SAE 3.02 : Conception d'une architecture distribuée avec routage en oignon, Documentation

Sommaire :

Note : les mots soulignés et avec un * seront définit dans le Glossaire

1. Introduction :

1.1 Présentation du projet :

Ce projet s'inscrit dans le cadre de la SAE 3.02 : Conception d'une architecture distribuée avec routage en oignon.
L'objectif est de permettre l'échange de messages sécurisés entre plusieurs utilisateurs. Dans un système de routage en oignon*, lorsqu'un utilisateur (Client A) souhaite envoyer un message à un autre (Client B), le message ne suit pas un chemin direct. Il transite par plusieurs routeurs* intermédiaires qui servent de relais.
Pour protéger la vie privée, nous utilisons le chiffrement asymétrique*. Le principe est le suivant : l'expéditeur "enveloppe" son message dans plusieurs couches de protection successives. À chaque fois que le message passe par un routeur, celui-ci retire une couche (il "épluche" l'oignon). Ce n'est qu'une fois arrivé au destinataire final que le message est entièrement déchiffré et lisible.
L'atout majeur de cette architecture est la confidentialité : chaque routeur ne connaît que l'appareil qui lui a envoyé le paquet et celui à qui il doit le transmettre. Personne ne connaît le chemin complet, ce qui garantit un anonymat total des échanges.

1.2 Le fonctionnement de l'oignon et des nœuds hybrides :

Pour bien comprendre comment les messages circulent, il faut imaginer que chaque participant au réseau est un nœud hybride*. Cela signifie qu'un même ordinateur peut changer de rôle selon la situation :

- Le rôle de Client (Expéditeur/Destinataire) : C'est l'ordinateur qui prépare le message. Pour garantir l'anonymat, il ne se contente pas d'envoyer le texte. Il l'enveloppe dans plusieurs couches de chiffrement, un peu comme si on mettait une lettre dans plusieurs boîtes imbriquées, chacune fermée par un cadenas différent.

- Le rôle de Routeur (Relais*) : Lorsqu'un nœud reçoit un message qui ne lui est pas destiné, il agit comme un relais. Il possède la clé pour ouvrir la boîte extérieure (la première couche).

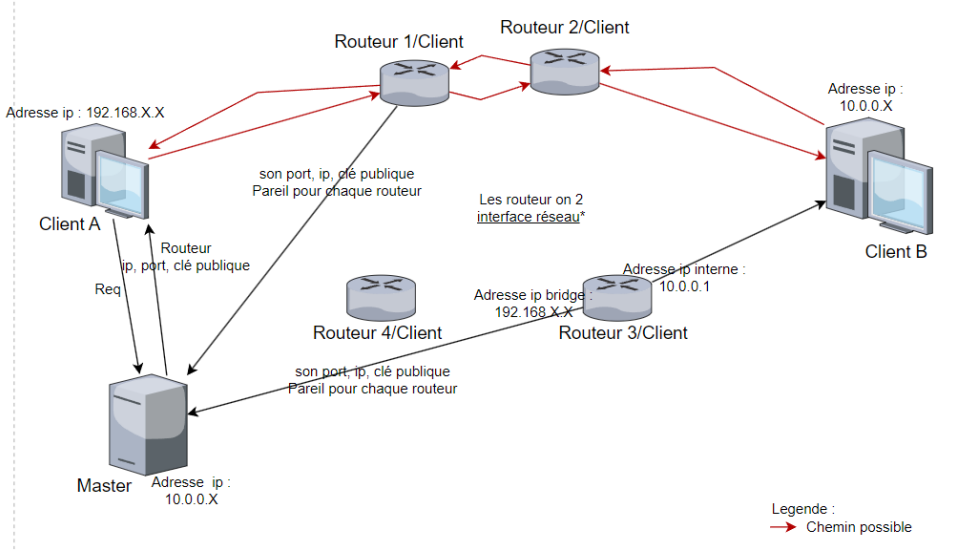
Une fois cette couche retirée, il découvre l'adresse du prochain destinataire et lui transmet le paquet restant, sans jamais pouvoir lire le message caché tout au centre.

- Le rôle indispensable du Master (Le Chef de Gare)
  Pour que ces échanges soient possibles, il faut que les clients sachent quels sont les routeurs disponibles à un instant T. C'est le rôle du Master. C'est un programme central qui sert d'annuaire :
  Chaque routeur qui s'allume s'enregistre au Master.
  Le Master enregistre leur adresse et leur "cadenas" (clé publique) dans une base de données.
  Quand un client veut envoyer un message, il demande au Master la liste des routeurs actifs pour pouvoir construire son circuit.
  L'avantage de l'hybride* : Puisque chaque machine du réseau peut être à la fois un client et un routeur, un observateur extérieur ne peut pas savoir si le message que vous envoyez vient de vous ou si vous ne faites que le faire passer pour quelqu'un d'autre. C'est ce mélange des rôles et l'organisation du Master qui rendent le système si robuste.

2. Architecture système :

2.1. Architecture réseaux (Schéma) :

L'architecture mise en place est une infrastructure distribuée hybride. Comme l'illustre le schéma ci-dessous, elle interconnecte* un hôte* physique et plusieurs machines virtuelles* à travers deux segments* réseaux bridge* et réseaux intnet* ce sont 2 réseau distincts :



2.2. Segmentation* Réseau et Adressage* :

Pour simuler un environnement réel et sécurisé, j'ai séparé le réseau en deux zones distinctes. Cette configuration permet de reproduire le passage d'un réseau public vers un réseau privé* :

- Le Réseau Bridge (192.168.X.X) : Ce segment est relié à mon réseau physique (Box domestique, 5G, etc.). Il sert de point d'entrée pour le Client A (installé sur le PC physique). C'est le lien entre le monde réel et l'infrastructure virtuelle.

- Le Réseau Interne / Intnet (10.0.0.X) : Ce segment est totalement isolé de l'extérieur. Il héberge le Master, le Client B ainsi que les interfaces* internes des routeurs. Il assure la communication privée et confidentielle entre les composants du système.

2.3. Les Nœuds Hybrides et la Double Interface :

L'une des particularités techniques majeures de ce réseau est la configuration des routeurs sur deux interfaces simultanées (Bridge et Intnet) :

- Le rôle de Pont : Les routeurs font office de passerelle*. Ils permettent la communication entre l'ordinateur physique et les machines isolées dans le réseau interne.
- La nature Hybride : Contrairement à un réseau classique, on parle ici de routeurs hybrides*. Cela signifie que chaque nœud possède une double capacité : il peut agir comme un simple relais (routeur) pour transmettre les données des autres, ou comme un utilisateur final (client) capable d'émettre et de recevoir ses propres messages.

2.4. Flux de Contrôle* et Annuaire (Le Master) :

Le Master agit comme le serveur central et le gestionnaire du réseau. Son rôle est d'organiser la rencontre entre les nœuds :

- Gestion des données : Il stocke dans une base de données MariaDB* les informations vitales des routeurs actifs : adresses IP, numéros de ports et clés publiques*.
- Interrogation : Lorsqu'un client souhaite envoyer un message, il émet une requête au Master pour obtenir cet annuaire. Ce flux de contrôle est indispensable avant tout envoi pour que le client puisse construire son circuit de chiffrement.

2.5. Cheminement des Messages :

Les flèches rouges sur le schéma représentent un "chemin possible" parmi une multitude de combinaisons. Le message ne suit pas une ligne droite : il "rebondit" de nœud* en nœud. À chaque étape, l'interface réseau correspondante (Bridge ou Intnet) est sollicitée pour faire transiter le paquet* d'un segment à l'autre. Ce mécanisme garantit que le Client A et le Client B peuvent communiquer de bout en bout sans jamais être en contact direct, assurant ainsi l'anonymat recherché.

3. Protocoles et Communication :

3.1. Gestion des Sockets*, des Protocoles* et des Threads* :

Pour que notre infrastructure puisse fonctionner et échanger des données en temps réel, j'ai combiné trois technologies fondamentales : les Sockets pour la connexion, deux types de Protocoles pour le transport*, et les Threads pour la gestion du multi-tâche.

A. Les Sockets : Le point de contact

Le Socket est l'interface logicielle qui permet à mes programmes Python de communiquer à travers le réseau.

- Le Serveur (Master et Routeurs) : Il reste en mode "écoute" sur un Port* spécifique, attendant qu'une entité vienne solliciter une connexion.
- Le Client : Il initie l'échange en ciblant l'adresse IP et le Port du destinataire.

