WSN - Final report on LT codes implementative project

Stefano Olivotto - Michele Polese

April 20, 2016

1 Introduction

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

2 Implementation of sender and receiver

Sender and receiver are implemented as two standalone classes, which can be launched by a command line interface with some options. They create all the objects needed, then open some sockets and perform transmission and reception of a file using network coding techniques. We implemented two different versions, one which relies on a Random Fountain coding, and the second that uses LT codes, which is much more difficult to implement but offers better performances.
In order to easily handle encoded packets, we defined the class `NCpacket`. It has a single private variable, a `NCpacketContainer` struct that stores an header as a 32 bit integer, a sequence number (`blockID`) as an 8 bit char, and the payload (i.e. the encoded data) as char array of fixed size. It has constructors that accept either these three parameters or the `blockID` and the whole chunk of uncoded data, and performs the encoding inside the constructor. The header represents a seed which is given to a random number generator in order to create the same encoding vector at sender and receiver side.
The RF implementation relies on C++ `rand()` to generate encoding vectors, and `NCpacket` objects are directly created in sender and receiver main methods. The LT implementation, instead, uses a factory paradigm to generate `NCpacket` objects, i.e. it does not directly call the object constructor but creates an helper, `NCpacketHelper`, which is initialized when the main method of sender (and receiver) is called. This allows to generate only once the Robust Soliton Distribution needed to perform coding and the C++ objects of the `random` class, which allow a more robust approach for the encoding vector generation. This class has a method that from the seed generates a vector (of variable size) with the position of ones in the encoding vector (which is much more efficient than handling the whole encoding vector, with few ones and thousands of zeros).
The receiver uses objects from a `TimeCounter` class that performs estimation of time intervals, using the approach inspired to [1]. The time intervals of interest are time between the reception of two packets, in order to perform packet gap detection, and the RTT, in order to estimate whether an ACK sent to the sender was received or not. RFC [1] proposes a method to estimate RTT for TCP connection, based on some filtering of RTT measurements. However, the order of magnitude of the quantities of interest is much smaller than the minimum value that is returned by an estimator working with [1] rules. Therefore some changes have to be made. Let s_{est} be the smoothed estimate of the quantity of interest, s_{var} an estimate of the variance, s a new measurement. Before any measurement is taken, s_{est} is initialized at 50 ms and $s_{var} = s_{est}/2$. Then, once a

new value s is available, these two variables are updated as follows

$$s_{var} = (1 - \beta)s_{var} + \beta|s_{est} - s| \tag{1}$$

$$s_{est} = (1 - \alpha)s_{est} + \alpha s \tag{2}$$

where $\alpha = 1/8$, $\beta = 1/4$ as suggested in [1].

The value returned by the `TimeCounter` object is finally

$$\max\left\{1, s_{est} + K \times s_{var}\right\} \text{ ms} \tag{3}$$

where a granularity of minimum 1 ms is set (instead of 1 s as in [1] and $K = 4$.

Finally, sender and receiver use some static functions which are place in a common `utils` class.

Let's now discuss the implementation of sender and receiver. The retransmission policy is a stop and wait per block, i.e. until a block is not successfully received packets for that block are sent. Flow diagrams for sender and receiver are in Fig. 1 and **??** respectively.

