WSN - Final Report on LT Codes Implementative Project

Stefano Olivotto - Michele Polese

May 23, 2016

Abstract

Network coding techniques can be used to efficiently transmit data over wireless links and networks. In this project we will describe an implementation of a sender/receiver pair that by using network coding and ARQ mechanisms transmits a file over different kind of connections. We will show that the choice of different parameters outlines trade-offs in terms of different performance metrics, and that the system reaches high efficiency and goodput when used over a real connection from Padova to Lausanne.

1 Introduction

Network coding techniques emerged at the beginning of the century as an efficient way to transmit data over links with high packet error rate, and to avoid having different retransmissions for different multicast destinations. The basic principle is that the transmitted packets are encoded version of the original ones, which are combined together according to different rules, and they also piggyback an encoding vector that specifies which packets were combined. The receiver is able to retrieve the original information by knowing the encoding vectors. If the transmission medium is not reliable and introduces packet losses, then new encoded packets created on the fly can be retransmitted, until the receiver is capable of decoding. This is why these codes are defined as rateless. The pioneer of network coding techniques is Michael Luby, which introduced as first the idea of Random Fountain (RF) [2] and then the Luby Transform (LT) codes [3].
In this project we implemented a system composed of a sender and a receiver[1], which use either RF or LT network coding techniques, together with ARQ, in order to transmit a file from source to destination, adapting to different kind of transmission media (Wireless Local Area Network - WLAN, internet, localhost). The remainder of this report is organized as follows. Firstly, in Sec. 2 we will describe the implementation and

[1]The source code is available at `https://github.com/mychele/wsn1516-finalproject`.

the technical issues that emerged during our work. Secondly, Sec. 3 will contain data and comments on a thorough experimental campaign we conducted in order to assess the performance of our implementation. Finally, some conclusions will be drawn in Sec. 4.

2 Technical Approach

The objective of the project is to implement a network coding system that works and guarantees reliable transmission of files over different communication systems. With such a software, we will be able to assess the performance of network coding schemes in various setups, thus studying the trade-offs between different parameters (redundancy, retransmission timeouts) and metrics (goodput, efficiency).

The system is composed of two main C++ scripts, sender and receiver, and of some utility classes which handle coding/decoding, time interval estimation, packet abstraction. We implemented two different versions of the network coding system, one which relies on Random Fountain (RF), and another that uses Luby Transform coding (LT). We will show that the latter offers a better performance at a price of a more complex implementation. Let's describe as first the utilities, then how sender and receiver work (i.e. the retransmission mechanism) and finally the details on the implementation of the network coding algorithms.

Utilities – The class `NCpacket` is an abstraction of an encoded packet, with a private variable (a `NCpacketContainer` struct) that stores an header of 32 bit, a sequence number (`blockID`) of 8 bit, and the payload (i.e. the encoded data) as an array of fixed size. The header represents a seed which is given to a pseudo random number generator (pRNG) in order to create the same encoding vector at sender and receiver side.

The RF implementation directly creates encoded `NCpacket` objects 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 (RSD) needed to perform coding and the C++ objects of the `random` class. `NCpacketHelper` objects have 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 `TimeCounter` class is used to perform estimation of time intervals, using the approach inspired to RFC 6298 [4]. The time intervals of interest are the ones between the reception of two packets (in order to

perform packet gap detection) and the RTT (to estimate whether an ACK was received by the sender). RFC 6298 proposes a method to estimate RTT for TCP connection, based on some filtering of raw 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 the rules described in [4]. 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 [4].

The value returned by the `TimeCounter` object is finally

$$e = \max\{M, s_{est} + K \times s_{var}\} \text{ ms} \tag{3}$$

where a granularity of minimum M ms is set (in [4] $M = 1000$ ms, we used $M = 1$ ms or $M = 100$ ms) and $K = 4$.

Sender – Flow diagram for the sender is in Fig. 1a. The retransmission policy is a stop and wait (S&W) per block, i.e. until a block is not successfully received new encoded packets for that block are sent. Both for the RF and LT implementations a rate K/N is set to guarantee a certain probability of successful decoding.

Given a certain K, the sender reads a portion of the file and creates N encoded packets (as `NCpacket` objects). Then it sends them as UDP datagrams and starts waiting for an acknowledgment from the receiver. When an ACK is received, the sender checks for the number P of packets still needed. If this is 0, it continues with the next block to be sent, otherwise it sends P new encoded packets. In the payload of the first transmitted packet the sender reserves 4 bytes to communicate to the receiver the file size, so that this terminates correctly once the whole file is received.

Receiver – The receiver flow diagram is in Fig. 1b. For each block of data, the receiver waits for the reception of N packets, then it attempts to decode them. If successful, it appends to the output file the decoded chunk of data and starts waiting for the next block. If not, it behaves differently in LT and RF implementations. In the latter, when running Gauß elimination, it is possible to know the number P of linearly dependent encoding vectors. Therefore the receiver sends an ACK to the sender carrying P. When new packets are received, they are all used for decoding, until K linearly independent packets are found. With LT, instead, this is not

