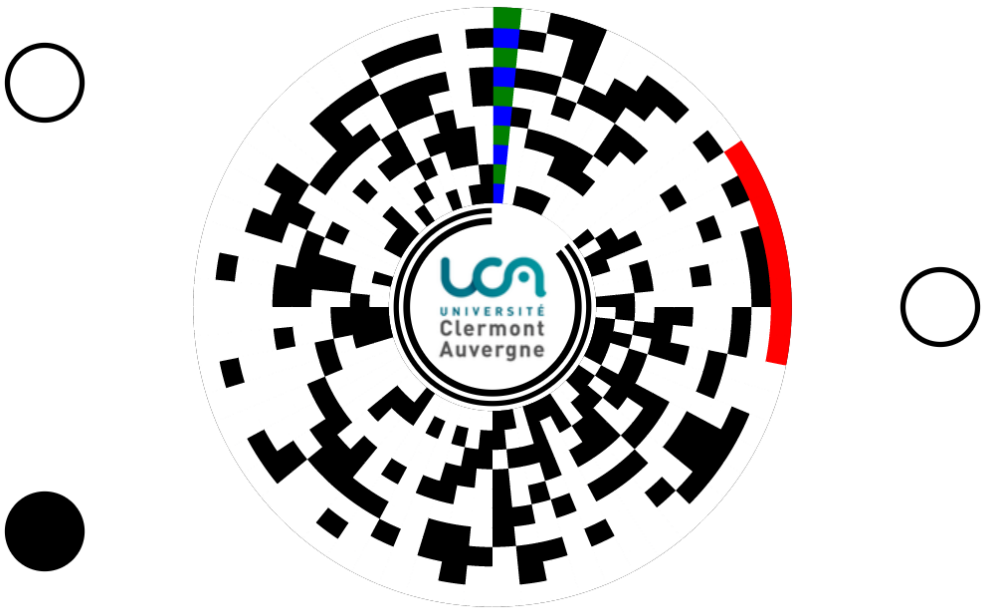
Alexis

PLESSIER

PROJET RESEAUX :

Protocole Graphique



L3 Informatique

18 Avril 2021

Table des matières

I. Introduction

Dès l'annonce du sujet de Travaux Pratiques Réseaux j'ai essayé de trouver un moyen original de pouvoir transmettre une information, mais être original dans le domaine de l'informatique n'est pas chose aisée. La première idée qu'il m'est venue était de transmettre l'information autour d'un cercle type cible. Une cible est un cercle composé de 8 divisions égales, ce qui correspondait bien avec la composition d'un octet, je trouvais donc le lien assez intéressant. J'ai donc commencé à coder cette idée et j'ai rapidement rencontré un obstacle : La taille du protocole pour le nombre d'information était assez ridicule. J'aurais pu continuer dans cette voie et définir ce protocole pour un système d'inventaire par exemple (ex : ISBN) ou bien un système de directives simples (ex : Centre automatisé de gestion des commandes Amazon). J'aurais également pu coder plus d'informations en utilisant des couleurs, mais je trouve cette méthode trop contraignante pour sa lecture, les aléas sont trop nombreux, c'est pourquoi j'essaierai de me passer le plus possible de couleurs. Je suis donc venu au fait de reprendre cette idée de départ et de la diviser à nouveau par 8 pour augmenter sa capacité par 8.

II. Programmation

1. Lancement du programme

Pour la programmation, j'ai décidé de faire ce projet en **Python (version 3)** car c'est l'un des langages que je maitrise le plus, mais aussi car c'est un langage très polyvalent me permettant beaucoup de libertés et possédant une communauté très active. J'étais sûr de ne pas rester bloquer au milieu de mon projet en faisant ce choix.

Pour mener à bien ce projet j'ai donc utilisé plusieurs bibliothèques, nécessaires notamment pour la partie graphique. Il est donc impératif de les posséder sur son ordinateur avant d'exécuter le programme :

- svgwrite (https://pypi.org/project/svgwrite/)
- numpy (https://numpy.org/install/)
- pillow (https://pillow.readthedocs.io/en/stable/installation.html)
- resizeimage(https://pypi.org/project/python-resize-image/)

- math et time, inclus dans **Python3**

La plupart des installations ont était faite avec **pip3** pour ma part.

Le lancement du programme se fait simplement avec la commande :

$ python3 ProGraph.py

Il suffit ensuite de suivre le menu affiché et le programme effectuera la demande.

