex the coding and decoding operations.

The practical solution to this problem is to divide your data in another structure than the simple packet : a block. A block will consist of a fixed number of packets. All packets within a block can be used in a single linear combination, but they cannot be combined with data from another block. Therefore, the size of each matrix will be limited to the size of the block, which makes things a lot easier.
However, it must be kept in mind that if the block is too small, the Network Coding scheme will not be able to correct loss. We must find the correct trade off between coding complexity and loss-recovery abilities.

Systematic Coding The term systematic is to be understood in its telecommunications meaning, i.e. a coding scheme in which the plain text data get send alongside the coded redundant information. So, how can it be applied to network coding ?

If we send for each packet within a block, the plain text version of the packet, we would obtain the resulting coefficient matrix :

$$\begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}$$

Therefore, if no losses occur, the receiver would not even have to perform any matrix inversion, or any complex operation. The complexity of the insertion operations are also greatly reduced.

3.2.2 Communication Protocol

While the new protocol is a complete Transport-Layer protocol, it was not practically possible to transmit it on top of the Network-Layer protocol, IP. Indeed most firewalls and routers can only recognize, and therefore transmit, standardized TCP or UDP packets. The communication will so take place on top of UDP, with one duplex UDP flow per Client System connected to the Proxy System.

In the next paragraphs I will describe the protocol, regarding the formatting and content of the messages. A similar protocol must have been developed by the team who proposed CTCP, but they never made such public. This protocol is therefore my own creation.

General format : multiplexer level The first layer for the transmitted messages is what I call the *multiplexer* level : all messages carry a header and a payload.

The message header has two goals :

- Provide a way to distinguish between different TCP streams
- Specify the type of message carried.

The data structure for the header can be found in the following table :

Field	Type	Comment
sport	`uint16_t`	Original TCP source port
dport	`uint16_t`	Original TCP destination port
remote_ip	`uint32_t`	Original destination IP address
type	`uint8_t`	Type of payload
randomId	`uint16_t`	A random identifier, used to avoid mux collisions

Message types Various messages can be transmitted over my communication protocol :

- **DATA**: From the encoder to the decoder, it carries coded data and related information, such as the coding coefficients
- **ACK**: From the decoder to the encoder, it carries information about the state of the decoder and is used to compute network delay
- **EMPTY**: Empty packet, used in certain cases for connection establishment
- **CLOSE**: Indicates that the local mux has been deleted, upon error or normal termination. The remote mux should be deleted on the reception of this message.

- **CLOSEAWAITING**: Notify the remote side that the local mux is not directional anymore. It corresponds to the unilateral FIN in TCP.

Given the list above, I am going to give further details on the two most important types :

Data packet This type of packet carry the encoded data. It is constituted of a header and a payload. The header structure is as follow :

Field	**Type**	**Comment**
blockNo	`uint16_t`	The block index for the current packet
packetNumber (first bit)	`uint8_t`	Indicates whether the packet is coded or plain text
packetNumber (seven last bits)	`uint8_t`	If the packet is coded, indicates the number of coefficients. If not, indicates the index of the current plain text packet in the block
seqNo	`uint32_t`	Always-incrementing, unique sequence number. Is also used as a seed to the PRNG when the packet is coded.

Thus, with such a simple protocol, we can notice that we embedded several limitations to the size of the transmission that can occur: the number of blocks cannot grow higher than $2^{16} - 1$; the sequence number, the total number of packets, cannot be higher than $2^{32} - 1$; the number of packets in a block is limited to $2^7 - 1$.

As we will not make the packet size higher than the standard Maximum Transmission Unit (MTU) of 1500 bytes used as a *de facto* standard on the Internet, we can compute the real values for the aforementioned limits :

- Number of packets in a block: $(2^7 - 1) * 1500 = 186kBytes$
- Total number of blocks: $(2^{16} - 1) * 186 = 11.6GBytes$
- Total number of packets: $(2^{32} - 1) * 1500 = 6000Gbytes$

Thus, the upper bound for a single transmission will be of more than 11Gbytes.