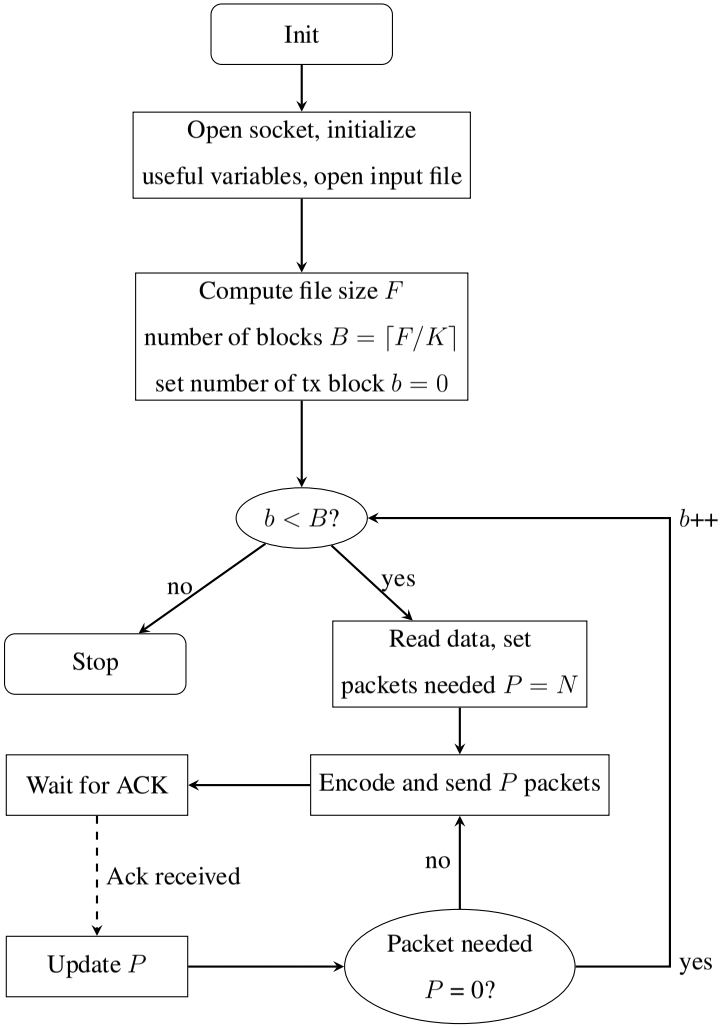
trivial and therefore we chose to query an entire new block by sending an ACK with $P = N$ to the sender. A possible improvement is to retransmit only a fraction of the N packets and combine them with a random subset of the ones already received. However, if the ratio K/N is low enough, the probability of unsuccessful decoding can be made as small as desired.

The receiver uses different timers for different events. The first (called `pg` in Fig. 1b) measures the time between the reception of two packets, in order to perform packet gap detection. If a packet is expected and the timer expires, the receiver assumes it was lost. Since the receiver has to collect N packets before decoding, if the number of received packets R is lower than N and the timer expires, an ACK is transmitted to the sender specifying the number of needed packets $P = N - R$. The second timer (`ACK`) is used when waiting for the first packet after an ACK. It thus waits for the ACK to be received, new packets to be encoded and for the first one to be delivered at the receiver (this interval is defined as a *round trip time* (RTT)). If it expires, the ACK is sent again. Both these timers use estimates provided by `TimeCounter` objects. For the second timer, however, a scaling factor is applied to the estimate e to account for the fact that at each (re)transmission a different number of packets P must be encoded, or a new chunk is read from file, thus different delays may be experienced. We observed that when the plain estimate e was used the system was very aggressive and unneeded retransmissions were triggered. Therefore we applied the scaling factor of $100 \cdot P/N$ when $P < N$ or 10 for $P = N$ which we observed offering a good tolerance against undesired retransmission while not degrading the goodput and throughput.

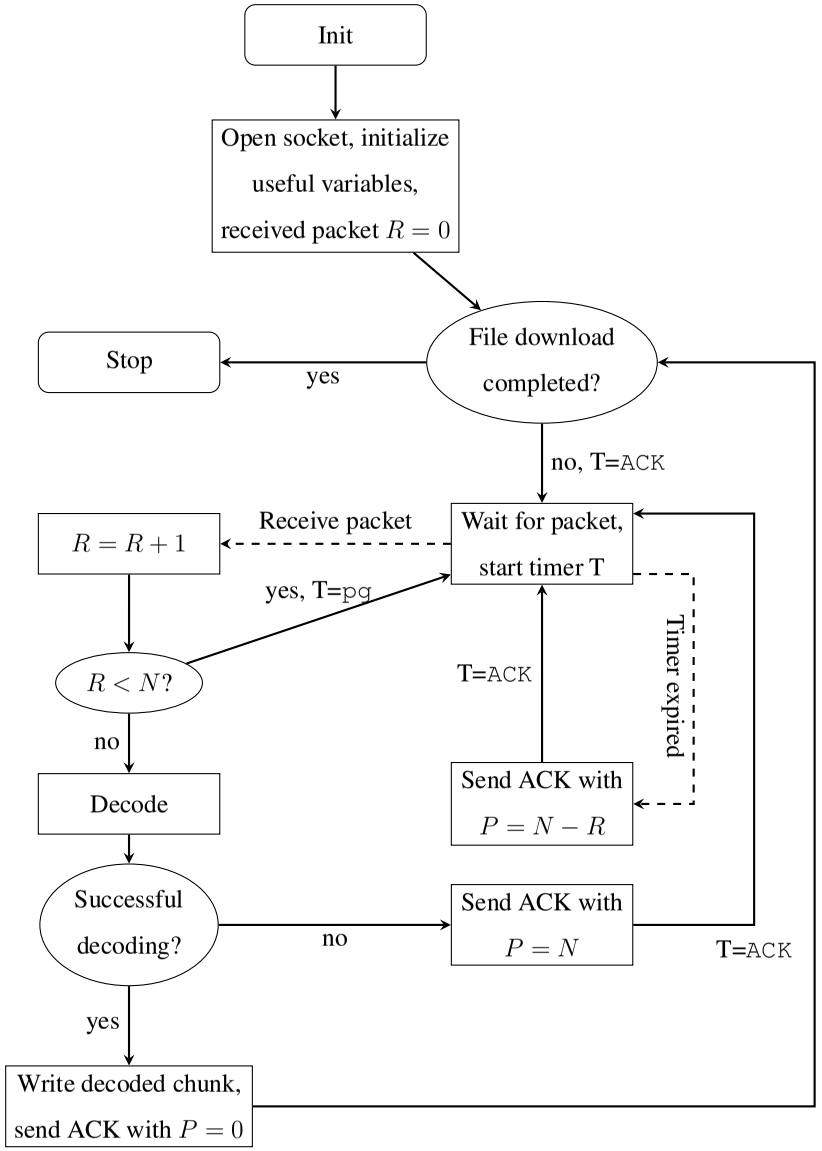
PER estimation at sender – The sender has also a `PER_MODE`. If this is enabled it estimates the *Packet Error Rate* (PER) of the channel and tries to adapt to it. The estimation is done with a moving average on a window of past block transmissions. In particular the sender records the average number of packets P_b needed to successfully transmit each block, and estimates the PER as $\widehat{PER} = 1 - N/P_b$. Then, once the sender needs to send P packets, it sends $P/(1 - \widehat{PER})$ packets trying to prevent channel losses without needing retransmissions. This lowers the efficiency of the system, but decreases the time needed to send the file. The estimation works better with the RF setup, since it transmits many more blocks than the LT one, of smaller size, thus it reacts more quickly to changes in the PER.