B. TCP* et UDP* : Choisir le bon transport

Nous avons fait le choix d'utiliser deux protocoles différents selon la nature de l'échange :

- Le choix du TCP (Transmission Control Protocol) : j'utilise pour le transfert des messages (l'oignon). C'est un protocole "fiable" qui garantit que les données arrivent complètes et dans le bon ordre. C'est crucial pour le chiffrement : si un seul octet est perdu ou inversé, le message final devient indéchiffrable.
- L'utilisation de l'UDP (User Datagram Protocol) : Pour le Master, j'ai privilégié l'UDP pour les échanges rapides, comme lorsqu'un routeur signale simplement sa présence ("Je suis en ligne"). C'est un protocole beaucoup plus léger et rapide que le TCP, idéal pour de petites notifications qui ne nécessitent pas d'établir une connexion lourde.

C. Les Threads : Le multi-tâche indispensable :

Sans les Threads, mon réseau serait "bloqué". En informatique, une opération de lecture sur un socket est dite "bloquante*" : le programme s'arrête tant qu'il n'a pas reçu de message.

- Le problème : Si le Master attend un message du Routeur 1, il ne peut pas répondre au Client A.
- La solution : Le Multi-threading* : L'analogie : Imaginez un serveur dans un restaurant. S'il devait attendre qu'un client finisse de manger pour s'occuper du suivant, le service serait impossible. Le multi-threading permet au serveur de s'occuper de plusieurs tables en même temps.
- En pratique : Grâce à la bibliothèque threading de Python, chaque fois qu'une nouvelle connexion arrive, le programme principal crée un Thread (un petit processus* indépendant).
- Résultat : Cela permet au Master de gérer des dizaines de routeurs simultanément et à chaque routeur hybride de relayer des messages tout en restant disponible pour ses propres fonctions de client.

3.2. Format des paquets : Structure de l'oignon :

A. Un protocole "fait maison" sans JSON :

Pour remplacer le format JSON, J'ai utilisé la technique de la concaténation*. Toutes les informations nécessaires au voyage du message sont collées les unes aux autres, séparées par un délimiteur* (par exemple, le caractère |).

Chaque paquet brut reçu par un routeur ressemble par exemple a ça : ADRESSE_IP_SUIVANTE | PORT_SUIVANT | DONNÉES_CHIFFRÉES

- L'adresse IP et le Port : Ce sont les coordonnées du prochain "arrêt" (le saut suivant). Ces informations sont en texte clair pour que le routeur sache où envoyer le reste.

- Les données chiffrées : C'est le reste de l'oignon. Pour le routeur actuel, ce n'est qu'une suite de caractères incompréhensibles.

B. La logique d'empilement* (Encapsulation) :
L'oignon est construit par le client de l'intérieur vers l'extérieur. C'est ce qu'on appelle l'encapsulation*.

1) Le cœur : Le message final (ex: "Salut B") est chiffré avec la clé du destinataire.
2) La couche intermédiaire : On ajoute l'adresse du destinataire devant ce bloc, puis on chiffre le tout avec la clé du dernier routeur.
3) La couche extérieure : On ajoute l'adresse du dernier routeur devant le bloc précédent, et on chiffre l'ensemble avec la clé du premier routeur.

C. Le mécanisme d'épluchage :

Lorsqu'un nœud hybride reçoit cette chaîne de caractères, il effectue les opérations suivantes :

1) Découpage : Il utilise la fonction .split('|')* pour séparer les instructions de routage du contenu chiffré.
2) Déchiffrement : Il utilise sa clé privée* RSA* pour déchiffrer la "charge utile" (le payload*).
3) Transmission : Une fois déchiffrée, cette charge laisse apparaître une nouvelle adresse IP et un nouveau port... qui correspondent au routeur suivant. Le routeur n'a plus qu'à renvoyer le morceau restant.

3.3. Cycle de vie d'un message : De la création à la livraison finale

le parcours d'un message dans mon réseau peut être comparé à une course de relais sécurisée où chaque coureur ne connaît que son prédécesseur et son successeur :

Étape 1 : La consultation de l'annuaire (Client => Master). Avant d'envoyer quoi que ce soit, le Client A doit savoir qui est connecte. Il envoie une requête UDP au Master. Le Master interroge sa base de données MariaDB et lui renvoie la liste des nœuds hybrides disponibles (IP, Port et Clé publique).

Étape 2 : La fabrication de l'oignon (Encapsulation). Le client prépare ensuite le message de manière isolée sur son ordinateur. Il choisit un chemin au hasard parmi les routeurs reçus. Il construit l'oignon "de l'intérieur vers l'extérieur" : il chiffre le message pour le dernier destinataire, puis enferme ce bloc dans une nouvelle couche pour le routeur précédent, en utilisant notre protocole de séparation par délimiteurs.

Étape 3 : Le premier saut* (Client => Routeur 1). Le client ouvre un Socket TCP vers le premier routeur de son circuit* et lui envoie la chaîne de caractères brute*. Le client a maintenant terminé son travail ; son identité est déjà protégée car il ne parle qu'au premier maillon*.

Étape 4 : Le relais et l'épluchage (Le travail des Routeurs). Le message arrive au premier routeur. Grâce au Multi-threading, le routeur peut traiter ce message tout en restant prêt à en recevoir d'autres.

1) Réception : Le routeur reçoit le paquet.
2) Déchiffrement : Il utilise sa Clé privée pour retirer la couche* extérieure.
3) Lecture des instructions : Il découvre, grâce au délimiteur, l'adresse IP et le Port du prochain nœud.
4) Transmission : Il renvoie le reste de l'oignon (toujours chiffré pour lui) au nœud suivant.

Étape 5 : La livraison finale (Routeur final => Client B). Le dernier routeur retire l'ultime couche. Il ne reste alors que le message original en clair et l'adresse du destinataire final (Client B). Le message est délivré.

- Résultat : Le Client B reçoit le message, mais il est incapable de savoir si le message vient du Client A ou s'il a été créé par l'un des routeurs du circuit. L'anonymat est total.

4. Sécurité et Cryptographie :

4.1. Implémentation de l'algorithme RSA* :

La sécurité de mon architecture repose sur le module CryptoManager que j'ai développé. Ce module gère l'intégralité du cycle de vie des clés et des messages sans dépendre d'une bibliothèque tierce pour les calculs de chiffrement et la gestion des clés :

- Pgcd*(a, b) : Utilise l'algorithme d'Euclide pour s'assurer que l'exposant choisi est premier avec $\phi(n)$ une condition indispensable pour que la clé soit valide.

```
def pgcd(a, b):
    while b != 0: a, b = b, a % b
    return a
```

- mod_inverse(e, phi) : Implémente l'algorithme d'Euclide* étendu pour calculer la clé privée d. C'est l'étape la plus complexe mathématiquement : trouver l'inverse modulaire*.

```
def mod_inverse(e, phi):
    t, nouveau_t = 0, 1
    r, nouveau_r = phi, e

    while nouveau_r != 0:
        quotient = r // nouveau_r

        t, nouveau_t = nouveau_t, t - quotient * nouveau_t
        r, nouveau_r = nouveau_r, r - quotient * nouveau_r
    if r > 1: return None
    if t < 0: t = t + phi
    return t
```

- est_premier(num) : Un algorithme qui permet de savoir si un nombre est un nombre premier* qui permet de sélectionner les nombres p et q nécessaires à la

génération du modulo* n.

```
def est_premier(num):
    if num < 2: return False
    if num == 2: return True
    if num % 2 == 0: return False
    for i in range(3, int(num ** 0.5) + 1, 2):
        if num %  i == 0: return False
    return True
```

- generer_paire_cle() : Cette fonction orchestre les outils précédents. Elle choisit les nombres premiers, calcule le modulo et retourne le couple (Clé Publique, Clé Privée).

```
def generer_paire_cle():
    premier = [i for i in range(100, 500) if est_premier(i)]
    p = random.choice(premier)
    q = random.choice(premier)
    while p == q: q = random.choice(premier)

    n = p * q
    phi = (p - 1) * (q - 1)

    e = random.randrange(1, phi)
    g = pgcd(e, phi)
    while g != 1:
        e = random.randrange(1, phi)
        g = pgcd(e, phi)

    d = mod_inverse(e, phi)
    return ((e, n), (d, n))
```

4.2. La classe CryptoManager : Gestion des clés :

La classe CryptoManager est le cerveau de la sécurité du projet. Elle automatise la gestion des clés pour chaque utilisateur ou routeur sans qu'ils aient à manipuler les calculs manuellement :

