

Université / U.F.R. des Sciences et Techniques

Compte-Rendu de TP Réseau

TP3 : Le Code de Hamming

Auteur :

REDJRADJ YACINE

Année Universitaire 2025-2026

Table des matières

1	Rappel du sujet	2
2	Analyse du sujet	2
2.1	Structure du mot de Hamming	2
2.2	Méthode de calcul et de vérification	2
3	Choix techniques effectués	2
3.1	Architecture MVC	2
3.2	Algorithme de positionnement	3
4	Résultats et tests	3
4.1	Génération d'un code de Hamming	3
4.2	Vérification d'un message valide	4
4.3	Correction d'erreur	4
5	Conclusion	5

1 Rappel du sujet

L'objectif de ce TP est d'implanter un programme en langage JAVA permettant de manipuler le Code de Hamming, un code correcteur d'erreurs linéaire. Le programme doit remplir deux fonctions principales décrites dans le sujet :

- **Le calcul d'un code de Hamming (Émetteur)** : À partir d'une suite de bits donnée, calculer les bits de contrôle de parité et générer le mot complet.
- **La vérification d'un message (Récepteur)** : Vérifier si un mot reçu contient des erreurs et localiser la position de l'erreur éventuelle.

Le programme doit également afficher les étapes intermédiaires ayant abouti au résultat.

2 Analyse du sujet

Le code de Hamming est un code de bloc capable de détecter jusqu'à deux erreurs et d'en corriger une seule sur le mot transmis.

2.1 Structure du mot de Hamming

Un mot de Hamming est défini par sa longueur totale x et la longueur du message utile y (forme "x-y"). La relation fondamentale liant le nombre de bits de parité r (ou i) à la longueur du message m est :

$$2^r \geq m + r + 1$$

Les bits de contrôle sont placés aux positions correspondant aux puissances de 2 (1, 2, 4, 8, ...).

2.2 Méthode de calcul et de vérification

1. **Génération** : Chaque bit de parité P_i est calculé en effectuant un OU Exclusif (XOR) sur un sous-ensemble spécifique des bits de données. Par exemple, P_1 couvre toutes les positions dont le bit de poids faible de l'index binaire est 1 (1, 3, 5, 7...).
2. **Vérification (Syndrome)** : À la réception, on recalcule les parités. Si le résultat (le syndrome) est 0, le message est correct. Sinon, la valeur du syndrome indique directement la position binaire de l'erreur à corriger.

3 Choix techniques effectués

Pour assurer la maintenabilité et la clarté du code, j'ai reconduit l'architecture utilisée lors du TP précédent.

3.1 Architecture MVC

- **Package Models (HammingCalcul.java)** : Contient toute la logique mathématique. J'ai utilisé des tableaux d'entiers (`int[]`) pour faciliter l'accès aux indices $1 \dots N$, ce qui est plus naturel pour Hamming que les indices $0 \dots N - 1$ des tableaux Java classiques.
- **Package Views** : Interface graphique Swing permettant la saisie du message et l'affichage des résultats.

- **Capture des logs** : Réutilisation de la classe `LogCapture` pour intercepter les `System.out.println` et afficher les détails du calcul du syndrome dans l'interface.

3.2 Algorithme de positionnement

L'une des difficultés techniques était de placer correctement les bits de données en sautant les indices 1, 2, 4, 8. J'ai implémenté une boucle qui vérifie si l'index courant est une puissance de 2 :

```
if ((Math.ceil(Math.log(i)/Math.log(2)) -  
    Math.floor(Math.log(i)/Math.log(2))) == 0) {  
    // C'est une position de controle (parite)  
}
```

4 Résultats et tests

Voici les tests de validation effectués avec le programme.

4.1 Génération d'un code de Hamming

Nous testons avec le message binaire 1011 ($m = 4$). Le programme calcule qu'il faut 3 bits de parité ($r = 3$) car $2^3 \geq 4 + 3 + 1$. Comme le montre la Figure 1, le mot généré est 0110011. Les étapes montrent le calcul de P1, P2 et P4.