2.1 Network Coding implementation

Random Fountain – Once the receiver has collected a suitable amount of packets, these are passed to the `decoding_function`. This function 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



(a) Sender flowchart



(b) Receiver flowchart. T can be a timer of two different kinds: `pg` for packet gap detection and `ACK` for RTT (network RTT + encoding of new blocks)

Figure 1: Sender and receiver flowcharts. The dashed arrows represent possible flows triggered by specific events.

objects of class `mat_GF2` which is implemented in library `NTL` [5]. We then purge the all 0s rows, and check if on left part of the matrix we have a complete identity matrix of order K. If that is the case, then we extract the right part of the matrix (i.e. we erase the identity matrix), which gives the inverse of the initial matrix. Using the inverse matrix, we can then apply the inverse XOR transformation (decoding operation) on the encoded packet payloads, and obtain the original uncoded information. In the other case, the function returns the number of missing rows to get K independent rows. The inversion algorithm is then re-applied with the original plus the additional encoding vectors (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 as starting from scratch).

Luby Transform – For the implementation of LT, we have to represent the graph of message passing. This is done by using two adjacency lists. This choice was 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 order to complete the decoding. This consideration 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. The parameters of the Robust Soliton Distribution have been set to $c = 0.03$ and $\delta = 0.5$, as suggested in [6].

2.2 Complications found

In the implementation of LT decoder we faced a trade-off between memory consumption and computational complexity. We chose to optimize for the latter, this however constrained the possible N and K values to $K \leq 5500$. Moreover, our choice of guaranteeing a low probability of decoding failure, prevented us from exploring the same intervals of relative redundancy for all values of K. Another issue was caused by the implementation of the `rand` function in C++, which is different in different versions of the Standard Library. Therefore the same seed generates different pseudo random sequences when sender and receiver use two different operative systems. We adopted the more modern C++11 `random` header which contains a Mersenne Twister which is platform independent, however the same problem arises with `random distribution` classes.

3 Results

In our experiments to evaluate the performance of the system using LT we have considered the transmission of a file of size 46.2 MB. Packet's header consists in 32 bits of pRNG seed plus 8 bits of `block_ID`, for a total header size of 40 bits. The ACK contains 32 bits to represent the number of packets the receiver still needs to get N correct packets (therefore we allow for a value of N up to $2^{32} \simeq 4 \cdot 10^9$, which is more than enough for our purposes) plus 8 bits of `block_ID`, again for a total of 40 bits. The various experimental scenarios have been summarized in Table 1.

scenario	payload (byte)	minimum timeout before RTX (ms)
localhost aggressive	2048	1
localhost efficient	2048	100
Wi-Fi	1467	100
Network	1467	100

Table 1: Main experimental setups.

The payload size of Wi-Fi and network is dictated by the MTU size, in order to avoid IP packet fragmentation which degrades the performance. We remark that with these settings we are able to transmit a file of size up to 2.25 PB in localhost and 1.61 PB over Wi-Fi or network (if we have no redundancy and we adjust the number of bytes used to communicate the file size).

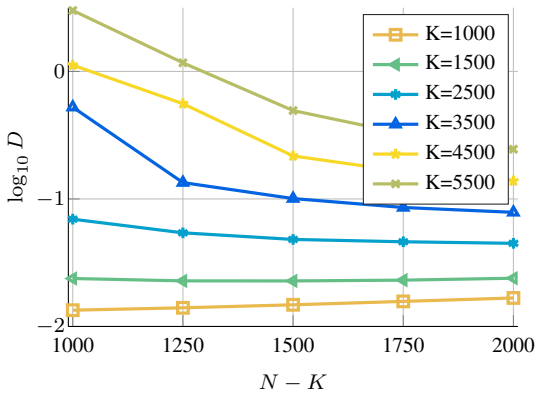
The Wi-Fi experiment was carried out using two different setups. In the *high SNR* scenario sender and router, router and receiver are at a distance $d = 3$ m, with one brick wall between sender and router. The *low SNR* scenario deployed sender and router in the same positions, but the receiver was moved 2 walls and 15 m away from the router. The Wi-Fi router was a 802.11a/b/g device. The transmission over the network has been carried out from Padova to Lausanne. The sender was connected via Wi-Fi to a connection with 10 Mbit/s in uplink, while the receiver was connected via Ethernet to a 100 Mbit/s symmetric connection (therefore the bottleneck is the 10 Mbit/s uplink of the sender). For localhost and network experiments, the receiver was executed on a workstation with a 3.4 GHz i7 processor and 8 GB of RAM, while in both Wi-Fi setups the laptop on which the receiver run had a 2.2 GHz i7 processor with 8 GB of RAM.