- Persistance* et Chargement (charger_ou_generer) : Le programme vérifie d'abord si les fichiers cle_publique.txt et cle_privee.txt existent. S'ils sont présents, il les lit et utilise la fonction .split(',') pour extraire les valeurs numériques :

```
def charger_ou_generer(self):
    if os.path.exists(self.c_pub) and os.path.exists(self.c_priv):
        try:
            print("[Crypto] Chargement des clés...")
            with open(self.c_pub, "r") as f:
                ligne = f.read().strip()
                p = ligne.split(',')
                self.publique = (int(p[0]), int(p[1]))

            with open(self.c_priv, "r") as f:
                ligne = f.read().strip()
                p = ligne.split(',')
                self.privee = (int(p[0]), int(p[1]))
            return
        except:
            print("[Crypto] Erreur lecture. On régénère.")
            return

    print("[Crypto] Génération nouvelle paire RSA...")
    self.publique, self.privee = generer_paire_cle()

    with open(self.c_pub, "w") as f:
        f.write(f"{self.publique[0]},{self.publique[1]}")

    with open(self.c_priv, "w") as f:
        f.write(f"{self.privee[0]},{self.privee[1]}")
```

- Génération Automatique : Si les fichiers sont absents, la classe appelle la fonction de génération pour créer une nouvelle identité de chiffrement et la sauvegarde immédiatement sur le disque pour une utilisation future.
- Exportation* (get_pub_avec_str) : Cette méthode est cruciale pour le Master. Elle permet de transformer la clé publique en une simple chaîne de caractères* (ex: "12345, 67890") afin qu'elle puisse être envoyée sur le réseau et stockée facilement dans la base de données.

```
def get_pub_avec_str(self):
    return f"{self.publique[0]}, {self.publique[1]}"
```

4.3. Transformation et Chiffrement des données :

C'est dans les méthodes* chiffrer et dechiffrer que se produit la "magie" du système. Comme le RSA ne traite que des nombres, j'ai mis en place un cycle de transformation précis :

1) Vers le numérique (ord) : Pour chaque caractère du message, j'utilise la fonction* ord() pour obtenir sa valeur ASCII*.
2) Calcul de puissance (pow) : J'applique ensuite la formule de chiffrement $C = M^e \ (mod\ n)$ en utilisant pow(char, e, n). Ce format est très efficace car il gère « La méthode du reste* »
3) Le format "CSV*" personnalisé : Une fois chiffrés, les nombres sont regroupés dans une liste* puis joints* par des virgules (",".join(nombres)). Cela permet de transporter le message chiffré comme une simple suite de chiffres, évitant ainsi les problèmes de caractères spéciaux illisibles dans les terminaux*.
4) Retour au texte (chr) : Lors du déchiffrement, le processus s'inverse : on découpe la chaîne aux virgules, on déchiffre chaque nombre, et on retrouve la lettre originale grâce à chr().

Méthode chiffrer :

```
def chiffrer(self, message, cle_pub_destinataire=None):
    cle_cible = cle_pub_destinataire if cle_pub_destinataire else self.publique
    e, n = cle_cible

    nombres = []
    for char in message:
        c = pow(ord(char), e, n)
        nombres.append(str(c))
    return ",".join(nombres)
```

Méthode déchiffrer :

```
def dechiffrer(self, message_chiffrer_str):
    if not message_chiffrer_str: return ""
    d, n = self.privee

    message_clair = ""
    try:
        nombres = message_chiffrer_str.split(',')
        for c_str in nombres:
            if c_str.strip():
                c = int(c_str)
                m = pow(c, d, n)
                message_clair += chr(m)
    except:
        return "[Erreur Déchiffrement]"
    return message_clair
```

4.4. Forces et limites du système actuel :

Cette implémentation "fait maison" présente des avantages certains mais aussi des points d'amélioration pour un usage réel :

- Forces :
  - Transparence totale : Contrairement aux bibliothèques professionnelles qui cachent leur fonctionnement, ici, chaque étape de la transformation des données est maîtrisée.
  - Modularité* : La classe est totalement indépendante. Elle peut être copiée-collée dans le script* du Master, le Routeur ou le Client sans aucune modification.
- Limites :
  - Sécurité des fichiers : Actuellement, la clé privée est stockée en texte clair* dans un fichier .txt. Dans un système réel, ce fichier devrait lui-même être chiffré par un mot de passe.
  - Vitesse : Le fait de chiffrer caractère par caractère et de tout transformer en texte (avec des virgules) augmente la taille des messages envoyés sur le réseau.

5. Explication des différente classe et leur main* :

5.1. Explication de la classe Master et de son main :

La classe Master :

Son rôle :

Le Master est le pilier central de l'architecture. Il agit comme un annuaire dynamique* et un gestionnaire de base de données*.

Son rôle est de recenser chaque routeur qui se connecte au réseau et de fournir cette liste aux clients afin qu'ils puissent construire leurs circuits de routage*. Il assure la cohérence du réseau en faisant le lien entre le monde physique et les adresses virtuelles*.

A) Gestion des logs* :

- definir_callback_gui(fonction) : Établit un lien avec l'interface graphique*. Elle permet au Master d'envoyer des notifications à la console visuelle sans être directement être lier au code de l'interface.

```
def definir_callback_gui(fonction):
    global CALLBACK_LOG_GUI
    CALLBACK_LOG_GUI = fonction
```

- journalisation_log(qui, type, message) : C'est la fonction de traçabilité*. Elle date et enregistre chaque événement, l'affiche dans la console et l'écrit dans un fichier .log permanent.

```
def journalisation_log(qui, type_message, message):
    maintenant = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    ligne_log = f"[{maintenant}] [{qui}] [{type_message}] {message}"

    print(ligne_log)

    if CALLBACK_LOG_GUI:
        try:
            CALLBACK_LOG_GUI(ligne_log)
        except: pass

    nom_fichier = f"journal_{qui.lower()}.log"
    try:
        with open(nom_fichier, "a", encoding="utf-8") as f:
            f.write(ligne_log + "\n")
    except Exception as e:
        print(f"Erreur d'écriture log : {e}")
```

B) Le Constructeur* __init__ : Initialisation et Configuration :
La méthode __init__ est le point de départ de la classe Master. Elle ne se contente pas de stocker des variables, elle prépare l'environnement de travail du serveur.

```
class Master:
    def __init__(self, port_tcp, db_host="localhost", db_user="root", db_password="toto", db_database="Routagedb", port_udp=50000):
```

Paramétrage de la Base de Données : Le constructeur crée un dictionnaire* de configuration (self.db_config). Cela permet de centraliser les identifiants* (Hôte, Utilisateur, Mot de passe) pour faciliter les connexions futures à a la base de données MariaDB.

```
        self.db_config = {
            "host": db_host,
            "user": db_user,
            "password": db_password,
            "database": db_database
        }
```

Définition des Canaux de Communication* : On définit ici les deux Ports vitaux pour le système :

- Le Port_TCP : Dédié aux échanges de données lourdes (annuaires, inscriptions).
- Le Port_UDP : Réservé au service de découverte rapide.

```
self.Port_TCP = port_tcp
self.Port_UDP = port_udp
```

Traçabilité Immédiate : Dès sa création, l'objet utilise la fonction de journalisation* pour confirmer que la configuration est chargée. Cela permet de vérifier instantanément si le script a bien lu les bons paramètres.

```
journalisation_log("MASTER", "INIT", f"Configuration chargée. Port TCP: {self.Port_TCP}")
self._vider_bdd()
```

Mise à zéro du système (self._vider_bdd()) : C'est une étape cruciale. Au moment où l'objet Master naît, il nettoie la table de routage. Cela assure que le réseau démarre sur une base "propre", sans restes de sessions précédentes qui pourraient contenir des adresses de routeurs désormais déconnectés.

C) Gestion de la Base de Données (MariaDB) :
Cette série de fonctions permet au Master d'utiliser MariaDB comme un annuaire dynamique. L'objectif est de toujours disposer d'une liste à jour des nœuds disponibles.