ACK packet The ACK-nowldgment packets do not carry any data, thus the entire content of the message is defined in the following table :

Field	Type	Comment
ack_currBlock	`uint16_t`	The latest block that has not been completely decoded
ack_seqNo	`uint32_t`	The sequence number from the corresponding acknowledged data packet
ack_loss	`uint16_t`	The count of loss packets, over the span of ack_total packets
ack_total	`uint16_t`	See above
ack_dofs	`uint8_t*`	An array of fixed size containing the number of received Degrees of Freedom (i.e. the number of innovative linear combination) received for each of the current blocks.

Of course, all the transmitted data is to be send in the standard network order (Big-Endian) when relevant.

4 Implementation and Problems solved

As I stated in the Functional Specification document, I started the project as a Python project, with C modules for low-level encoding and decoding and Scapy [2] to interact with the network.

Unfortunately it proved not to be usable because of the poor performances of this whole setup, and the need to interact with relatively low-level data in various parts of the program. Subsequently I switched the entire project to plain C.

In the next sections, I am going to present the software development model I used and give details on the decisive steps, including parts of the code.

4.1 Methodology

After having done most of the research work, I had to start everything from scratch without external guidance. That was the most ambitious project I've been realizing so far, thanks to a pragmatic, agile-like, development process. At each stage of the development:

1. List the forthcoming features that need to be implemented.
   e.g Perform the coding operations, given the data vectors
2. On paper, design high-level algorithm.
3. On paper, design the data structures.
   Matrix = 2-dimensional Array of arrays + size, vector = Array

4. Implement the algorithm in the code, fine tune on toy examples.
5. Use a test function to check for results
6. Assess the correctness of memory management, using valgrind [3] on the test function

This process could be summed up with the well-known mantra : *Make it work, make it right, make it fast* that fitted this project quite well. For the last step, I used the GNU profiler, gprof [1].

4.2 Finite-Field operations

As stated in the design section, I needed a way to interact on numbers in a finite field in order to perform the network coding operations. Knowing that speed would be an issue, I decided to go for the logarithm method in order to implement the multiplication and division operation.

The two files `galois_field.h` and `galois_field.c`, define and implement the four basic operations.

```
uint8_t gadd(uint8_t a, uint8_t b) {
    return a ^ b;
}

uint8_t gsub(uint8_t a, uint8_t b) {
    return a ^ b;
}

uint8_t gmul(uint8_t a, uint8_t b) {
    if((a == 0) || (b == 0)){ // One of the term is zero, the result will be
         zero. No need for lookup
        return 0;
    } else {
        /* a*b = antilog(log(a) + log(b)) */
        return atable[ltable[a] + ltable[b]];
    }
}

uint8_t gdiv(uint8_t a, uint8_t b) {
    if(a == 0){ // 0/x = 0 for all x
        return 0;
    } else if(b == 0){ // We got a division by 0 ; it should never happen !
        printf("Code tried to divide %x by %x !\n", a, b);
        exit(1);
    } else {
        /* a/b = antilog(log(a) - log(b)) */
        return atable[255 + ltable[a] - ltable[b]]; // Note : log(a)-log(b)
            can be < 0, hence the +255
    }
}
```

```c
// Returns a non-zero number
uint8_t getRandom(){
    int n = random(); // Returns a number between 0 and RAND_MAX
    while(((n & 0xFF) == 0x00)){
        n = random();
    }
    return (uint8_t)n;
}
```

The trick here resides in the pre-computed tables for the log and antilog values. These values are stored in tables that allow for a quick look-up every time it is needed.
Given the size of the field (2^8), we should need 255 values per table. However I decided to store the actual tables twice (hence 511 values per table), in order to avoid a modulo operation which would be needed otherwise. The final size for these tables is still reasonable, as they occupy roughly 1kBytes.

4.3 Matrices

The downsize of using a custom field is that I could not use the standard optimized libraries (e.g LAPACK) to perform matrices operations. Thus, I needed to implement the data structure and necessary operations.

