930 Route des Colles

06410 Biot

Tél : 04 92 96 50 50

www.polytechnice.fr/

Team TELA

Stéganographie

Prototypage

Loïck MAHIEUX

Abdelkarim ANDOLERZA

Thomas GILLOT

Étienne LASKAR

Table des matières

[I. Choix des prototypes à réaliser 3](#_Toc432603221)

[II. Dissimulation en Java 4](#_Toc432603222)

[A. Récupération et manipulation de fichier 4](#_Toc432603223)

[B. Manipulation d’image 10](#_Toc432603224)

[1. Afficher les pixels d’une image 10](#_Toc432603225)

[2. Manipulation les pixels d’une image 11](#_Toc432603226)

[3. Manipulation des pixels d’une image en binaire 12](#_Toc432603227)

[4. Mesurer la dégration d’une image 14](#_Toc432603228)

[III. Révélation en C 15](#_Toc432603229)

[A. Lecture d’une image pixel par pixel 15](#_Toc432603230)

[B. Manipulation de fichiers 15](#_Toc432603231)

[IV. Bibliographie 16](#_Toc432603232)

# Choix des prototypes à réaliser

Dans le cadre de ce projet nous nous **orientons** sur une **dissimulation** du message en **Java** et une **révélation** de celui-ci en **C** selon le schéma suivant :

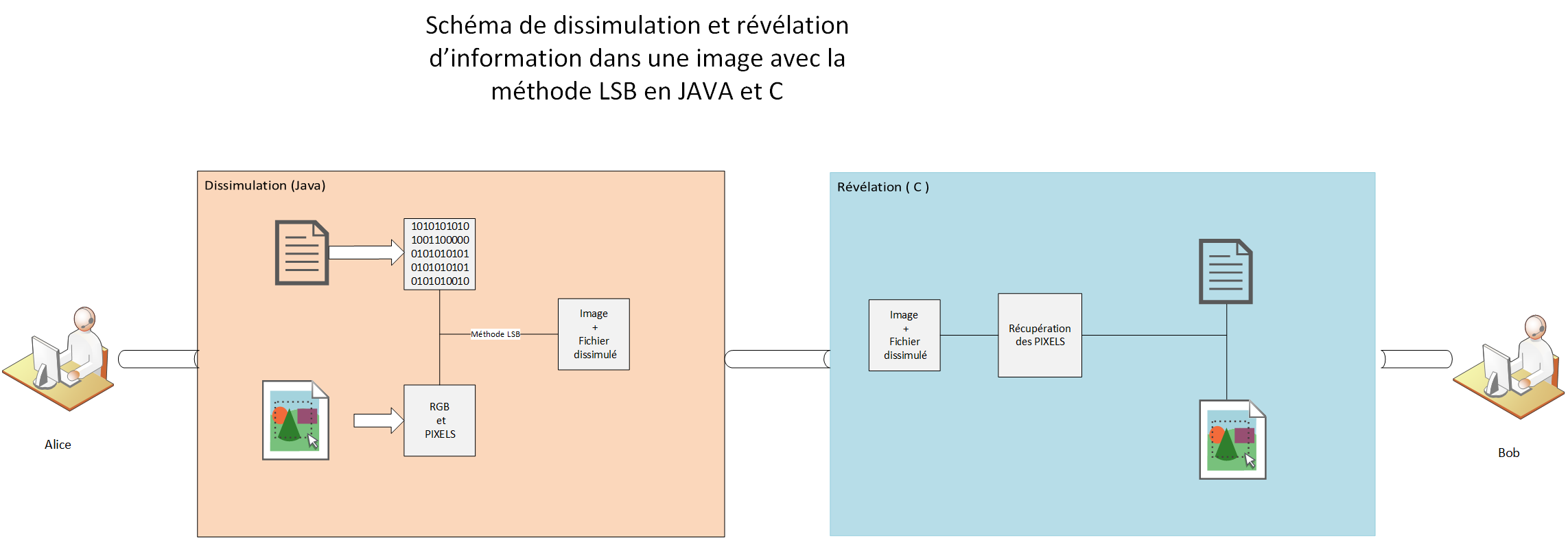


Figure : Schéma général du projet

Pour notre **choix** de **prototypes**, nous nous sommes vraiment **appuyés** sur ce **schéma** afin de **déterminer** exactement les **points** sur lesquels nous allons **travailler**.