A) _get_db_connection() : L'accès sécurisé
C'est la méthode utilitaire qui sert de "clé" pour ouvrir la base de données.
encapsulation : Elle utilise **self.db_config pour injecter* proprement les identifiants.
Robustesse* : Le bloc try...except est vital. Si le service MariaDB est arrêté, le Master ne "plante" pas ; il enregistre l'erreur dans les logs et attend la prochaine tentative :

```
def _get_db_connection(self):
    try:
        return mysql.connector.connect(**self.db_config)
    except mysql.connector.Error as err:
        journalisation_log("MASTER", "ERREUR BDD", f"Échec de connexion : {err}")
        return None
```

B) _vider_bdd() : La remise à zéro (Reset) :

Appelée au démarrage, elle utilise la commande SQL TRUNCATE TABLE :

- Raison technique : Contrairement à un simple DELETE, le TRUNCATE réinitialise également les compteurs d'ID.
- Intégrité* : Cela garantit qu'un routeur déconnecté par erreur lors d'une session précédente ne soit pas proposé aux nouveaux clients.

```python
def _vider_bdd(self):
    conn = self._get_db_connection()
    if conn:
        try:
            cursor = conn.cursor()
            cursor.execute("TRUNCATE TABLE TableRoutage")
            conn.commit()
            journalisation_log("MASTER", "NETTOYAGE", "Base de données vidée au démarrage.")
        except Exception as e:
            journalisation_log("MASTER", "ERREUR BDD", f"Échec du vidage : {e}")
        finally:
            if conn.is_connected(): conn.close()
```

C) enregistrer_ou_mettre_a_jour_routeur(ip, port, cle) :

C'est la fonction la plus importante pour la gestion des nœuds. Elle suit une logique en plusieurs étape :

- Vérification : Elle cherche si un routeur existe déjà avec le même couple IP / Port.
- Mise à jour (UPDATE) : Si le routeur est connu, elle met simplement à jour sa Clé Publique (au cas où il aurait généré une nouvelle paire de clés en redémarrant).
- Insertion (INSERT) : Si c'est un nouveau venu, elle l'inscrit et récupère son lastrowid* pour lui attribuer un numéro d'identifiant unique.

```python
def enregistrer_ou_mettre_a_jour_routeur(self, ip, port, cle):
    conn = self._get_db_connection()
    if not conn: return None
    try:
        cursor = conn.cursor()
        check_query = "SELECT id FROM TableRoutage WHERE ip = %s AND port = %s"
        cursor.execute(check_query, (ip, port))
        resultat = cursor.fetchone()
        nouvelle_id = None

        if resultat:
            routeur_id = resultat[0]
            update_query = "UPDATE TableRoutage SET cle = %s WHERE id = %s"
            cursor.execute(update_query, (cle, routeur_id))
            conn.commit()
            journalisation_log("MASTER", "MISE A JOUR", f"Routeur {ip}:{port} (ID {routeur_id}) mis à jour.")
            nouvelle_id = routeur_id
        else:
            insert_query = "INSERT INTO TableRoutage (ip, port, cle) VALUES (%s, %s, %s)"
            cursor.execute(insert_query, (ip, port, cle))
            conn.commit()
            nouvelle_id = cursor.lastrowid
            journalisation_log("MASTER", "INSCRIPTION", f"Nouveau routeur {ip}:{port} enregistré (ID {nouvelle_id}).")
        return nouvelle_id
    except Exception as e:
        journalisation_log("MASTER", "ERREUR BDD", f"Échec Enregistrement {ip}: {e}")
        return None
    finally:
        if conn.is_connected(): conn.close()
```

D) get_tous_les_routeurs() :

dictionary=True : Dans get_tous_les_routeurs, ce paramètre permet de récupérer les résultats sous forme de dictionnaires Python, ce qui rend le code beaucoup plus lisible pour manipuler les IP et les Clés.

finally : Bloc crucial qui garantit que la connexion à la base de données est systématiquement fermée, évitant ainsi de saturer le serveur MariaDB avec des

connexions "fantômes".

```
def get_tous_les_routeurs(self):
    conn = self._get_db_connection()
    if conn:
        try:
            cursor = conn.cursor(dictionary=True)
            cursor.execute("SELECT * FROM TableRoutage")
            resultats = cursor.fetchall()
            return resultats
        except Exception as e:
            journalisation_log("MASTER", "ERREUR BDD", f"Lecture échouée : {e}")
            return []
        finally:
            if conn.is_connected(): conn.close()
    return []
```

E) compter_routeurs() :

Cette fonction tous simplement permet de compter le nombre de routeur stocker dans la base de données.

```
def compter_routeurs(self):
    conn = self._get_db_connection()
    if conn:
        try:
            cursor = conn.cursor()
            cursor.execute("SELECT COUNT(*) FROM TableRoutage")
            res = cursor.fetchone()[0]
            return str(res)
        except: pass
        finally:
            conn.close()
    return "0"
```

D) Le Traitement des Requêtes : La méthode _handle_client :

Cette méthode est le cœur opérationnel du Master. Elle est exécutée dans un Thread indépendant pour chaque nouvelle connexion, ce qui lui permet de répondre à plusieurs routeurs ou clients simultanément sans attente.

1. Analyse et "Parsing*" du protocole :

   Le Master commence par réceptionner les données brutes via _recevoir_tout et les décode en texte.

```
def _recevoir_tout(self, sock):
    contenu = b""
    sock.settimeout(2.0)
    try:
        while True:
            partie = sock.recv(8192)
            if not partie: break
            contenu += partie
            if len(partie) < 8192:
                break
    except: pass
    return contenu
```

C’est ici que mon protocole "fait maison"
intervient :

- Découpage par délimiteur : J'utilise la fonction split('|') pour isoler les différentes parties du message.
- Identification de l'intention : Le programme analyse le premier mot du message (le "Header*") pour savoir s'il s'agit d'une inscription, d'une demande d'annuaire ou d'une simple statistique.

```
def _handle_client(self, conn, addr):
    journalisation_log("MASTER", "CONNEXION", f"Nouvelle connexion de {addr[0]}")
    try:
        while True:
            data = self._recevoir_tout(conn)

            if not data: break
            message = data.decode('utf-8', errors='ignore').strip()

            if "|" in message and "REQ" not in message and "INSCRIPTION" in message:
                try:
                    parties = message.split('|')
                    if parties[0] == "INSCRIPTION":
                        ip_r = parties[1]
                        port_r = int(parties[2])
                        cle_r = parties[3]
```

2. La logique d'inscription (Le dialogue ACK*/NACK*) :
   Lorsqu'un message commence par INSCRIPTION, le Master extrait l'IP, le Port et la Clé Publique du routeur :
   - Mise à jour BDD : Il sollicite la fonction de base de données vue précédemment.
   - Confirmation (ACK) : Si l'inscription réussit, le Master renvoie un message de succès personnalisé : ACK|ID_DU_ROUTEUR. Ce message "Accusé de Réception" (Acknowledge) permet au routeur de confirmer qu'il est bien enregistré. En cas d'échec, il renvoie NACK (Negative Acknowledge).

```
            if nouvelle_id:
                conn.sendall(f"ACK|{nouvelle_id}".encode())
                time.sleep(0.1)
            else:
                conn.sendall(b"NACK")
    except Exception as e:
        journalisation_log("MASTER", "ERREUR", f"Format invalide pour {addr[0]}: {e}")
        conn.sendall(b"Erreur de format")
```

3. Diffusion de l'Annuaire (L'état du réseau) :
   Si un client demande la liste des clés (REQ_LIST_KEYS), le Master réalise une opération de formatage* :
   - Il transforme les lignes de la base de données MariaDB en une longue chaîne de caractères structurée : « ID:1;IP:192.168.1.1;PORT:5000;KEY:xxxx. »
   - Cette structure permet au client de "reconstruire" facilement l'annuaire de son côté pour choisir son circuit de routage.

```
            elif message == "REQ_LIST_KEYS" or message == "ANNUAIRE|GET":
                journalisation_log("MASTER", "ANNUAIRE", f"Envoi de l'annuaire complet à {addr[0]}")
                conn_bdd = self._get_db_connection()
                if conn_bdd:
                    cursor = conn_bdd.cursor()
                    cursor.execute("SELECT id, ip, port, cle FROM TableRoutage")
                    res = cursor.fetchall()
                    conn_bdd.close()
                    items = []
                    for r in res:
                        rid, rip, rport, rcle = r[0], r[1], r[2], r[3]
                        items.append(f"ID:{rid};IP:{rip};PORT:{rport};KEY:{rcle}")
                    reponse = "\n".join(items)
                    conn.sendall(reponse.encode())
                    time.sleep(0.2)

            # Demande Nombre
            elif message == "REQ_NB_ROUTEURS":
                nb = self.compter_routeurs()
                conn.sendall(nb.encode())
                time.sleep(0.1)

    except Exception as e:
        journalisation_log("MASTER", "ERREUR", f"Erreur Client {addr}: {e}")
    finally:
        conn.close()
```

4. Robustesse et sécurité du flux :
   - Gestion des erreurs : Tout le traitement est enfermé dans un bloc* try...except. Si un message arrive corrompu ou mal formé, le Master ne "crashe" pas ; il journalise l'erreur et ferme proprement la connexion.
   - Fermeture systématique (finally) : Quoi qu'il arrive (succès ou erreur), le bloc finally garantit que le Socket est refermé (conn.close()), libérant ainsi les ressources système pour les futures connexions.