In all these regimes we consider a redundancy for which the probability of decoding failures (i.e. the receiver correctly receives N packets, but they are not sufficient to decode the K information packets) is very low (typically $< 10^{-3}$), and therefore negligible.

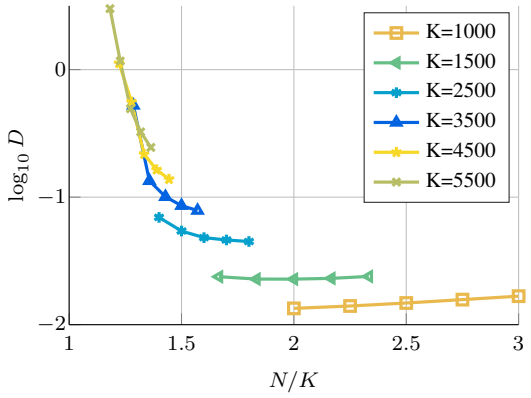
The first very important trade-off is between decoding time and transmission time as a function of the redundancy, which translates in most of the cases in a trade-off between goodput and efficiency. In fact, for a fixed K and PER, adding redundancy decreases the time to decode (for our cases of interest), but it also increases the time to transmit the file (because there are more packets to transmit). In general we can write that $delay = transmission\_time + decoding\_time$ where *transmission_time* includes all time spent in propagation, waiting, retransmitting, etc. over the channel, and *decoding_time* is the time that it takes to the decoder to decode a block of N packets once they have all been correctly received. We also remark that transmission time slightly decreases when K increases. In fact packets of the same block are transmitted using a system similar to selective repeat (the receiver tells the sender how many packets it needs, and the sender transmits them), while blocks are sent in a S&W fashion (i.e. a block is transmitted only after the previous one has been successfully decoded). As a consequence we have that delay slightly increases with the number of blocks. We can sum up these considerations in Table 2, where we show the dependencies of *transmission_time* and *decoding_time* on one parameter by keeping all others fixed.

	K	$N-K$	$\frac{N}{K}$	PER
transmission_time	$\searrow$		$\nearrow$	$\nearrow$
decoding_time	$\nearrow$	$\sim\searrow$		independent

Table 2: Monotonicities.



(a) Average time to decode a packet vs absolute redundancy



(b) Average time to decode a packet vs relative redundancy

Figure 2: Average time to decode a packet, for different K. Notice that y axis is in logarithmic scale.

In Fig. 2 we show the decoding time (localhost scenario) for a set of values of K and redundancy. Clearly, in general this time decreases as we increase redundancy, since there are more connections in the graph of message passing, and therefore at each iteration we are able to resolve more nodes. Hence less iterations are

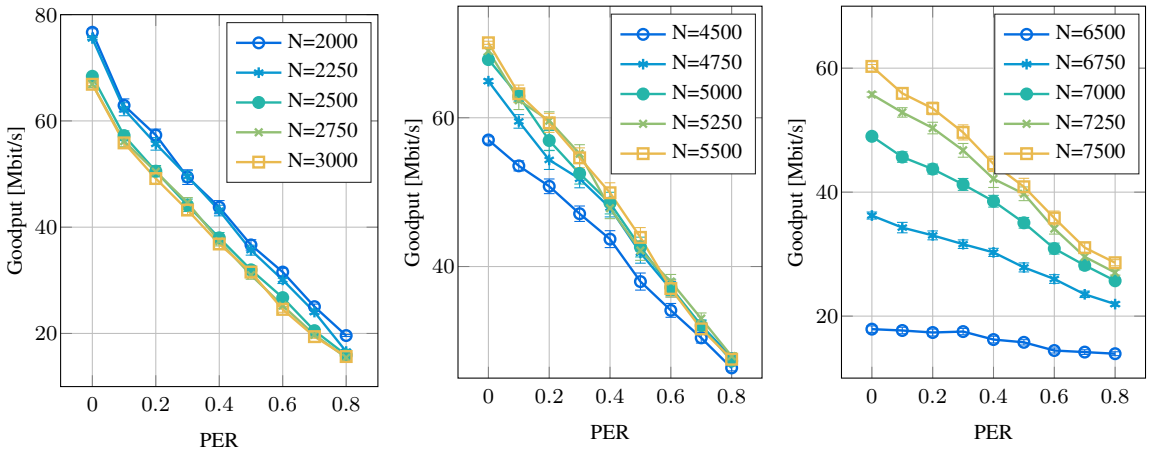
needed to decode the packets. However, as we can see from Fig.2b, with the redundancy above a certain value the time to read and write the graph for message passing (which increases with N) becomes dominant over the time (number of iterations) needed for message passing itself, and therefore the decoding time increases. Nevertheless, this increase is much less significant than the decrease at low relative redundancies, and for practical purposes we can assume that decoding time from a certain value on stays constant. Roughly speaking, we can see that around a value of relative redundancy $\frac{N}{K} = 1.5$ there is a change in regime: the derivative of the decoding time according to redundancy changes dramatically. In particular as we approach 1 the increase in decoding time is exponential. We also remark (in part by guessing the behavior of the lines if we were able to explore the whole interval of redundancies for all values of K) that, if we fix a given relative redundancy, the decoding time increases with K.