The data structure I decided to go to is quite simple and standard for matrices, a 2-dimensional array and the number of rows and columns :

```c
typedef struct matrix_t {
    uint8_t** data;
    int nRows;
    int nColumns;
} matrix;
```

Implemented operations include generation, freeing, copying and multiplication. The latter operation required further optimization, because it took long to execute and ended up being called extremely often in the course of the program execution. Here is the final iteration of the code :

```c
matrix* mMul(matrix a, matrix b){
    int i, j, k;
    matrix* resultMatrix;
    uint8_t factor;
    uint8_t *aVector, *bVector, *resVector;

    // Check dimension correctness
    if(a.nColumns != b.nRows){
        printf("mMul : Error in Matrix dimensions. Cannot continue\n");
        exit(1);
    }

    // Create the result matrix to the correct size
    resultMatrix = mCreate(a.nRows, b.nColumns);
    for(i = 0; i < resultMatrix->nRows; i++){
        aVector = a.data[i]; // i-th row from a
        resVector = resultMatrix->data[i]; // i-th row on the result matrix
        for(j = 0; j < a.nColumns; j++){
            factor = aVector[j]; // For each element of the vector
            if(factor != 0x00){
                bVector = b.data[j];
                for(k = 0; k < b.nColumns; k++){
                    // Perform the operation on the result vector
                    resVector[k] = gadd(resVector[k], gmul(factor, bVector[k])
                        );
                }
            }
        }
    }
    return resultMatrix;
}
```

As you can see, this is a rather convoluted way to perform a matrix product. The goal is to avoid cache-misses by optimizing the way the matrix is parsed : we need to avoid jumping from rows to rows as much as possible ; as the rows would be stored as a contiguous array and thus will be stored in the fastest cache.

4.4 Random Linear Network Coding

Once the fundamental bricks has been laid down, a way to actually perform the coding and decoding operations was needed.

These operations, along with all the state and data handling, are implemented in `encoding.c` and `decoding.c`.

4.4.1 Coding

Let's start with the simplest, how to encode data with the network coding scheme ? As explained in the Design/Theory section, it is just a matter of performing a matrix multiplication over data and coefficients. This is done in the following function, from `encoding.c` :

```
/* Using nPackets from data, generate the coefficients and write the encoded
      information in buffer */
void generateEncodedPayload(matrix data, int nPackets, uint32_t seed,
     uint8_t* buffer, int* bufLen){
    srandom(seed); // Initialize the PRNG with seed value
    matrix* coeffs = getRandomMatrix(1, nPackets); // Generate the
        coefficients
    int origDataNRows = data.nRows; // Save the original state from data
    data.nRows = nPackets; // Alter it artificially for the multiplication
        to be ok
    matrix* result = mMul(*coeffs, data); // Perform the multiplication

    // Copy the result to the buffer
    memcpy(buffer, result->data[0], result->nColumns);
    *bufLen = result->nColumns;

    data.nRows = origDataNRows; // Restore data state

    // Free the newly allocated matrices
    mFree(coeffs);
    mFree(result);
}
```

4.4.2 Decoding

The decoding operations are slightly more complicated, as we do not have a complete state to start the decoding with: we only receive packets one at a time, without knowing the total number of packets that we will receive for this block.

The solution is to store a matrix for the received coefficients and a matrix for the data. Thanks to the upper bound on the size of each block and packet size, we can allocate these matrices to be of size (respectively) Block_Size * Block_Size and Block_Size * Packet_Size.

When a new packet is received, we have to handle the two following cases :

- If the packet is a plain text packet of index i
  → Set the i-th column to 1 in the coefficient matrix
  → Copy the data in the data matrix
- The packet is coded :
  → Generate the set of coefficients according to the seed (i.e. The sequence number), copy it into the coefficient matrix
  → Copy the coded data in the data matrix

Then, we need to iterate over the two matrices, to try to reduce the coefficient matrix to the unit Matrix :

$$\begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}$$

This can be achieved by applying standard matrix operations: row multiplications and subtractions. Since these operations are applied on both matrix, as soon as we got only a *one* and all the other values to *zero* in the coefficient matrix, we can conclude that the row in the data matrix effectively contains a decoded version of the payload, which can be transmitted to the application.

4.5 Communication Protocol

As for communicating over the network, the most difficult part has been the specification of the fields, as it appears in the Design section. The process has been incremental, as I often realized that a new field was needed after implementing a first version of the protocol.