E) Services Réseaux :
Cette dernière partie du code gère l'ouverture des canaux de communication. J'ai utilisé le Multi-threading pour que le Master puisse effectuer deux tâches simultanément :

1. Le Service de Découverte (_lancement_service_decouverte) :
   Ce service utilise le protocole UDP. Contrairement au TCP, l'UDP permet d'envoyer des messages rapides sans établir de connexion permanente :
   - Fonctionnement : Le Master écoute sur le port 50000. Si une machine (Routeur ou Client) envoie la question "Ou_est_le_master?", le Master lui répond immédiatement "Je_suis_le_master".

- Intérêt : Cela permet une configuration Plug & Play*. L'utilisateur n'a pas besoin de taper manuellement l'adresse IP du Master, les scripts se trouvent tout seuls sur le réseau local.

```
def _lancement_service_decouverte(self):
    socketUDP = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    socketUDP.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    try:
        socketUDP.bind(("0.0.0.0", self.Port_UDP))
        journalisation_log("MASTER", "UDP", f"Service découverte sur UDP/{self.Port_UDP}")
        while True:
            try:
                data, addr = socketUDP.recvfrom(1024)
                if data.decode().strip() == "Ou_est_le_master?":
                    socketUDP.sendto(b"Je_suis_le_master", addr)
            except: pass
    except Exception as e:
        journalisation_log("MASTER", "ERREUR", f"[UDP] Erreur : {e}")
    finally:
        socketUDP.close()
```

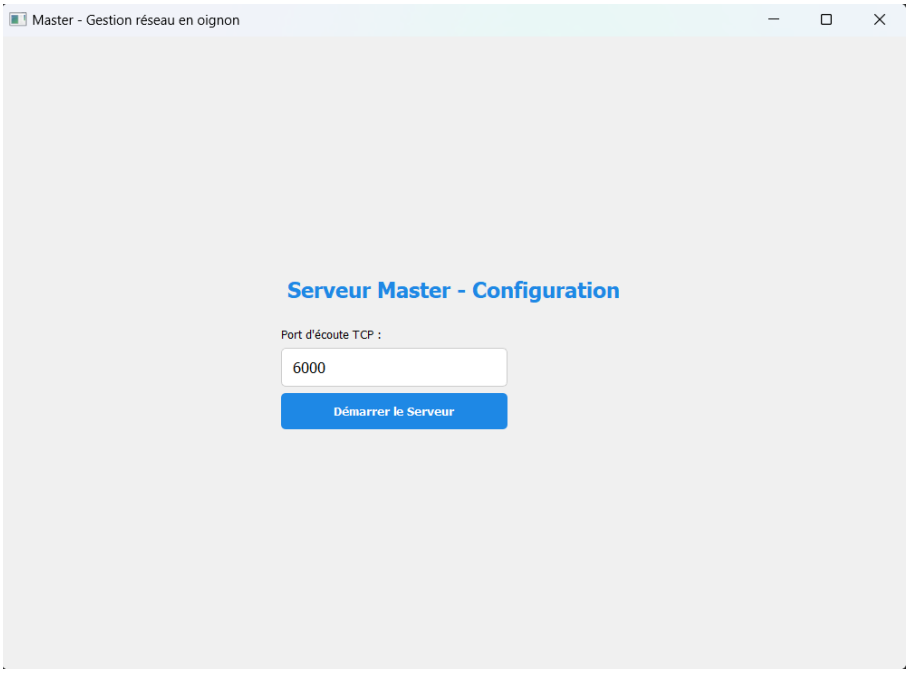
2. Le Serveur Principal (_demarrer_services) :
   Une fois le thread UDP lancé, _demarrer_services se consacre exclusivement à la gestion du protocole TCP, qui est le cœur de mon système d'annuaire :
   - Initialisation : Il crée le socket, le lie à l'adresse IP (bind) et se met en mode écoute (listen(50)). Le chiffre 50 permet de mettre en attente plusieurs demandes de connexion simultanées.
   - La boucle d'acceptation* (while True) : C'est ici que le Master "vit". Il reste bloqué sur socket.accept() jusqu'à ce qu'un Routeur ou un Client se connecte.
   - Le passage de relais : Dès qu'une connexion est acceptée, le Master ne s'occupe pas lui-même de la suite. Il délègue le travail à un nouveau Thread (_handle_client) et retourne instantanément au début de sa boucle pour attendre le prochain arrivant.

```
def _demarrer_services(self):
    journalisation_log("MASTER", "INIT", "Démarrage des threads services...")

    threading.Thread(target=self._lancement_service_decouverte, daemon=True).start()

    socketTCPmaster = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    socketTCPmaster.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    try:
        socketTCPmaster.bind(('0.0.0.0', self.Port_TCP))
        socketTCPmaster.listen(50)
        journalisation_log("MASTER", "TCP", f"Serveur en écoute sur 0.0.0.0:{self.Port_TCP}")

        while True:
            conn, addr = socketTCPmaster.accept()
            threading.Thread(target=self._handle_client, args=(conn, addr), daemon=True).start()

    except Exception as e:
        journalisation_log("MASTER", "CRASH", f"Erreur critique du serveur : {e}")
    finally:
        socketTCPmaster.close()
```

Le main du Master :

Le script principal (main_master.py) utilise la bibliothèque PyQt5. Son rôle est d'encapsuler la logique réseau complexe dans une fenêtre interactive, permettant de configurer et de surveiller le réseau visuellement sans lignes de commande*.

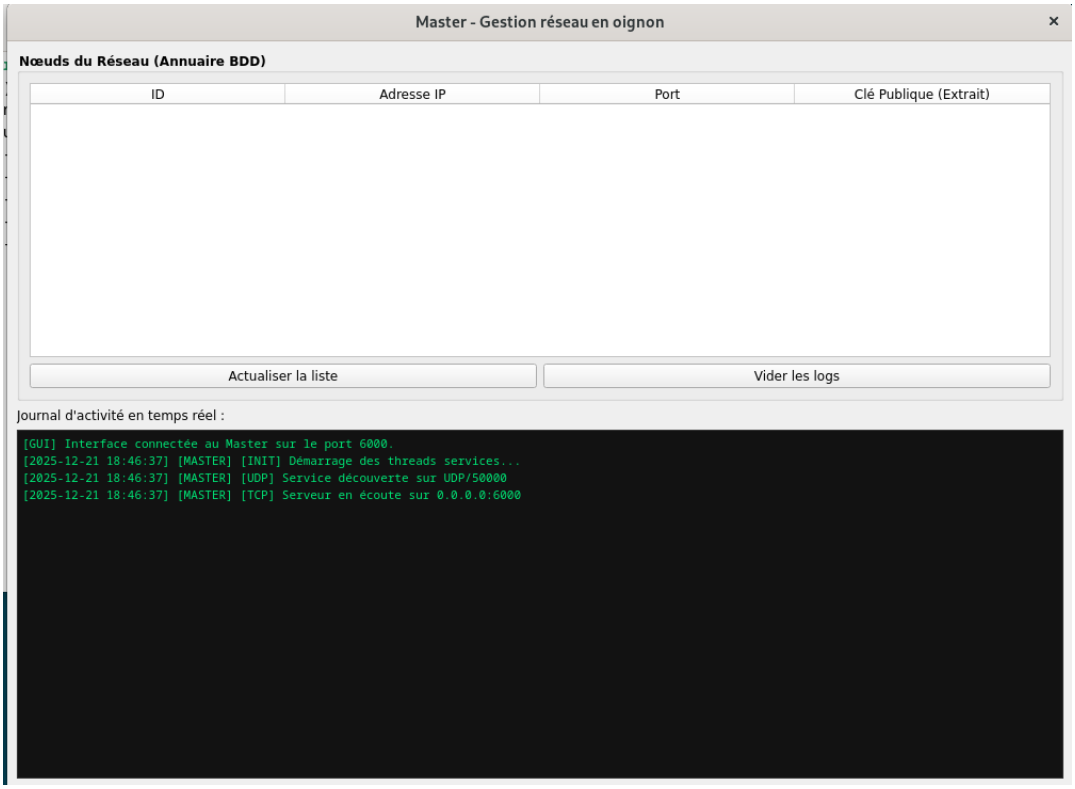
1. Architecture multi-pages (QStackedWidget) :
   J'ai organisé l'application en deux étapes distinctes pour améliorer l'ergonomie* :
   - Page de Configuration (PagePort) : Une interface épurée pour définir le port d'écoute. Elle intègre une validation de saisie (via QMessageBox) pour s'assurer que le port est un nombre valide avant de lancer le serveur.
     interface :