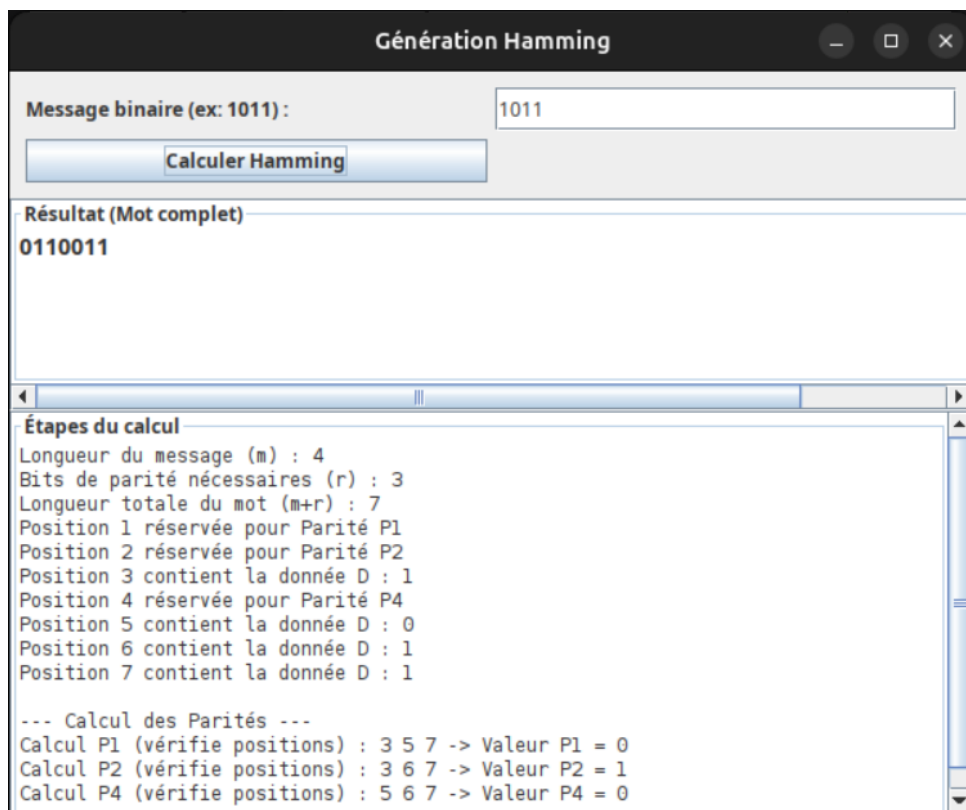


FIGURE 1 – Génération du code Hamming pour le message 1011

4.2 Vérification d'un message valide

Nous insérons le message précédemment calculé (0110011) dans le module de vérification. Le programme recalcule les parités. Le syndrome étant nul, il indique que le message est **CORRECT** (Figure 2).