This trade-off between transmission time and decoding time is very well shown in Figures 3 and 4, which show the performance obtained in localhost with an aggressive ARQ (i.e., minimum timeout before retransmission of 1 ms). First of all we remark that the goodput (i.e., information bit per second) decreases approximatively linearly with the packet error probability. Moreover, for K small, adding redundancy decreases the goodput. In fact for K small and redundancy starting at $N - K = 1000$ (i.e., 100%), transmission time prevails over decoding time, which, at these levels of redundancy, increases slowly as a function of N. Therefore, since the transmission time increases with redundancy (we can assume that transmission time increases approximately linearly in N), the overall delay increases. Hence, for K small, the goodput decreases by adding redundancy. On the other hand, if we increase K, we reduce the relative redundancy. From 2b we see that for $K = 3500$ or $K = 5500$, decoding time dramatically depends on redundancy (and also increases with K), and therefore when redundancy is small the time to decode the file has more impact than the time to transmit it. Hence, in these cases the goodput increases by adding redundancy. As a consequence, if efficiency is not to be optimized, it is better, for K big enough, to choose high redundancy in order to decode faster, rather than reduce N to transmit faster.

We also observe that goodput decreases with K. In fact, in general, in localhost, since transmission time is small, for a fixed absolute redundancy, increasing K allows to marginally improve the time to transmit (fewer packets to transmit because the relative redundancy is lower and fewer blocks to send), while it significantly increases the time to decode (because the relative redundancy decreases).

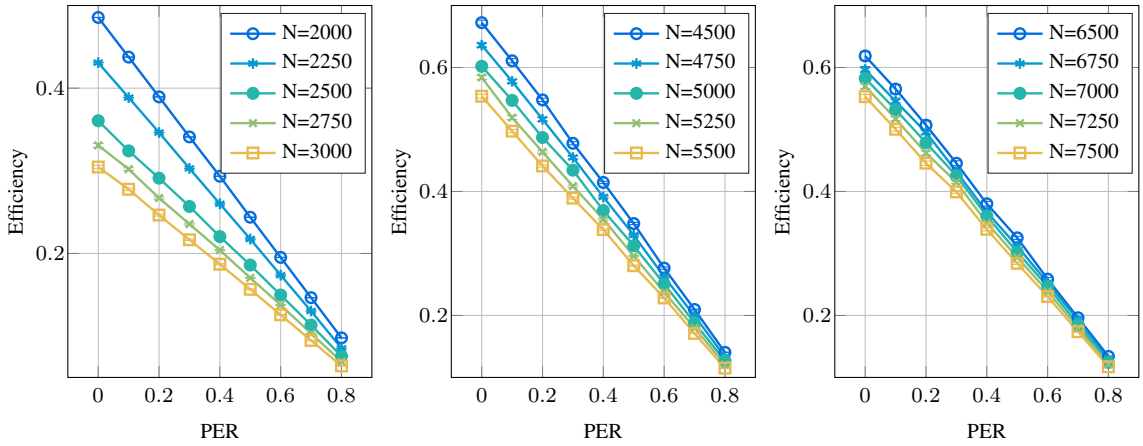
However the efficiency, i.e. the ratio between information and sent packets, always decreases with redundancy (since we are not taking into account the delay we previously defined).

We also remark that the difference in goodput for a given K and two values of N decreases as PER increases.



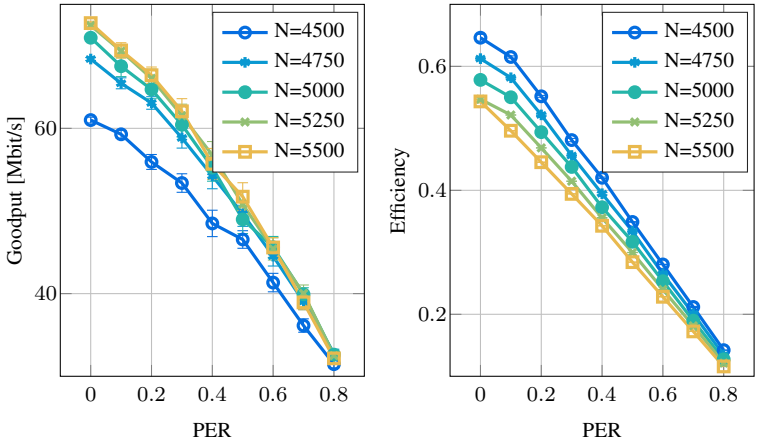
(a) $K = 1000$ (b) $K = 3500$ (c) $K = 5500$

Figure 3: Goodput for $K \in \{1000, 3500, 5500\}$ in localhost, aggressive setup.



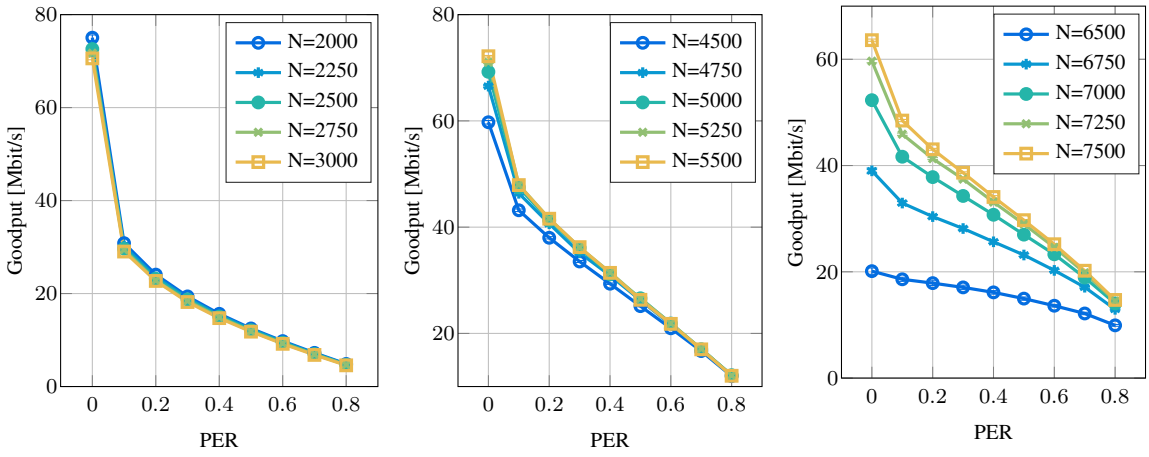
(a) $K = 1000$ (b) $K = 3500$ (c) $K = 5500$

Figure 4: Efficiency for $K \in \{1000, 3500, 5500\}$ in localhost, aggressive setup.