I am going to show the code that handles the data packets, as an example of how the specification translates into real code.

Data structure :

```
typedef struct datapacket_t {
    uint16_t blockNo; // Block number of the packet
    uint8_t packetNumber; // Flag (1bit) | Packet index in block (if uncoded
        ), number of packets used for coding (if coded)
    uint32_t seqNo; // Sequence number (always increment)
    uint8_t* payloadAndSize; // uint16 | real payload. Note : the uint16
        gets encoded when the rest of the payload is.

    int size; // Size of the array payloadAndSize. NOT TRANSMISSIBLE !
} datapacket;
```

Marshalling and un-marshalling the data :

```
void dataPacketToBuffer(datapacket p, uint8_t* buffer, int* size){
    uint8_t tmp8;
    uint16_t tmp16;
    uint32_t tmp32;

    tmp16 = htons(p.blockNo);
    memcpy(buffer, &tmp16, 2);
    tmp8 = p.packetNumber;
    memcpy(buffer + 2, &tmp8, 1);
    tmp32 = htonl(p.seqNo);
    memcpy(buffer + 3, &tmp32, 4);

    memcpy(buffer + 7, p.payloadAndSize, p.size);

    (*size) = 7 + p.size;
}

datapacket* bufferToData(uint8_t* buffer, int size){
    uint8_t tmp8;
    uint16_t tmp16;
    uint32_t tmp32;
    datapacket* p = malloc(sizeof(datapacket));

    memcpy(&tmp16, buffer, 2);
    p->blockNo = htons(tmp16);
    memcpy(&tmp8, buffer + 2, 1);
    p->packetNumber = tmp8;
    memcpy(&tmp32, buffer + 3, 4);
    p->seqNo = ntohl(tmp32);

    p->payloadAndSize = malloc((size - 7) * sizeof(uint8_t));
    memcpy(p->payloadAndSize, buffer + 7, size - 7);
    p->size = size - 7;

    return p;
}
```

4.6 Networking Issues

As it might be inferred from the blog, various solutions were offered as to the way to implement the system. Here is a short list of the various possibilities that I explored:

- Intercept outgoing TCP connections
  - Read directly on the packets through a raw socket, as tcpdump does
    →*Slow, lots of processing to do ; abandoned*
  - Redirect matching packet to a TUN/TAP interface
    →*Faster, still many processing as we obtain plain packets*
  - Redirect the connections to a local regular TCP socket that handles it
    →*Access only to the data, simple and fast.* **Adopted**
- Communicate with the server
  - Send hand-crafted TCP packets
    →*Complicated, but good for firewall evasion and avoid restrictive traffic policies. Too slow with Scapy.*
  - Send the messages over UDP datagrams
    →*Simple, need firewall compliance at client and proxy side.* **Adopted**

The main obstacle to use the regular TCP socket to read data from the Client Application at the Client System is that I needed to both redirect the traffic and keep a trace of its original destination. At first I thought it was not possible through regular TCP sockets, hence the first convoluted approaches which implied reading directly from packets.

But fortunately, it is possible to obtain the original destination of a redirected stream via a syscall :

```
getsockopt(newSock, SOL_IP, SO_ORIGINAL_DST, (struct sockaddr *) &
    destinationAccept, (socklen_t *)&destinationLen);
```

The redirection in itself can be done at netfilter level by appending an iptables rule :

```
iptables -t nat -A OUTPUT -p tcp -d <DESTINATION ADDRESS> -j REDIRECT --to-
    ports <LOCAL LISTENING PORT>
```

4.7 Congestion Control

The standard TCP protocol provides two main functionalities :

- Reliable, in-order byte-stream delivery
- Congestion Control

So far we have covered only the first aspect of the problem, but the control of the congestion is of crucial importance to our protocol as well, because a congested network would lead to increase in losses and latency.

The baseline TCP implementation (Reno) uses losses as an indication of network congestion and subsequently back its sending window off. As we are working with packet-erasure networks, this assumption does not hold anymore. The CTCP algorithm uses a variation on the Vegas algorithm: the congestion window is calculated as a function of the RTT delay.