2. Construction de la matrice

Pour la construction de la matrice, j'ai choisi d'utiliser le format Longueur-Donnée-Redondance-Fin dans un premier temps. L''utilisateur entre alors la chaîne de caractère qu'il veut coder et à ce moment-là je peux calculer la longueur du message, cette information sera la première de ma matrice. J'ai établi la limite maximum de cette longueur à 255. Je traite ensuite le message et transforme chaque caractère par son code ASCII. Grâce à la matrice que j'ai actuellement, je peux calculer et ajouter ma redondance d'information (détaillé dans le paragraphe suivant) et ajouté une donnée de fin arbitrairement choisie (255 ici).

Une fois ma matrice d'entiers obtenue je peux transformer ces entiers en binaire. Attention, les valeurs des données de base n'excédant pas 128 peuvent se coder sur 8 bits mais ma redondance, elle, se code sur 16 bits. Je repère donc les octets de redondance et les divise en deux octets de 8 bits. Il ne me reste plus qu'a inverser les bits de chaque octet pour pouvoir ensuite l'implémenter graphiquement comme convenu.

3. Construction de la redondance

Pour implémenter la redondance j'ai décidé de suivre le code de Reed-Solomon. J'ai essayé de m'intéresser au code Reed-Solomon(255,223) mais je ne pense pas avoir toutes les qualités mathématiques pour pouvoir l'implémenter. Je me suis donc inspiré de ce code et ai implémenté une version plus légère de Reed-Solomon. Tous les 5 octets

de message, je code 4 octets de redondance, équivalent à RS(40,32). Bien sûr si la taille du message n'est pas un multiple de 5, les octets restant possède quand même leur redondance. Pour m'assurer d'un taux efficace de correction de l'erreur j'ai décidé d'effectuer cette redondance 2 fois, le « schéma » de ma matrice devient donc : Longueur – Données – Données + Redondance – Données + Redondance – Fin

Plus en détails, la redondance s'effectue avec deux nombres :

a. La somme des (au plus) 5 octets précédents est contenu dans une première variable. Sa valeur maximum étant de 763, il me faut bien plus 8 bits pour la coder.

b. La somme pondérée de ces nombres multiplié par leur rang dans le paquet. Sa valeur maximale étant de 2033, il me bien encore une fois plus de 8 bits pour le coder. Pour simplifier les choses j'ai donc décidé de les coder sur 16 bits chacun.

Encore une fois, il faut faire attention, la redondance de l'information va ici également s'effectuer sur l'octet codant la longueur du message mais pas celui codant la fin qui ne sert ici que de balise. En codant la taille, la fin du message se retrouve facilement.

Au moment de décoder, s'il y a une erreur, elle sera trouvée par une différence entre la somme du décodage et la 1ère valeur de la redondance. La soustraction des deux sera la valeur à modifier (Valeur de l'erreur). Il y aura également une différence entre la 2ème valeur de la redondance et la somme pondérée des nombres multipliés par leur rang, cette nouvelle différence divisée par l'erreur nous donne l'octet à modifier (ou sa partie entière si nombre réel).

4. Constructions Graphiques

Comme dit plus haut, ma représentation graphique se base sur une cible divisée en 64 parties d'aire égale. J'ai décidé de laisser un cercle de 140 pixels de diamètre au centre pour l'esthétique et aussi pouvoir implémenter une image sans altérer le code. Selon la taille du message, l'information est codée entre deux cercles de rayon 130, 180 ou 230 pixels, il existe donc 3 catégories de représentation graphique selon sa taille. Le plus important ici est de savoir combien de cercle nous devons dessiner pour pouvoir faire apparaître toute l'information nécessaire sachant que chaque cercle peut coder 8 octets.

a) Construction des cases : Pour la construction des cases, je dois savoir le nombre de cercle que j'ai à construire et pour cela je dois savoir le nombre d'octets que j'ai à coder. Ceci va s'effectuer en 2 étapes, tout d'abord le calcul en fonction de la redondance choisi me donne la formule

nbCercle = (3*LongueurData + (2*partieEntière(LongueurData/5))+1)/8

Si ce résultat n'est pas un multiple de 8, on rajoute 1 à nbCercle.