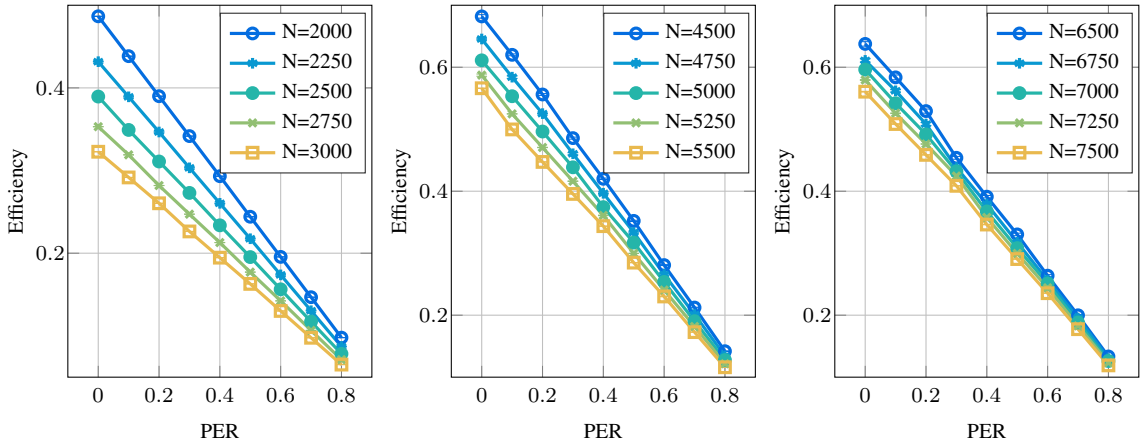
(a) Goodput (b) Efficiency

Figure 5: Goodput and efficiency for $K = 3500$, using PER estimation, aggressive setup.



(a) $K = 1000$ (b) $K = 3500$ (c) $K = 5500$

Figure 6: Goodput for $K \in \{1000, 3500, 5500\}$ in localhost, efficiency-oriented setup.



(a) $K = 1000$ (b) $K = 3500$ (c) $K = 5500$

Figure 7: Efficiency for $K \in \{1000, 3500, 5500\}$ in localhost, efficiency-oriented setup.

In fact we can write, for $N > N'$ and $PER > PER'$

$$[delay(N, PER') - delay(N', PER')] - [delay(N, PER) - delay(N', PER)] =$$
$$= [tx\_time(N, PER') - tx\_time(N, PER)] - [tx\_time(N', PER') - tx\_time(N', PER)]$$

because the decoding time does not depend on PER. Let's assume that the transmission time in localhost is a linear function of the number of retransmissions (we call α the constant of linearity). This is justified by the fact that in localhost, given the high rate, the time to transmit a set of packets weakly depends on the number of packets (within reasonable limits). If, moreover, we model the number of packets left to transmit as a geometric sequence in the number of retransmission r, then we can write $N \cdot PER^r = c \Rightarrow r = -\frac{\ln(N)-\ln(c)}{\ln(PER)}$, where c is a small constant at which we consider that all packets have been received (e.g. $c = 0.1$). Therefore

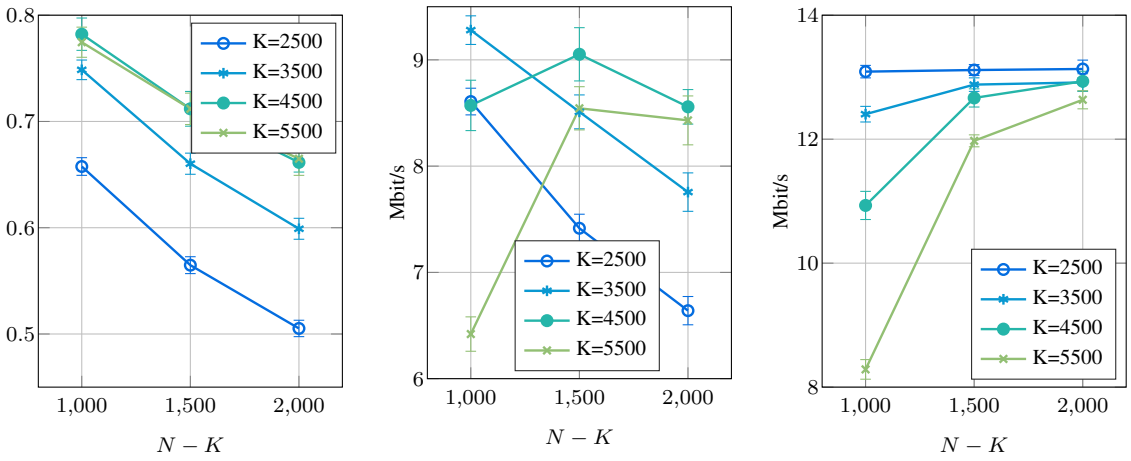
$$[delay(N, PER') - delay(N', PER')] - [delay(N, PER) - delay(N', PER)] =$$
$$= \alpha \frac{\ln(N)}{\ln PER \ln PER'} \ln \frac{PER'}{PER} - \alpha \frac{\ln(N')}{\ln PER \ln PER'} \ln \frac{PER'}{PER} = \frac{\alpha}{\ln PER \ln PER'} \ln \frac{PER'}{PER} \ln \frac{N}{N'} < 0$$

since $N > N'$ and $1 > PER > PER' > 0$. Therefore the gap in delay increases as PER increases, and since the goodput depends on the inverse of the delay, the gap in goodput decreases as PER increases.