It has been one of the most difficult thing to implement, because the MIT team used 4 different algorithms in their 4 different papers (Sadly true !) and it is not easy to test, as network conditions are neither easy to work with nor reproducible. Thus, I wrote a little test script which could simulate a basic network: source, sink and a router with a infinite queue in between in order to test the various algorithms.

My current iteration of the algorithm is different from the 4 presented, as I decided to use two floating averages for short-term and long-term RTT values. I had to abandon the solution based on the minimum RTT measurement, as it would not work if the network path or conditions had changed. I also dropped the solution based on only one RTT measure, as the jitter proved to be a problem on cellular systems.
Here is the code for this algorithm:

```
/* The function is called upon the reception of an ACK */
if(state->slowStartMode){ // While we are in slow-start mode, increment at
    each event
   state->congestionWindow += 1;
   if(state->congestionWindow > SS_THRESHOLD){
      state->slowStartMode = false;
   }
} else { // Congestion avoidance mode
   delta = 1 - (state->longTermRttAverage / state->shortTermRttAverage);
   if(delta < ALPHA){
      // Increase the window :
      state->congestionWindow += (INCREMENT / state->congestionWindow);
   } else if(delta > BETA) {
        // Decrease the window
        state->congestionWindow -= (INCREMENT / state->congestionWindow);
    }
    // If delta is in between, do not change the window
}
```

The algorithm is quite flexible and it is easy to tweak the parameters in order to obtain the best behavior. After some testing, I settled for the following :

$$\alpha = 0.05; \beta = 0.2; SS\_THRESHOLD = 10; INCREMENT = 3$$

Note: As the code base is quite large (approx. 3,000 lines), I decided not to include the complete code. However, you can browse it from my GitHub account which is linked in the last section.

5 Results

Overall, I would say that the project is successful: I managed to implement a working prototype and learned a LOT in the process. In the following, I will restate my functional requirements and assess their success :

Identifier : 01
Description : The client software must be able to intercept outgoing TCP connections
Criticality : High
Technical Issues : Setting iptables rules correctly, keep states in memory
Dependencies : $\emptyset$

$\Rightarrow$ Successful, iptables rule + socket.

Identifier : 02
Description : The client software must send and receive data with the proxy, *efficiently*
Criticality : High
Technical Issues : Design and implement a communication protocol, Use a loss correcting encoding
Dependencies : 01

$\Rightarrow$ Successful, CTCP protocol

Identifier : 03
Description : The proxy software must relay encoded communications from the client, to normal communications with servers
Criticality : High
Technical Issues : Design and implement a communication protocol, Use a loss correcting encoding, Establish multiple TCP connections, keep states in memory
Dependencies : 02

⇒ Successful, thanks to the client/proxy architecture.

Identifier : 04
Description : The system should be verbose, and allow the user to know what is going on
Criticality : Low
Technical Issues : Programming style : I need to keep it in mind
Dependencies : ∅

⇒ Mixed results, as the displayed information would not be that useful to someone not familiar with the code base.

Identifier : 05
Description : The system should revert hosts to their previous state when disabled
Criticality : Medium
Technical Issues : Correct use of iptables rules
Dependencies : ∅

⇒ Abandoned. An understanding of iptables rules management is expected from the user.

Identifier : PERF01
Description : In a loss less network, the system should not perform worse than 10% slower than plain TCP
Criticality : Medium
Technical Issues : Efficiency of the communication protocol
Dependencies : ∅

⇒ Successful, as long as the computing power needed to encode/decode does not overcome network speed. See the following section.

Identifier : PERF02
Description : In a lossy network, the system should perform measurably better than plain TCP
Criticality : High
Technical Issues : Efficiency of the loss correcting encoding, Efficiency of the communication protocol
Dependencies : PERF01

⇒ Successful, see the following section for more results.

5.1 Test results

5.1.1 Simulation

In order to compare the new solution to standard TCP, I set 3 virtual machines in place : one acting as a client, one router and one proxy/server.