Ensuite, du au décalage (égale au nombre de cercle) que nous verrons dans le point suivant, si nbCercle*64 – (nbTotalBits + nbCercle) < 0, nous ajouter au nombre de cercle :

partieEntière(-1*(nbCercle*64 – (nbTotalBits + nbCercle)/8)+1)

Une fois le nombre de cercle nécessaire, nous pouvons diviser chaque cercle en 64 cases de 5,625°. Pour calculer une case, je prends 11 points de chaque angle sur le cercle inférieur et 11 points sur le cercle supérieur. J'ai donc une liste de chacune de mes cases allant de 0 à nbTotalBits comme ma matrice. S'il y a un 1 à l'indice i dans ma matrice, la case i est remplie de noir sinon remplie de blanc.

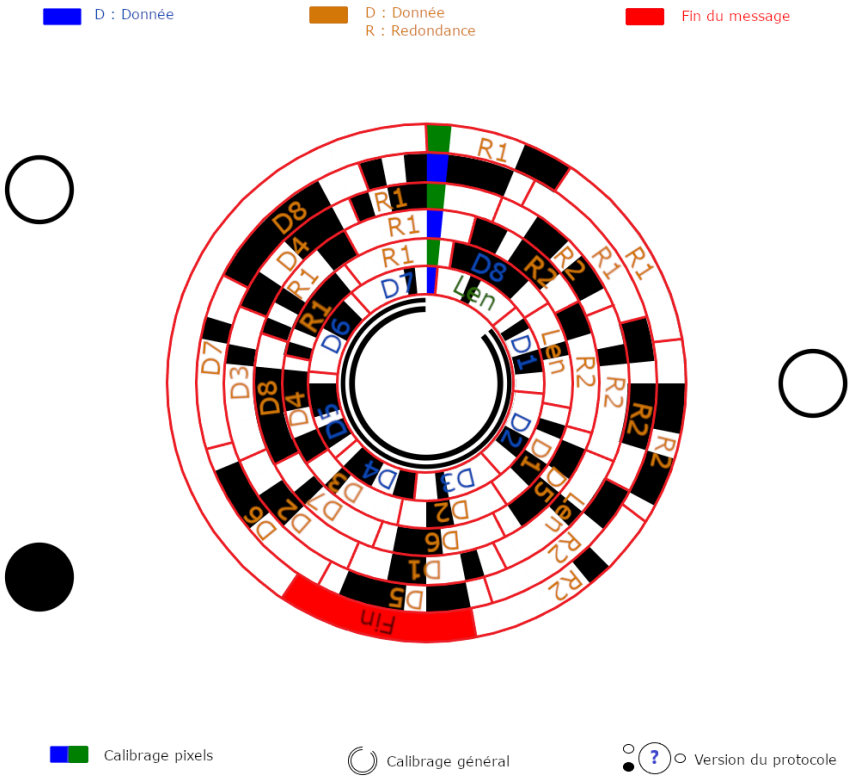
b) Calibrage : Le calibrage s'effectue à l'aide de 3 cercles de plus petites tailles autour du cercle de l'information. Selon la catégorie (et donc la taille) de l'information, le cercle dédié est plein ou vide.

Dans le cercle contenu l'information, chaque case désignant le premier bit de chaque cercle est colorée de bleu ou de vert. Cette méthode informe sur la taille de chaque pixel sur le cercle mais aussi sur la longueur totale du message (une approximation à 8 octets prêt).

Il y a également deux cercles au centre ouvert sur un angle de 50,625° (correspondant à un éventail des 9 premiers bits). Ces cercles sont notamment utiles si le code est tourné dans plusieurs orientations, nous savons que cet angle commence aux marqueurs de couleurs et correspondent à un angle précis, nous ne pouvons pas nous tromper quant à la lecture de ce dernier.

c) <u>Drapeau Fin</u> : Pour marquer la fin du message je fais apparaitre un octet à la fin des 2 redondances en rouge, qui n'est pas pris en compte dans la redondance, ce drapeau est juste physique. Les bits suivant se drapeau sont automatiquement mis en blanc.

Pour résumer, voici un schéma du fonctionnement du protocole :



5. Ajout d'une image

L'ajout d'une image n'est pas chose aisée, son intégration dépend énormément de sa taille et de la taille dédiée à son effet. Dans mon protocole graphique, un cercle de 50 pixels de rayon est disponible (20 pixels pour le calibrage, 50 pixels de vide au milieu). Pour avoir une image correspondante j'ai décidé de de rendre mon image de base carré avec la plus petite valeur entre sa longueur et sa largeur, à la suite de quoi, je rogne cette image pour faire apparaitre un cercle de diamètre de cette valeur. Il y a donc une légère perte dans les 4 coins de l'image mais inévitable au format recherché. Dès que j'ai cette image rogner, je n'ai plus qu'a définir ses dimensions en 50 pixels par 50 pixels et l'intégrer au centre de mon cercle.