# Dissimulation en Java

## Récupération et manipulation de fichier

* Pourquoi ce prototype était nécessaire pour votre équipe vis à vis du projet ?

Pouvoir mettre en œuvre un prototype de récupération et manipulation est essentiel dans un projet tel que celui la afin de pouvoir aussi bien insérer que cacher des donner binaires dans un fichier tel qu’un texte ou une image.

Une description du prototype (les “bouts de code” intéressant, les difficultés associés), une description des bénéfices que ce prototype apportera à votre projet lorsqu'on basculera en phase d'implémentation.

* Récupération du type de fichier

Pourquoi ce prototype était nécessaire pour votre équipe vis à vis du projet?

Ce prototype était nécessaire pour notre projet car cela va nous permettre de savoir quelle manipulation faire sur le fichier en question. En effet, on ne fera pas les mêmes manipulations de Bit et l’ouverture du fichier ne sera pas la même (c’est-à-dire que les bibliothèques à utiliser ne seront pas les mêmes).

Nous avons donc cherché sur Internet les différentes bibliothèques disponibles sur l’API Java. Nous avons trouvé deux méthodes permettant de faire cela :

* la méthode *getContentType(File f)* de l’objet MimetypesFileTypeMap qui prend un objet de type “File” en paramètre et qui renvoie un String de type (par exemple pour fichier .txt) “plain/txt”. Le problème de cette méthode est qu’elle ne se contente que d’analyser l'extension du fichier, de plus pour certains fichiers il ne donne pas le type précis de fichier (comme les fichiers .zip) et il retourne la chaîne “application/octet-stream”. Mais malgré cela cela distingue quand mêmes les familles de fichiers les plus importantes(les images et les fichiers texte par exemple).
* la méthode static “public abstract String probeContentType(Path path)” (tirée de l’API Java) de l’objet FileTypeDetector. Elle prend en paramètre un objet de type Path. Elle renvoie un String (comme la méthode getContentType) du type (pour un fichier .zip) “application/zip”. Elle décrit le type MIME c’est-à-dire un identifiant de format de données sur internet (comme la précédente d’ailleurs).

Par rapport à la méthode précédente elle s’appuie sur l’extension du fichier et pour certains fichiers sur les octets du fichier. Il y a donc plus de précision avec cette méthode qu’avec la dernière. Mais, il y a des inconvénients à cette méthode notamment c’est un static (ce qui a été déconseillé dans le retour de la démonstration numéro 1). Il y a aussi un autre problème, comme pour la précédente méthode, c’est qu’elle se base principalement sur l’extension.

Les deux méthodes se basant sur l’extension et même si l’extension n’est pas complètement fiable, il faut se poser la question si il ne voudrait pas mieux récupérer directement l’extension du fichier. Nous avons donc créé un objet pour récupérer l’extension directement.

Voici les deux classes que nous avons implémentées pour la détection du type de fichier :

* Classe utilisant la bibliothèque FileTypeDetector

*public class Mime {*

*private File f;*

*//private MimetypesFileTypeMap mT ;*

*public Mime (File f) {*

*this.f = f;*

*//mT = new MimetypesFileTypeMap();*

*}*

*public void getMime () throws IOException {*

*String type[] = Files.probeContentType(f.toPath()).split("/");*

*System.out.println(type[1]);*

*}*

*}*

* Classe ne se basant que sur l’extension du fichier

*public class Extension {*

*private File f;*

*public Fin(File f) {*

*this.f = f;*

*}*

*public void getExtention(){*

*String name = f.getName();*

*String ext = name.substring(name.lastIndexOf("."));*

*System.out.println(ext.substring(1));*

*}*

*}*

* Manipulation binaire des fichiers

Pourquoi ce prototype était nécessaire pour votre équipe vis à vis du projet?

Pour pouvoir manipuler et dissimuler un fichier dans une image, il faut comprendre la structure des fichiers en général. C’est ainsi que nous avons travaillé sur le fichier texte qui est facilement compréhensible en premier lieu.