Using the `netem` kernel component, we can add delay and losses on each of the router's output links. The throughput is measured by doing a HTTP GET, via the `wget` tool.
Parameters used :

- Latency: 50+/-5 ms per link =¿ total of 100+/-10
- Losses: loss probability of 0.1 per link with a correlation factor of 15%

As a result, we obtain a TCP throughput of 67.1kBytes/s while CTCP is able to achieve 636kBytes/s, almost ten times faster.

5.1.2 Real-World

Having deployed the proxy onto a quite fast server in South-East Asia, I've been able to run a small-scale test campaign in order to compare the CTCP system to standard TCP on different links. The test procedure was to download a large file over HTTP with wget. The HTTP server was on the same host a the Proxy System.

Link	Standard TCP	CTCP	Comment
WiFi 802.11n (Eduroam) Losses: 0%	1204kBytes/s	1587kBytes/s	*Increase linked to the Vegas-like congestion control. CTCP performs 30% better.*
HSPA 3.5G (3 mobile) Losses: 0.001%	131.7kBytes/s	120kBytes/s	*The Link-Layer must already take care of some corrections for standard TCP ; CTCP performs 9% worse.*
EDGE 2.5G (3 mobile) Losses: 28%	1.48kBytes/s	2.6kBytes/s	*The loss rate is really high, but TCP is still able to operate. Some adjustments must have been done at the Link-Layer. However, CTCP performs 70% better.*

Unfortunately we got a conflict between the Link-Layer mechanisms and this Transport-Layer CTCP, so it's hard to get convincing real-world results from these tests. Further testing should be performed with other types of local link, such as a busy WiFi hotspot.

6 Future Work

While the first results are very promising, a lot of additional work is to be done. This includes:

- **Perform a proper test campaign on a wider range of links**
  - Test on various WiFi hotspots
  - Try different 3G operators (apparently LycaMobile's network behave differently...)
  - Get an operator's cooperation to compare the various schemes (with and without Link-Layer loss-correction for example)
- **Continue to debug and improve the system's performances**
- **Develop and launch a Proxy system that could handle user registration. It could then be used to launch a paid service.**

7 Bibliography and Links

For reference, here is my *GitHub* account, where the project is published as Free Software:
https://github.com/GregoireDelannoy/TCPeP
and here is the DCU blog for this project :
http://student.computing.dcu.ie/blogs/gregoire/

References

[1] **Gprof :** gnu profiler, reports where the program is spending its time, free software. http://http://valgrind.org/.

[2] **Scapy :** python framework for packet forging and dissecting, free software. http://www.secdev.org/projects/scapy/.

[3] **Valgrind :** memory managment checking tool, free software. http://http://valgrind.org/.

[4] Hari Balakrishnan, Venkata N Padmanabhan, Srinivasan Seshan, and Randy H Katz. A comparison of mechanisms for improving tcp performance over wireless links. *Networking, IEEE/ACM Transactions on*, 5(6):756–769, 1997.

[5] MinJi Kim, Jason Cloud, Ali ParandehGheibi, Leonardo Urbina, Kerim Fouli, Douglas Leith, and Muriel Medard. Network coded tcp (ctcp). *arXiv preprint arXiv:1212.2291*, 2012.

[6] Ioannis Koukoutsidis. Tcp over 3g links: Problems and solutions. *arXiv preprint arXiv:0903.4959*, 2009.

[7] Stefan Parkvall, Erik Dahlman, Pal Frenger, Per Beming, and Magnus Persson. The high speed packet data evolution of wcdma. In *Personal, Indoor and Mobile Radio Communications, 2001 12th IEEE International Symposium on*, volume 2, pages G–27. IEEE, 2001.

[8] Kostas Pentikousis, Marko Palola, Marko Jurvansuu, and Pekka Perala. Active goodput measurements from a public 3g/umts network. *Communications Letters, IEEE*, 9(9):802–804, 2005.

[9] Jay Kumar Sundararajan, Devavrat Shah, Muriel Médard, Michael Mitzenmacher, and Joao Barros. Network coding meets tcp. In *INFOCOM 2009, IEEE*, pages 280–288. IEEE, 2009.

[10] Leonardo Andrés Urbina Tovar. *Applying network coding to TCP*. PhD thesis, Massachusetts Institute of Technology, 2012.