In Fig. 5 we show the results we obtain by enabling the PER estimation mode for $K = 3500$. In this setup, the sender tries to estimate the PER, in order to improve the goodput. We see that effectively we have quite a significant increase in goodput, while the efficiency remains approximately the same (except for the case $PER = 0$, which gives some estimation troubles, probably due to convergence problems of estimators). Therefore we have that our PER estimation method proves to be quite useful, even though performance doesn't change dramatically. This also is due to the fact that the number of blocks (i.e., 7 for $K = 3500$) used in an LT setup transmission is quite small and therefore estimation is not very accurate.

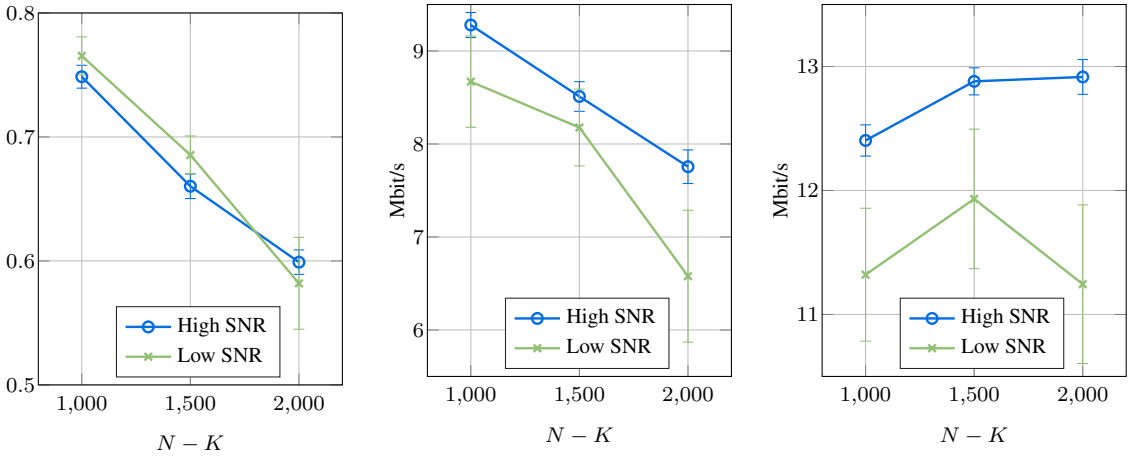
In Figures 6 and 7 we show results for the efficiency-oriented setup, where the minimum timeout for packet gap detection is increased from 1 ms to 100 ms. As expected, goodput is lower with respect to the aggressive scenario, since we are increasing the delay (with the exception of the case PER=0, for which, in fact, no packets are lost, and therefore there are no retransmissions and this timeout has little impact on the performance). We also see that efficiency only improves marginally. An hypothesis for that is that in localhost transmission time for a packet is less than 1 ms, therefore by increasing the minimum timeout we are not significantly decreasing the number of packets we retransmit, and therefore efficiency remains more or less the same.

Again, in Fig. 8 we see the trade-off between goodput and efficiency in the Wi-Fi scenario (*high SNR*). Efficiency is always monotonic with respect to redundancy, whereas the behavior of the goodput is more complicated, and depends on which of the transmission time and the decoding time prevails. In particular



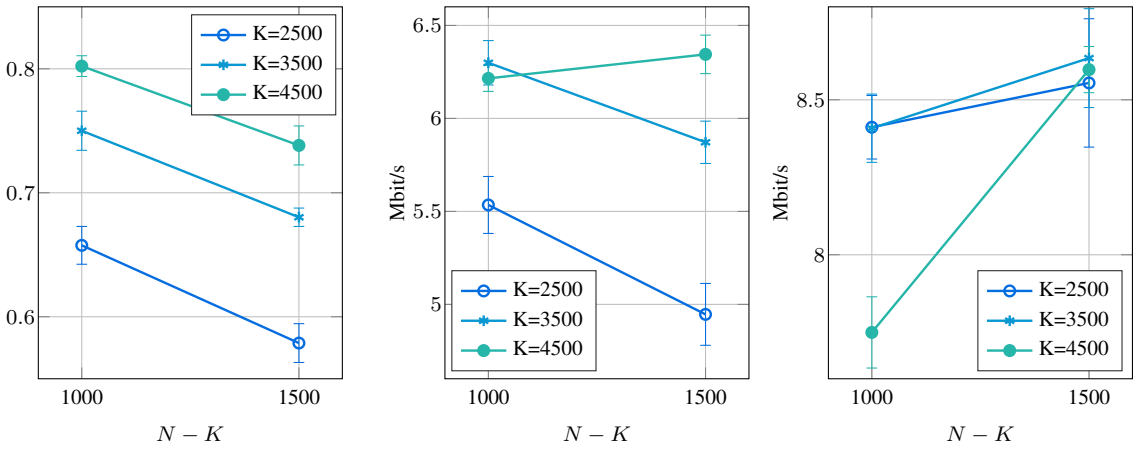
(a) Efficiency (b) Goodput (c) Throughput

Figure 8: Metrics for Wi-Fi simulation (*high SNR*).



(a) Efficiency (b) Goodput (c) Throughput

Figure 9: Comparison between setup with *high SNR* and setup with *low SNR*, for $K = 3500$.