Nous avons donc travaillé sur différents aspects de la manipulation de fichiers avec en premier lieu l’ouverture d’un fichier texte. Pour se faire, nous nous sommes renseignés sur différents sites qui proposent tous approximativement la même solution, disponible dans la librairie java.io intégrée au JDK Java. Cette solution est d’utiliser les objets **InputStream**, **InputStreamReader** ainsi que **BufferReader**. Selon la javadoc en ligne, l’objet InputStream permet de charger un fichier passé en paramètre (via un lien relatif à l’endroit où l’on appelle cet objet, ou un lien brut. Le deuxième objet, quand à lui, prendra en paramètre l’InputStream créé précédemment et va servir de “pont” afin de convertir des octets en caractères. Le dernier objet à être créé, BufferReader va en toute logique prendre en paramètre l’InputStreamReader afin de pouvoir lire des lignes complètes de fichiers (dans le cas d’un fichier texte comme le nôtre par exemple). Vous trouverez ci-dessous le code fait dans l’équipe pour la lecture de fichier :

//Récupérer le contenu d'un fichier en décimal, binaire et en chaîne de caractère

public static byte[] getContenu(File f) throws IOException {

byte[] lignes = new byte[(int) f.length()];

String ligne;

InputStream ips = new FileInputStream(f);

InputStreamReader ipsr = new InputStreamReader(ips);

BufferedReader br = new BufferedReader(ipsr);

while ((ligne = br.readLine()) != null) {

lignes = ligne.getBytes("UTF-8");

}

//Fermeture du bufferReader afin de libérer des ressources qui lui sont confiées

br.close();

return lignes;

}

Bien entendu, dans ce code, nous faisons un double travail puisque lorsque nous récupérons le caractère dans la boucle while, nous demandons son **byte**, ce qui n’est pas nécessaire dans les cas courants, mais nous l’avons utilisé afin de manipuler les méthodes proposées.   
 Lorsque l’on retourne **lignes,** qui est un tableau de byte, on l’envoie à une autre méthode qui permet d’afficher sur la console le contenu du fichier.

Voici le contenu de cette méthode :

public static void afficherContenu(byte[] contenu) {

System.out.println("Affichage du contenu du fichier : ");

System.out.println("Caractère -> Byte -> Binaire \n");

for (byte b : contenu) {

System.out.println((char) b + "-> " + b + "-> "

+ Integer.toBinaryString(b));

}

System.out.println("Fin de lecture \n");

}

Nous voyons ici que l’on peut facilement récupérer un caractère grâce à des bytes en procédant à un **cast** de la valeur. Pour la valeur en binaire, il suffit d’utiliser l’objet **Integer** qui propose une conversion d’un byte en binaire, en le stockant dans une chaine **String.**

En sortie console, nous avons ainsi affiché pour un fichier texte ceci :

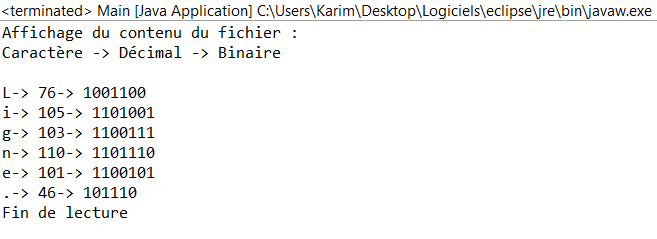


Figure : Console lecture d'un fichier texte

Après avoir réussi à ouvrir un fichier on a eu une idée, qui est de modifier le contenu grâce aux bytes obtenus. La méthode ci-dessous montre le travail effectué. Bien entendu, la modification est basique, mais on a préféré la garder dans le cas où nous devrions modifier le texte à l’intérieur du fichier afin de dissimuler encore une fois l’information. Ceci n’est qu’une hypothèse du travail attendu et des specs.

//Récuperer un tableau de décimaux et modifier le contenu

public static byte[] modifierFichier(byte[] contenu) {

int i = 0;

for (byte b : contenu) {

b += 10;

contenu[i] = b;

i++;

}

return contenu;

}

En rajoutant +10 aux bytes lus dans le tableau, nous obtenons ceci pour le fichier texte contenant “Ligne.” montré précédemment :

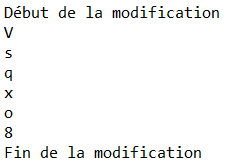


Figure : Console de modification de fichier texte

En dernier lieu, nous avons voulu créer un fichier, afin de pouvoir y stocker le nouveau contenu par exemple, ou pour une autre utilisation que l’on découvrira lors des specs. Afin de réussir cette tâche, nous avons encore une fois consulté plusieurs sources et les techniques ainsi que les avis se rejoignent. Java nous propose **FileWriter** et **BufferedWriter** pour respectivement créer le fichier et ensuite écrire à l'intérieur. Les sites visités proposent en tuto de créer un fichier en y insérant un String. Cependant nous voulions insérer le contenu présent dans notre **byte[]** présenté précedemment. L’idée était donc de concaténer les bytes castés en char dans un String pour se ramener à la solution proposée dans les tutos. On voit que la concaténation n’est ici qu’un += de String ce qui n’est pas vraiment adapté en général. Nous nous y attelleront bien entendu afin de transformer ça en **StringBuilder** qui propose une méthode **append()** afin de concaténer proprement tout en étant plus efficace.