6. Détails des fonctions

- ASCII(**chaine**): Ajoute la longueur en premier élément, transforme chaque caractère de la chaine en ASCII

- ajoutRedondance(**li**): Ajoute la redondance à une liste li (de codes ASCII dans notre cas)

- decToBin8or16bits (**n**): prend un nombre n, s'il est inférieur à 128, le code sur 8 bits, sur 16 bits sinon. Cette condition vient du fait que les valeurs ASCII ne dépassent pas 127. Or la redondance peut elle aussi ne pas dépasser 127, pour les différencier, lors de l'application de la redondance, j'ajoute 128 aux bits de redondance et lors de la transformation en binaire tous les nombres supérieurs à 128 désigneront ma redondance, je n'ai plus qu'à enlever 128 à ce résultat et à les coder sur 16 bits.

- change16To8(**li**): prend une liste li et transforme les mots de 16 bits en 2 mots de 8 bits.

- remplaceDecEnBin(**mat**) : Prend une matrice d'entiers, transforme ces entiers en binaire et inverse les bits (le bit 0 devient le bit 7 ou 15 et inversement, pour convenir au remplissage du protocole graphique)

- case(**cercle**): construit le nombre de cases nécessaires au codage de l'information. Chaque case est définie par 22 coordonnées formant la case. La fonction prend le nombre de cercle en paramètre pour connaitre le nombre de cases nécessaire et renvoie une liste de ces coordonnées, la première case est définie par L[0] et contient 22 points.

- rempliData(**li**): prends une liste li étant ma matrice finale avec les bits inversés, si la case est la première de chaque cercle elle est rempli par du bleu ou du vert sinon un entier de décalage est incrémenté, sinon si cette case est un 1 elle est remplie de noir sinon de blanc. Si nous sommes sur le dernier octet, les bits sont coloriés en rouge.

- sw.Drawing('ResultatProtocoleGraphique.svg', profile='full') : Créer un SVG sur lequel dessiner du nom renseigné.

- dwg.polyline(points=final[0], stroke="red",fill="red") : Créer un polygone avec les coordonnées de la première case de la liste final, lui donnant des contours rouges et le remplissant de rouge

- dwg.circle(center=c, r=radiusCercleMilieu, stroke='black',fill='white') : Créer un cercle de coordonnées de centre c , de rayon radiusCercleMilieu, lui donnant des contours noirs et le remplissant de noir.

- dwg.image('final.PNG',insert=(c[0],c[1])) : Insère une image de nom final.PNG aux coordonnées (c[0],c[1]). Ces coordonnées correspondent au coin supérieur gauche de l'image.

III. Améliorations Possible

En prenant mon projet avec des pincettes et un esprit critique, je me suis rendu compte que certaines améliorations pourraient être intéressantes :

- Dans mon protocole, les cases de deux cercles différents n'ont pas la même taille, ce qui m'oblige à mettre un témoin graphique pour les déterminer. Il serait peut-être plus pratique de multiplier le nombre de case d'un cercle à l'autre par **PI**, ainsi les cases auraient la même taille et on coderait $\mathbf{PI}^{\text{nbCercle}}$ fois plus d'information qu'avec le modèle actuel. Le nombre de cercle serait alors redéfini et diminué, l'information serait alors peut-être plus lisible car sa taille (en pixel) serait définie à l'avance.

- L'ajout d'une meilleure redondance est une pratique indispensable pour effectuer la meilleure correction possible. Le code de Reed-Solomon (255,223) est un code dit parfait qui ne laisse passer aucune erreur, son implantation sera donc un plus pour ce protocole graphique

- Toujours en termes de redondance, l'apparition de celle-ci n'est pas assez cyclique, deux blocs apparaissent l'un après l'autre (Données-Redondance puis Données-Redondance), il faudrait mieux mélanger ces blocs de données pour permettre un taux de correction maximal mais ceci pourrait poser problème lors du décodage.

IV. Conclusion

Pour conclure, j'ai bien aimé travailler sur ce projet car la vision de sa progression est très motivante et satisfaisante. J'ai pu découvrir des bibliothèques inconnues et améliorer mon niveau dans le langage Python qui me parait être un langage indispensable à maitriser. J'ai également été très curieux sur le domaine du réseau et de la transmission de données mais aussi du lien avec les Mathématiques pour la sécurisation de l'envoie et la redondance (Code de Reed-Solomon).