(a) Efficiency (b) Goodput (c) Throughput

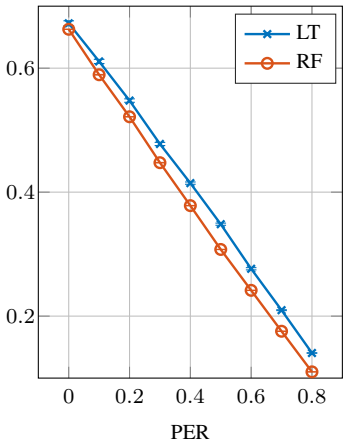
Figure 10: Metrics for Network simulation.

we see that goodput increases dramatically for high K when we increase redundancy from 1000 to 1500. This, again, is due to the fact that too little redundancy (in ratio with respect to K) yields longer decoding times, which can dominate over the transmission time, and therefore having little redundancy degrades the performance. This explains the peak at $N - K = 1500$ for high K: at lower redundancy performance is dictated by decoding time (which increases as redundancy decreases), whereas for higher redundancy performance depends mainly on transmission time (which increases as redundancy increases, since we have more packets to transmit, and has a larger impact on non localhost scenarios). Therefore goodput depends both on an increasing and decreasing function with respect to redundancy, which explains the non-monotonicity. We also remark that clearly throughput increases with redundancy, since we reduce the time to decode, whereas the time to transmit the single packets doesn't change very much (if the transmission rate of the packet is high enough, which is the case for Wi-Fi 802.11g). This is confirmed by the fact that the increase in throughput is greater for high K, because for these values the time to decode depends exponentially on redundancy (see Fig. 2b).

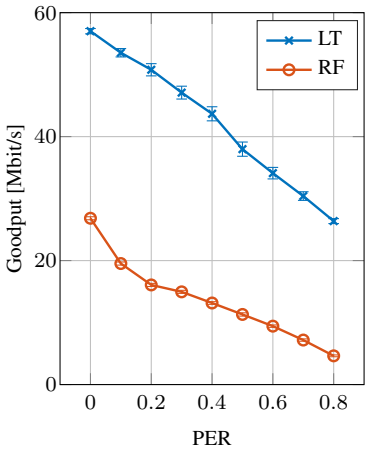
In Fig. 9 we compare the experimental results obtained using Wi-Fi in a *high SNR* scenario and *low SNR* for $K = 3500$ and different $N - K$. Even though experimental measurements show quite a large variance (as can be noted by observing the confidence interval), it appears quite clearly that, as expected, in the high SNR scenario we obtain a better performance in goodput with respect to the low SNR case. Notice also that the gap between the throughput mean values is greater than the gap between goodput mean values. This shows that network coding increases robustness against the level of noise in the channel. However measurement uncertainties do not allow to draw further and more precise conclusions (even though we can assume that the qualitative behavior does not differ very much from the other scenarios, and the same considerations apply).

Fig. 10 shows the results of packet transmission between Padova and Lausanne. Also here we see the same qualitative behavior of the other scenarios. In particular we remark the crossing of the goodput curves for $K = 3500$ and $K = 4500$ is due to the fact that when redundancy is too small, the time to decode a block increases significantly and impacts on the performance (even though less with respect to the Wi-Fi scenario, since in this setup the transmission time is higher, as we have to transmit a packet through a network of approximatively 12 hops, and the decoding is run on a faster workstation).

Fig. 11 shows a choice of N and K which yields approximately the same efficiency for RF and LT, but allows for a much higher goodput with LT, proving therefore the usefulness of LT. In fact, using RF, we are constrained to low values of N and K (respectively 17 and 12) because of the complexity of the Gauß elimination. Low values of K oblige to a greater relative redundancy in order to have low possibility of failing



(a) Efficiency



(b) Goodput

LT [ms]	RF [ms]
0.15	0.61

(c) Decoding time per packet

Figure 11: Comparison between LT ($K = 3500$, $N = 4500$) and RF ($K = 12$, $N = 17$).

the decode, and moreover result in more blocks (of K packets). Furthermore, as it can be seen in Fig. **??** we have that decoding time for RF is bigger than for LT, even if K and N are much smaller, confirming the fact that the LT decoding algorithm is much faster than Gauß elimination. All this implies a bigger delay. Hence, the goodput decreases.

4 Conclusions

In this project we implemented a system to transmit and receive a file over a noisy channel using network coding. Results show good performance, sufficient to allow a real utilization of the system (although clearly much more performing solutions exist).

We have compared two different algorithms for network coding: RF and LT, confirming that the crucial drawback of RF is its decoding time, which becomes unacceptable for large values of N and K. We have also shown different trade-offs for LT, the most important being the one between efficiency and goodput, and how they depend on different parameters, mainly K and the redundancy, but also timeouts.

In this project we have learned how to design and code a working system. This requires, other than the main algorithms, also taking care of many minor details, which however may make the system fail if not accurately designed (e.g., retransmission protocols, timeouts). In particular it also requires performance-oriented coding, which is not something trivial. For example, a good amount of effort was devolved into trying different solutions (e.g., vectors VS array, bitset VS array of booleans VS `mat_GF2`) to find the most performing ones. We have also learned to automatically collect and arrange data in significant ways in order to meaningfully evaluate the performance of the system and highlight trade-offs.

References

[1] R. Ahlswede, Ning Cai, S. Y. R. Li and R. W. Yeung, Network information flow, in IEEE Transactions on Information Theory, vol. 46, no. 4, pp. 1204-1216, July 2000

[2] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, A digital fountain approach to reliable distribution of bulk data, in Proceedings of the ACM SIGCOMM 98 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM 98), 1998

[3] M. Luby, LT Codes, in Proceedings of the 43rd Annual Symposium On Foundations of Computer Science, 16-19 November 2002

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

[5] V. Shoup, NTL: A library for doing number theory, 2016 `http://www.shoup.net/ntl/`, Version 9.6.2

[6] D. J. C. MacKay, Fountain codes, in IEEE Proceedings - Communications, vol. 152, no. 6, pp. 1062-1068, December 2005