//Récupérer le contenu d'un fichier en décimal et créé un fichier

public static void creerFichierMod(byte[] cont) throws IOException {

File file = new File("./testMod.txt");

String content = "";

for (byte b : cont) {

content += (char) b;

}

FileWriter fw = new FileWriter(file.getAbsoluteFile());

BufferedWriter bw = new BufferedWriter(fw);

bw.write(content);

bw.close();

System.out.println("Fichier modifié créé");

}

Le plus difficile lors de cette phase de prototypage, sera de récupérer les bits du fichiers afin de les manipuler, et si ce n’est vraiment pas possible, rester au format des byte.

## Manipulation d’image

### Afficher les pixels d’une image

Pour ce projet, il est **essentiel** de savoir **manipuler** des **images**. Nous avons donc décidé de faire des **prototypes** nous permettant de nous **familiariser** avec les images.

Pour commencer, nous avons cherché un moyen de **lire** une **image**. Nous avions un **objectif** précis, **afficher** les **pixels** en RGB d’une image.

Cependant, une image peut-être sous **différents** **formats**. Pour cela, nous avons opté pour une **bibliothèque** inclue dans Java, **ImageIO**.

ImageIO permet de créer un **BufferedImage** à partir d’un fichier. ImageIO gère les formats suivants :

* JPEG
* PNG
* BMP
* WBMP
* GIF

Une fois le BufferedImage créé, il est facile de parcourir une image pixel par pixel.

public class ImageReader {

private File file;

private BufferedImage image;

public ImageReader(String filename) throws IOException {

file = new File(filename);

image = ImageIO.*read*(file);

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

for (int i = 0; i < image.getWidth(); i++) {

for (int j = 0; j < image.getHeight(); j++) {

Color c = new Color(image.getRGB(i, j));

sb.append(c.toString() + "\n");

}

}

return sb.toString();

}

}

### Manipulation les pixels d’une image

Par la suite, nous avons décidé de réaliser divers prototypes de manipulation d’image.

Nous avons réalisé divers filtres pour se **familiariser** avec la manipulation pixel par pixel.

Le premier **prototype** permet de **passer** une **image** en **couleur** en **nuances** **de** **gris**.

Il est donc nécessaire de parcourir l’image pixel par pixel mais cette fois-ci, il faut pouvoir les modifier.

Toutefois, pour réaliser ce protoype, **plusieurs** **options** s’offrent à nous. Nous avons donc pris la décision **d’implémenter** deux **solutions** **différentes** afin d’en **déterminer** qu’elle est la **meilleure**.

La première méthode consiste en l’utilisation d’un objet **Color** qui représente le pixel. Il est donc possible avec cet objet de lire le **RGB**. Par contre, il n’est **pas possible** de **modifier** de un objet Color, il est donc nécessaire de **créer deux objets Color** lors de la modification d’un pixel.

public void filter() throws IOException {

for (int i = 0; i < image.getWidth(); ++i) {

for (int j = 0; j < image.getHeight(); j++) {

Color color = new Color(image.getRGB(i, j));

int c = (color.getRed() + color.getGreen() + color.getBlue()) / 3;

color = new Color(c, c, c);

image.setRGB(i, j, color.getRGB());

}

}

}

L’autre méthode est d’utilisation un **Raster**. Il représente, tout comme l’objet **Color**, un pixel.