Extrait du code :

```python
class PagePort(QWidget):
    port_valide = pyqtSignal(int)

    def __init__(self):
        super().__init__()
        mise_en_page = QVBoxLayout()
        mise_en_page.setAlignment(Qt.AlignCenter)

        titre = QLabel("Serveur Master - Configuration")
        titre.setStyleSheet("font-size: 24px; font-weight: bold; color: #1E88E5; margin-bottom: 20px;")
        mise_en_page.addWidget(titre)

        self.input_port = QLineEdit("6000")
        self.input_port.setPlaceholderText("Port TCP")
        self.input_port.setFixedWidth(250)
        self.input_port.setStyleSheet("padding: 10px; font-size: 16px; border-radius: 5px; border: 1px solid #CCC;")

        mise_en_page.addWidget(QLabel("Port d'écoute TCP :"))
        mise_en_page.addWidget(self.input_port)

        self.btn_start = QPushButton("Démarrer le Serveur")
        self.btn_start.setFixedWidth(250)
        self.btn_start.setStyleSheet("background-color: #1E88E5; color: white; padding: 12px; font-weight: bold; border-radius: 5px;")
        self.btn_start.setCursor(Qt.PointingHandCursor)
        self.btn_start.clicked.connect(self.valider_configuration)
        mise_en_page.addWidget(self.btn_start)

        self.setLayout(mise_en_page)

    def valider_configuration(self):
        try:
            port = int(self.input_port.text().strip())
            self.port_valide.emit(port)
```

- Tableau de Bord (PageDashboard) : Une fois le serveur lancé, l'utilisateur accède au monitoring*. J'utilise un QStackedWidget pour basculer de l'un à l'autre de manière fluide.



extrait du code :

```python
class PageDashboard(QWidget):
    def __init__(self):
        super().__init__()
        self.master_backend = None

        self.pont = LogBridge()
        self.pont.nouveau_signal_log.connect(self.ajouter_log_ecran)

        self.init_ui()

    def init_ui(self):
        mise_en_page_global = QVBoxLayout()

        # Tableau
        groupe_table = QGroupBox("Nœuds du Réseau (Annuaire BDD)")
        groupe_table.setStyleSheet("QGroupBox { font-weight: bold; }")
        mise_en_page_table = QVBoxLayout()

        self.table = QTableWidget(0, 4)
        self.table.setHorizontalHeaderLabels(["ID", "Adresse IP", "Port", "Clé Publique (Extrait)"])
        self.table.horizontalHeader().setSectionResizeMode(QHeaderView.Stretch)
        self.table.setAlternatingRowColors(True)
        self.table.setStyleSheet("background-color: white; gridline-color: #ccc;")
        mise_en_page_table.addWidget(self.table)

        layout_boutons = QHBoxLayout()
        self.btn_refresh = QPushButton("Actualiser la liste")
        self.btn_refresh.clicked.connect(self.charger_donnees_bdd)

        self.btn_clear = QPushButton("Vider les logs")
        self.btn_clear.clicked.connect(lambda: self.console.clear())
```

2. Le "Pont" de communication (LogBridge) et les Signaux :
   C'est un point technique crucial de mon développement. En PyQt5, on ne peut pas modifier l'interface graphique directement depuis un Thread secondaire (celui du Master) sans risquer un crash.
   - La Solution : J'ai créé une classe LogBridge utilisant les Signals*(pyqtSignal).
   - Le Fonctionnement : Le Master envoie ses logs au "pont", qui émet un signal. L'interface capte ce signal et met à jour la console de manière sécurisée. Cela garantit la stabilité totale du programme malgré le multi-threading.

```
class LogBridge(QObject):
    nouveau_signal_log = pyqtSignal(str)
```

3. Visualisation de la Base de Données (QTableWidget) :
   L'interface rend l'annuaire MariaDB concret et lisible :
   - Extraction dynamique*: La méthode charger_donnees_bdd appelle le backend pour transformer les lignes SQL en lignes de tableau.

```
def charger_donnees_bdd(self):
    if not self.master_backend: return

    try:
        routeurs = self.master_backend.get_tous_les_routeurs()

        self.table.setRowCount(0)
        for row_idx, r in enumerate(routeurs):
            self.table.insertRow(row_idx)

            r_id = str(r['id'])
            r_ip = r['ip']
            r_port = str(r['port'])
            r_cle = str(r['cle'])[:30] + "..."

            self.table.setItem(row_idx, 0, QTableWidgetItem(r_id))
            self.table.setItem(row_idx, 1, QTableWidgetItem(r_ip))
            self.table.setItem(row_idx, 2, QTableWidgetItem(r_port))
            self.table.setItem(row_idx, 3, QTableWidgetItem(r_cle))

    except Exception as e:
        self.ajouter_log_ecran(f"[GUI] Erreur lecture BDD : {e}")
```

   - Formatage : Les informations sont organisées en colonnes (ID, IP, Port). Pour la Clé Publique, qui est très longue, j'ai implémenté un système de tronquage* ([:30] + "...") pour garder une interface propre.

Extrait du code :

```
def init_ui(self):
    mise_en_page_global = QVBoxLayout()

    # Tableau
    groupe_table = QGroupBox("Nœuds du Réseau (Annuaire BDD)")
    groupe_table.setStyleSheet("QGroupBox { font-weight: bold; }")
    mise_en_page_table = QVBoxLayout()

    self.table = QTableWidget(0, 4)
    self.table.setHorizontalHeaderLabels(["ID", "Adresse IP", "Port", "Clé Publique (Extrait)"])
    self.table.horizontalHeader().setSectionResizeMode(QHeaderView.Stretch)
    self.table.setAlternatingRowColors(True)
    self.table.setStyleSheet("background-color: white; gridline-color: #ccc;")
    mise_en_page_table.addWidget(self.table)

    layout_boutons = QHBoxLayout()
    self.btn_refresh = QPushButton("Actualiser la liste")
    self.btn_refresh.clicked.connect(self.charger_donnees_bdd)

    self.btn_clear = QPushButton("Vider les logs")
    self.btn_clear.clicked.connect(lambda: self.console.clear())

    layout_boutons.addWidget(self.btn_refresh)
    layout_boutons.addWidget(self.btn_clear)
    mise_en_page_table.addLayout(layout_boutons)

    groupe_table.setLayout(mise_en_page_table)
    mise_en_page_global.addWidget(groupe_table)
```

4. Le Journal "Console" en temps réel :
   Sous le tableau, une zone de texte (QTextEdit) imite un terminal (fond noir, police monospace verte).
   - Auto-scroll : J'ai ajouté une logique de défilement automatique vers le bas (sb.maximum()) pour que l'utilisateur voit toujours les derniers événements sans avoir à manipuler la souris.
     code :

```
def ajouter_log_ecran(self, texte):
    self.console.append(texte)
    sb = self.console.verticalScrollBar()
    sb.setValue(sb.maximum())
```

5. L'orchestration finale : La classe MasterApp et le bloc Main :
   Cette dernière partie du code est le "chef d'orchestre" de l'application. Elle gère la structure de la fenêtre et le passage de la configuration au fonctionnement réel :
   A. La navigation par pile* (QStackedWidget)
      Pour éviter d'encombrer l'utilisateur avec plusieurs fenêtres, j'ai utilisé un QStackedWidget :
      - Le concept : Imaginez un paquet de cartes. On ne voit que la carte du dessus.
      - Application : Au démarrage, la "carte" affichée est la page de configuration du port. Une fois le port validé, on "glisse" la carte suivante sur le dessus : le tableau de bord. Cela permet une navigation fluide et moderne dans une interface unique.