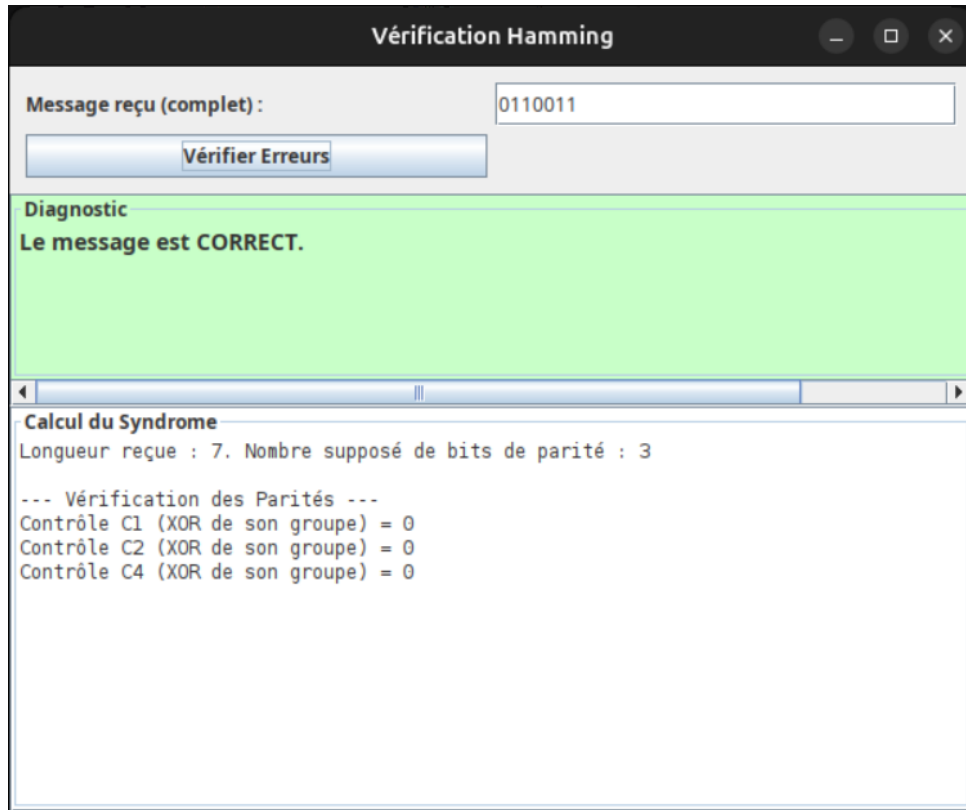


FIGURE 2 – Vérification d'un message sans erreur

4.3 Correction d'erreur

Pour tester la capacité de correction, nous modifions le bit à la position 7 (le dernier bit passe de 1 à 0) : message 0110010. Le programme détecte l'erreur et identifie correctement la position 7 (Figure 3), ce qui permettrait de corriger le bit automatiquement.

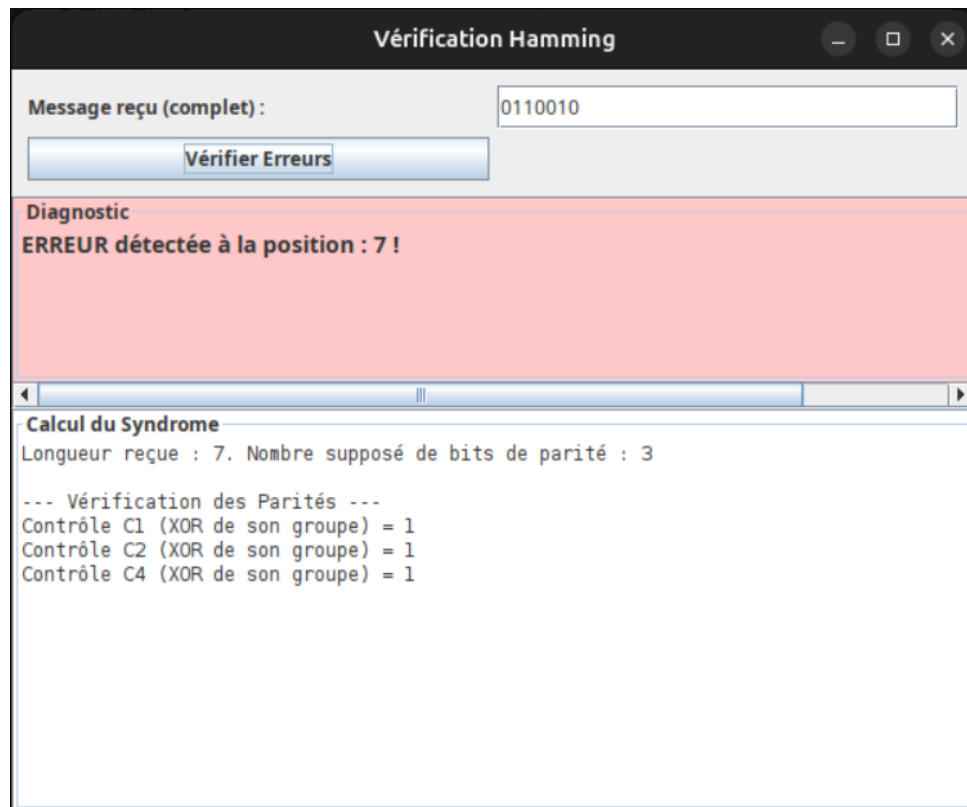


FIGURE 3 – Détection d’erreur et localisation de la position

5 Conclusion

Ce TP a permis de mettre en pratique l’algorithme de Hamming. Contrairement au CRC qui repose sur une division polynomiale, Hamming repose sur une gestion matricielle et positionnelle des bits.

Difficultés rencontrées : La principale difficulté a été la gestion des indices. En informatique, les tableaux commencent à 0, mais la théorie de Hamming utilise des positions commençant à 1. Une adaptation a été nécessaire dans la classe `HammingCalcul` pour éviter les erreurs de décalage ("Off-by-one errors").

Perspectives : Le programme actuel gère la détection et la localisation de l’erreur simple. Une amélioration serait d’ajouter un bit de parité global (Hamming Étendu) pour être capable de distinguer une erreur simple d’une erreur double.