Il existe également un objet **WritableRaster**, qui tout comme le Raster représente un pixel mais **autorise** également la **modification** de celui-ci.

public void filterRaster() {

WritableRaster raster = image.getRaster();

for (int i = 0; i < raster.getWidth(); ++i) {

for (int j = 0; j < raster.getHeight(); j++) {

int pixel[] = new int[3];

raster.getPixel(i, j, pixel);

int c = (pixel[0] + pixel[1] + pixel[2]) / 3;

pixel[0] = c; pixel[1] = c; pixel[2] = c;

raster.setPixel(i, j, pixel);

}

}

}

Après différents **tests**, nous avons eu des **résultats** beaucoup **plus** **rapides** avec l’utilisation d’un **Raster**. À titre d’exemple, lors du passage en nuances de gris d’une image en couleur, en utilisant l’objet **Color**, le temps d’exécution moyen en d’environ **800ms**, avec le **Raster**, il n’est que de **250ms** en moyenne.

**// faire le rapport avec le schéma**

### Manipulation des pixels d’une image en binaire

Dans un second temps, il nous a semblé nécessaire de **prototyper** la **manipulation** de **bits** dans les pixels d’une image pour par la suite, pouvoir manipuler les bits des pixels pour appliquer la **méthode** du **LSB**.

Dans la méthode suivante, nous nous sommes contenté pour l’instant de faire un “**ou**” **bit à bit** sur chaque **composante** du pixel afin de pouvoir par la suite tester **combien** de **bits** il est possible de **modifier** par **composante**.

public void modifier(int r, int g, int b) throws IOException {

WritableRaster raster = image.getRaster();

for (int i = 0; i < image.getWidth(); ++i) {

for (int j = 0; j < image.getHeight(); j++) {

int pixel[] = new int[3];

raster.getPixel(i, j, pixel);

pixel[0] |= r;

pixel[1] |= g;

pixel[2] |= b;

raster.setPixel(i, j, pixel);

}

}

}

### Mesurer la dégration d’une image

Pour finir, il est nécessaire de pouvoir **mesurer** la **dégradation** d’une **image** pour vérifier que le fait de **cacher** un **fichier** dans une image en utilisant la **méthode** **LSB** ne va pas complètement modifier l’image et ainsi rendre **repérable** la **dissimulation**.

Pour cela, nous avons prototypé une **première** méthode qui consiste à calculer **l’erreur** **relative** de chaque **composante** de chaque **pixel**.

**public Double compareRaster(File file) throws IOException, NotComparableException {**

**BufferedImage image2 = ImageIO.read(file);**

**if(image.getWidth() != image2.getWidth() || image.getHeight() != image2.getHeight())**

**throw new NotComparableException();**

**Double diff = 0.;**

**Raster raster = image.getRaster();**

**Raster raster2 = image2.getRaster();**

**for (int i = 0; i < raster.getWidth(); i++) {**

**for (int j = 0; j < raster.getHeight(); j++) {**

**int rgb[] = new int[3];**

**int rgb2[] = new int[3];**

**raster.getPixel(i, j, rgb);**

**raster2.getPixel(i, j, rgb2);**

**diff += Math.abs((rgb2[0] - (rgb[0] + 1)) \* 100) / (rgb[0] + 1);**

**diff += Math.abs((rgb2[1] - (rgb[1] + 1)) \* 100) / (rgb[1] + 1);**

**diff += Math.abs((rgb2[2] - (rgb[2] + 1)) \* 100) / (rgb[2] + 1);**

**}**

**}**

**return diff / (double)(raster2.getWidth() \* raster.getHeight() \* 3);**

**}**

**// corrélation**

# Révélation en C

## Lecture d’une image pixel par pixel

## Manipulation de fichiers

**// Écriture d’un fichier à partir de bits**

[Figure 1 : Schéma général du projet 3](#_Toc432603843)

[Figure 2 : Console lecture d'un fichier texte 7](#_Toc432603844)

[Figure 3 : Console de modification de fichier texte 8](#_Toc432603845)

# Bibliographie

* <http://docs.oracle.com/javase/7/docs/api/javax/activation/MimetypesFileTypeMap.html>
* <https://docs.oracle.com/javase/7/docs/api/java/nio/file/Path.html>
* <https://docs.oracle.com/javase/7/docs/api/java/nio/file/spi/FileTypeDetector.html>
* <https://www.gnu.org/software/classpathx/jaf/javadoc/javax/activation/MimetypesFileTypeMap.html>
* <http://docs.oracle.com/javase/8/docs/api/javax/imageio/package-summary.html>
* <http://b.kostrzewa.free.fr/java/td-fichiers/texte.html> :
  + Définition InputStream / InputStreamReader / BufferReader
* <http://www.mkyong.com/java/how-to-write-to-file-in-java-bufferedwriter-example/>
  + FileWriter / BufferedWriter