Code :

```
self.stack = QStackedWidget()
self.page_config = PagePort()
self.page_dashboard = PageDashboard()

self.stack.addWidget(self.page_config)
self.stack.addWidget(self.page_dashboard)

self.page_config.port_valide.connect(self.lancer_dashboard)

self.setCentralWidget(self.stack)
```

B. La méthode lancer_dashboard : Le déclencheur*
C'est le point de bascule du programme. Lorsque le signal de validation du port est reçu :
- Elle change l'index du widget* (setCurrentIndex(1)) pour afficher le monitoring.
- Elle ordonne au backend de démarrer officiellement le serveur avec le port choisi.

```
def lancer_dashboard(self, port):
    self.stack.setCurrentIndex(1)
    self.page_dashboard.demarrer_serveur(port)
```

C. Le bloc d'exécution* (if __name__ == "__main__":) :
- Style "Fusion" : J'ai forcé l'utilisation du style "Fusion" pour garantir que l'interface ait la même apparence professionnelle, que le correcteur l'exécute sur Windows, Mac ou Linux.
- Boucle d'événements : app.exec_() lance la boucle infinie de l'interface graphique. Elle attend que l'utilisateur clique sur un bouton ou qu'un log arrive pour mettre à jour l'affichage.

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    app.setStyle("Fusion")
    fenetre = MasterApp()
    fenetre.show()
    sys.exit(app.exec_())
```

5.2. Explication de la classe Client et de son main :

La classe Client :

Son rôle :

Le Client est l'initiateur de la communication. C'est lui qui possède la "vue d'ensemble" du réseau (grâce au Master) et qui applique les couches de sécurité RSA pour garantir son propre anonymat.

A) Initialisation et Écoute passive* :

Comme le Master, le Client utilise un système de journalisation et de Threads pour rester réactif :

- __init__ : Il initialise sa propre instance de CryptoManager pour les opérations RSA et lance immédiatement un fil d'exécution pour l'écoute.

```
class Client:
    def __init__(self, routeur_ip, routeur_port, port_ecoute_local):
        self.Routeur_IP = routeur_ip
        self.Routeur_Port = routeur_port
        self.Port_en_ecoute = port_ecoute_local
        self.crypto_outils = CryptoManager()
        self.annuaire_cache = {}
```

- _ecouter_message_entrants() : Cette fonction tourne dans un Thread séparé. Elle permet au client de recevoir des messages (confirmations ou réponses) sur son port local* pendant qu'il continue de préparer ses envois.

```
def _ecouter_message_entrants(self):
    socketTCPentrant = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    socketTCPentrant.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    try:
        socketTCPentrant.bind(("0.0.0.0", self.Port_en_ecoute))
        socketTCPentrant.listen(5)
        journalisation_log("CLIENT", "ECOUTE", f"Client prêt à recevoir sur le port {self.Port_en_ecoute}")
        while True:
            try:
                conn, addr = socketTCPentrant.accept()
                data = self._recevoir_tout(conn)
                if data:
                    message = data.decode('utf-8', errors='ignore')
                    journalisation_log("CLIENT", "RECEPTION", f"Message reçu : {message}")
                conn.close()
            except: pass
    except Exception as e:
        journalisation_log("CLIENT", "ERREUR", f"Erreur écoute réception : {e}")
```

B) Gestion de l'Annuaire Réseau :

Pour construire un chemin, le client doit connaître les "portes" disponibles :

- recuperer_annuaire_complet() : Le client se connecte à sa passerelle (le Master) et demande la liste des nœuds.

```
def recuperer_annuaire_complet(self):
    try:
        socketTCPannuaire = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        socketTCPannuaire.settimeout(5.0)
        socketTCPannuaire.connect((self.Routeur_IP, self.Routeur_Port))
        socketTCPannuaire.sendall(b"REQ_LIST_KEYS")
        reponse_bytes = self._recevoir_tout(socketTCPannuaire)
        socketTCPannuaire.close()

        reponse = reponse_bytes.decode("utf-8")
        annuaire = {}
        for ligne in reponse.split('\n'):
            if "ID:" in ligne:
                parties = ligne.replace('|', ';').split(';')
                infos = {p.split(':')[0]: p.split(':')[1] for p in parties if ':' in p}
                k = infos['KEY'].split(',')
                annuaire[infos['ID']] = {
                    'ip': infos['IP'],
                    'port': int(infos['PORT']),
                    'cle': (int(k[0]), int(k[1]))
                }
        self.annuaire_cache = annuaire
        journalisation_log("CLIENT", "INFO", f"{len(annuaire)} nœuds réseau chargés.")
        return annuaire
    except Exception as e:
        journalisation_log("CLIENT", "ERREUR", f"Impossible de récupérer l'annuaire : {e}")
        return {}
```

- Parsing des données : La fonction reçoit une chaîne de caractères brute et utilise des séparateurs (; et :) pour reconstruire un dictionnaire Python (self.annuaire_cache). Elle stocke l'IP, le Port et surtout la Clé Publique de chaque routeur.

C) La fabrication de l'oignon (Le cœur de la sécurité) :
C'est la fonction la plus technique du projet : construire_oignon(). Elle transforme un message simple en un paquet prêt pour le routage en oignon :

1) Couche de sortie* (Exit Node*) : On prend le message final et on y ajoute un en-tête (ex: DEST:FINAL). On chiffre le tout avec la clé publique du dernier routeur du chemin.
2) Couches intermédiaires : On prend ce paquet déjà chiffré et on l'emballe dans une nouvelle instruction : NEXT_IP:x;NEXT_PORT:y | [Paquet précédent].
3) Chiffrement successif : On chiffre cette nouvelle enveloppe avec la clé publique du routeur précédent, et on recommence jusqu'à atteindre le premier saut.
Resultat : On obtient un paquet où chaque routeur ne peut lire que l'adresse de son successeur immédiat*, sans jamais voir le contenu final ou la destination réelle.

```python
def construire_oignon(self, message, chemin_ids, annuaire, mode="CLIENT", ip_c=None, port_c=None):
    id_sortie = chemin_ids[-1]
    cle_sortie = annuaire[id_sortie]['cle']

    if mode == "CLIENT":
        header = f"RELAY:CLIENT;IP:{ip_c};PORT:{port_c}"
    else:
        header = "DEST:FINAL"

    payload = f"{header}|{message}"
    paquet_chiffre = self.crypto_outils.chiffrer(payload, cle_sortie)
    routeurs_intermediaires = list(reversed(chemin_ids[:-1]))
    id_suivant = id_sortie

    for id_actuel in routeurs_intermediaires:
        info_suiv = annuaire[id_suivant]
        cle_actu = annuaire[id_actuel]['cle']
        instruction = f"NEXT_IP:{info_suiv['ip']};NEXT_PORT:{info_suiv['port']}|{paquet_chiffre}"
        paquet_chiffre = self.crypto_outils.chiffrer(instruction, cle_actu)
        id_suivant = id_actuel

    return paquet_chiffre
```

D) Expédition et Routage :
La fonction envoyer_message() orchestre la sélection du chemin :

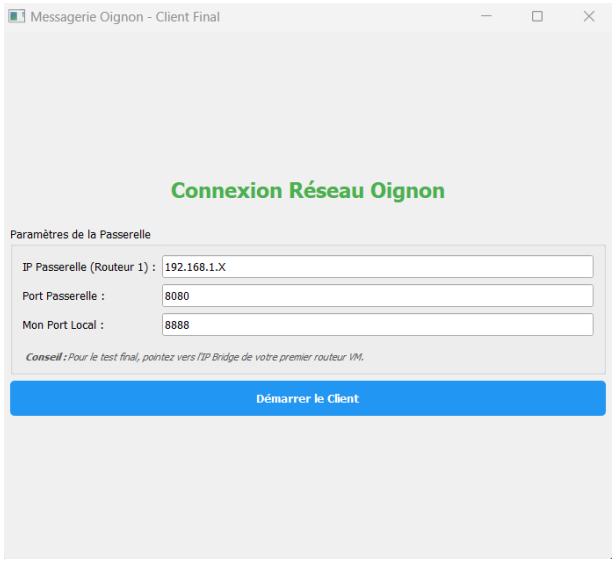
- Sélection aléatoire : Le client choisit au hasard un nombre de nœuds (défini par nb_sauts) parmi ceux disponibles dans son annuaire. Cela garantit que le chemin change à chaque envoi, renforçant l'anonymat.
- interface Réseau (Bridge et Local) : Le code intègre une vérification intelligente de l'adresse IP. Il détermine s'il doit utiliser l'interface réseau locale ou passer par une adresse publique (Bridge) pour atteindre le premier nœud.
- Transmission TCP : Une fois l'oignon construit, le client se connecte au premier nœud de la liste et lui transmet la totalité du paquet chiffré avant de fermer la connexion.

Extrait du code :