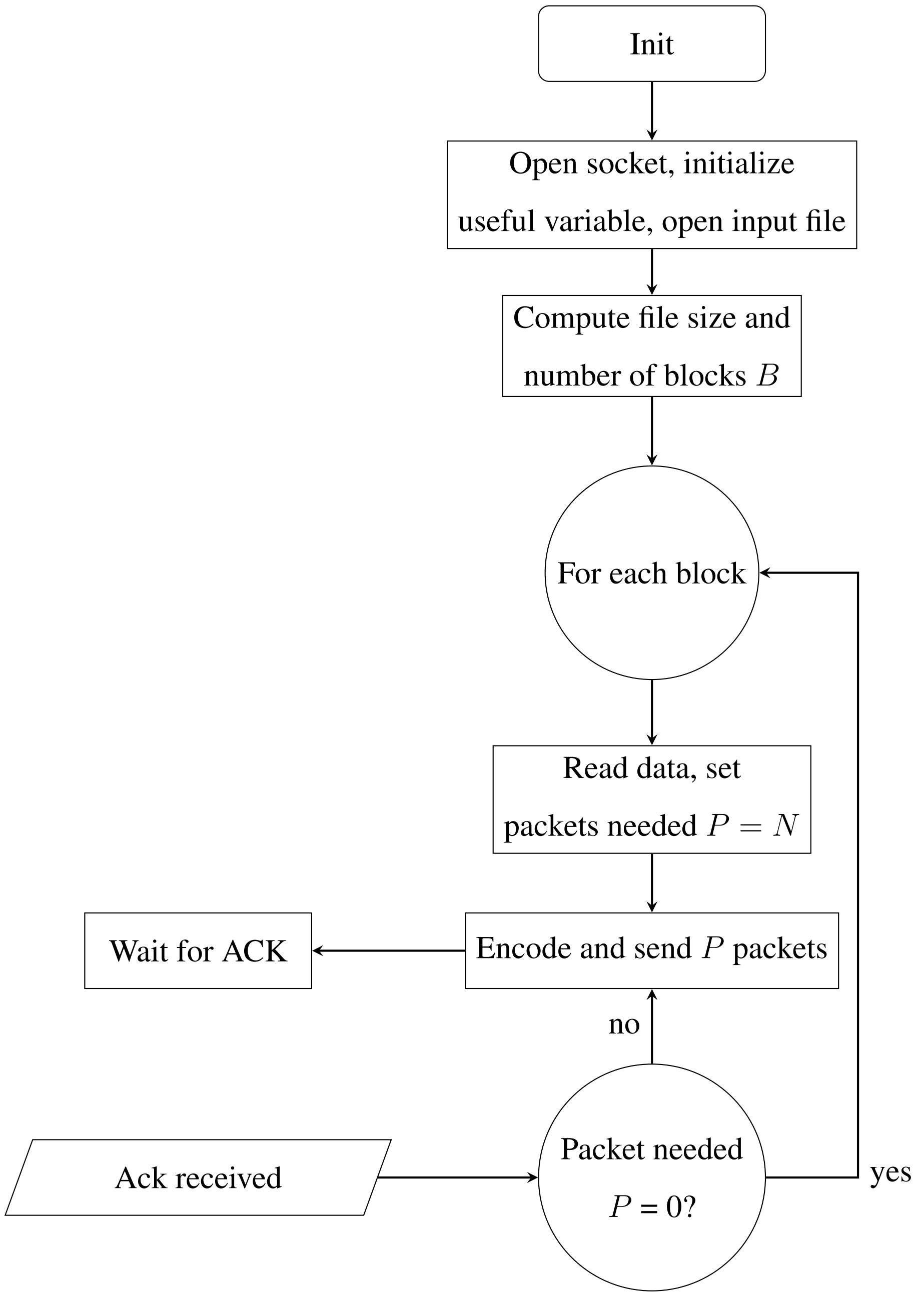
Init
Open socket, initialize useful variable, open input file
Compute file size and number of blocks B
For each block
Read data, set packets needed $P = N$
Wait for ACK
Encode and send P packets
no
Ack received
Packet needed P = 0?
yes

Figure 1: Sender flowchart

3 Implementation of RF and LT

3.1 RF

Once the received has collected a suitable amount of packets, these packets are passed to the decoding_function. This functions arranges the encoding vectors in a matrix and tries to invert it using Gauß-Jordan elimination. In order to get an efficient implementation of this algorithm in $\mathbb{GF}(2)$ we used objects of class mat_GF2 which is implemented in library NTL [2]. We then check if on left part of the matrix we have a complete identity matrix. If it's the case, then we extract from the right part of the matrix the first K rows, which give the inverse of the initial matrix. Using the inverse, we can then apply the inverse XOR transformation, on the dedoced packet payloads, and obtain the original uncoded payloads. In the other case, the function returns the number of missing rows to get K independent ones. The inversion algorithm is then re-applied with the original plus the additional packets (unfortunately, there is no way to exploit the partial results of the failed Gauß-Jordan elimination to reduce the computations for the successive trials: the complexity is the same of starting from scratch).

3.2 LT

For the implementation of LT, we have to represent the graph of message passing. This is done by using two adjacency lists. This choice have been made in order to minimize the computing time of the algorithm, even though it results quite expensive in terms of memory usage. Packets are then resolved according to the message passing algorithm. In this case, if the algorithm fails, there is no easy way to determine how many independent encoding vectors would still be needed in oder to complete the decoding. Because of this and the fact that theoretically, for K sufficiently large, this event would be very unlikely, encouraged us to opt for a simpler system, in which if the decoding fails, then the receiver simply asks for another group of N packets to the sender.

4 Results and conclusions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

References

[1] IETF, RFC 6298, Computing TCP's Retransmission Timer, June 2011

[2] http://www.shoup.net/ntl/