```python
def envoyer_message(self, cible, message, nb_sauts):
    self.recuperer_annuaire_complet()
    ip_dest, port_dest = cible
    ids = list(self.annuaire_cache.keys())

    if not ids:
        journalisation_log("CLIENT", "ERREUR", "Annuaire vide. Envoi impossible.")
        return "Erreur"

    id_cible_dans_annuaire = None
    for rid, info in self.annuaire_cache.items():
        if str(info['port']) == str(port_dest):
            id_cible_dans_annuaire = rid
            break

    relais_pour_entree = [i for i in ids if i != id_cible_dans_annuaire]

    if not relais_pour_entree:
        id_entree = id_cible_dans_annuaire
        journalisation_log("CLIENT", "ALERTE", "Un seul nœud dispo : l'entrée sera la cible.")
    else:
        id_entree = random.choice(relais_pour_entree)
```

Le main de client :
L'Interface Graphique (GUI) du Client Final :
Le script main_client.py pilote la classClient via une interface PyQt5. Il transforme des processus cryptographiques complexes en une application de messagerie simple d'utilisation.

1. Configuration et Connexion (PageConfig) :
   Pour que le client puisse rejoindre le réseau, il doit d'abord savoir où se trouve la "porte d'entrée" :
   L'interface :

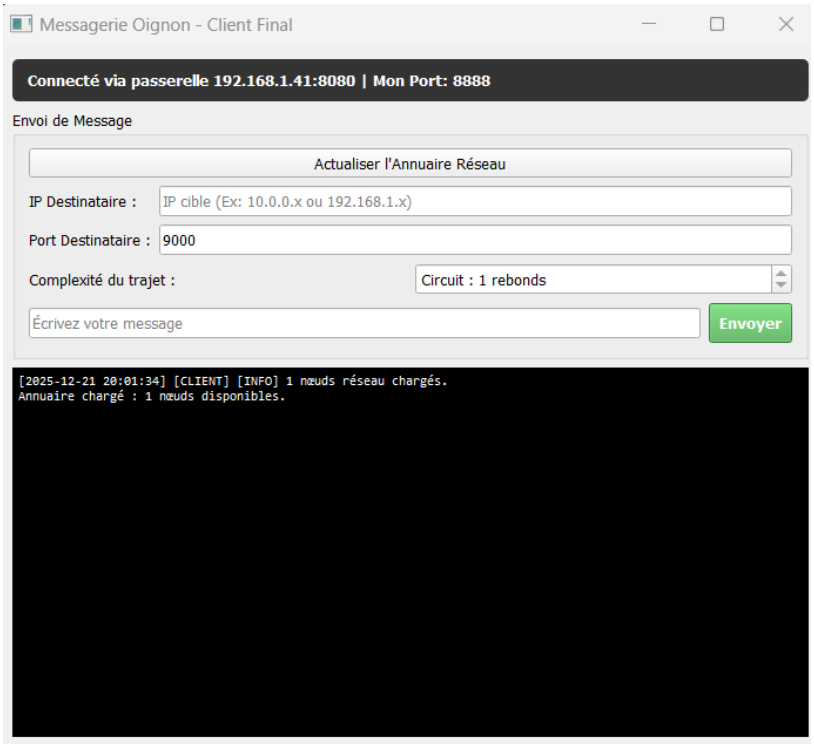


- Paramètres de la Passerelle : L'utilisateur saisit l'IP et le port du premier routeur (le Master/Passerelle).
- Port Local : Le client définit aussi son propre port d'écoute pour recevoir les réponses, ce qui montre que chaque client est aussi un mini-serveur passif.

- Validation : Un signal config_validee est émis uniquement si les données sont cohérentes, déclenchant le passage à l'interface de messagerie.

```
def valider(self):
    ip = self.input_ip.text().strip()
    if not ip: return QMessageBox.warning(self, "Erreur", "L'IP est requise.")
    try:
        pr = int(self.input_pr.text())
        pc = int(self.input_pc.text())
        self.config_validee.emit(ip, pr, pc)
    except ValueError:
        QMessageBox.warning(self, "Erreur", "Ports invalides.")
```

Interface de la page de messagerie :



2. Gestion Dynamique de l'Annuaire :
   L'une des fonctionnalités les plus intelligentes de cette interface est la synchronisation avec le réseau :
   - Actualisation en arrière-plan : La méthode get_annuaire lance un Thread pour interroger le Master. Cela évite que la fenêtre ne se bloque pendant que les données transitent sur le réseau.

```
def get_annuaire(self):
    if self.client_backend:
        threading.Thread(target=self._th_annuaire).start()
```

   - Adaptation du circuit : Le composant QSpinBox (Complexité du trajet) ajuste son maximum automatiquement en fonction du nombre de nœuds réellement disponibles dans l'annuaire. On ne peut pas demander un circuit de 5 rebonds s'il n'y a que 3 routeurs en ligne.

```
l_common = QHBoxLayout()
self.spin_sauts = QSpinBox()
self.spin_sauts.setRange(1, 10)
self.spin_sauts.setPrefix("Circuit : ")
self.spin_sauts.setSuffix(" rebonds")
l_common.addWidget(QLabel("Complexité du trajet :"))
l_common.addWidget(self.spin_sauts)
mise_en_page_groupe.addLayout(l_common)
```

3. Envoi de Message et "Sauts" (Hops) :
   C'est ici que l'utilisateur interagit avec le concept de routage en oignon :
   - Cible et Message : L'utilisateur définit l'IP de sa cible et son message.
   - Nombre de rebonds : L'utilisateur choisit le niveau d'anonymat souhaité. Plus il y a de "sauts", plus le message passera par des routeurs différents.
   - expédition Asynchrone : L'appel à client_backend.envoyer_message est encapsulé dans un thread. Ainsi, l'utilisateur peut continuer à écrire ou consulter ses logs pendant que le client calcule les couches de chiffrement RSA et contacte le premier nœud.

4. Console de Monitoring :
   comme pour le Master, une console noire (QTextEdit) affiche les coulisses du programme :
   - Transparence : L'utilisateur voit précisément quand l'oignon est construit, quel circuit a été choisi (ex: Circuit créé : ['1', '4', '2']), et si le paquet a bien été livré au premier saut.

5.3. Explication de la classe Routeur et de son main :

Son rôle :
Le programme python, Routeur représente l'unité fondamentale du réseau. Contrairement au Master qui est unique, les routeurs sont destinés à être multipliés pour densifier le réseau. Sa conception est hybride : il possède à la fois les capacités de réception d'un serveur et les facultés d'émission d'un client.

A) Architecture Hybride et Réutilisation de Code :
   les fonctions suivantes ne seront pas redétaillées ici car elles sont identiques à la classe Client :
   - construire_oignon() et envoyer_message() : Permettent au routeur d'initier une communication.
   - recuperer_annuaire_complet() : Permet au routeur de connaître ses pairs.
   - journalisation_log() : Assure la traçabilité.

   Info supplémentaire : Cette architecture hybride permet à n'importe quel routeur de devenir un "point d'entrée*" ou un "nœud de sortie*" de manière dynamique.

B) La Spécificité du Routeur : Le rôle de Relais
   Ce qui transforme cette classe en un véritable routeur, c'est sa capacité à traiter les paquets qu'il reçoit pour le compte d'autre :
   1. Réception passive : Le routeur maintient un socket TCP ouvert via _ecouter_message_entrants. Contrairement au client final qui ne fait qu'afficher

le message reçu, le routeur doit analyser le contenu.
Extrait de la fonction :

```python
def _module_ecoute_reseau(self):
    socketTCP_Routeur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    socketTCP_Routeur.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    try:
        socketTCP_Routeur.bind(("0.0.0.0", self.port_local))
        socketTCP_Routeur.listen(5)
        journalisation_log(self.nom_log, "ECOUTE", f"Prêt à relayer sur le port {self.port_local}")
    except Exception as e:
        journalisation_log(self.nom_log, "FATAL", f"Impossible d'ouvrir le port d'écoute : {e}")
        return

    while True:
        try:
            conn, addr = socketTCP_Routeur.accept()
            donnees = self._recevoir_tout(conn)

            if donnees:
                try:
                    message_str = donnees.decode('utf-8').strip()
                    if "REQ_LIST_KEYS" in message_str:
                        lignes = []
                        for rid, info in self.annuaire.items():
                            k = info['cle']
                            lignes.append(f"ID:{rid};IP:{info['ip']};PORT:{info['port']};KEY:{k[0]},{k[1]}")
                        conn.sendall("\n".join(lignes).encode('utf-8'))
                        time.sleep(0.1)
                        conn.close()
                        continue
                except: pass

                threading.Thread(target=self._analyser_paquet, args=(donnees,)).start()

            conn.close()
        except Exception as e:
```

2. L'Épluchage (Peeling) : Lorsqu'un paquet chiffré arrive, le routeur utilise sa Clé Privée (gérée par le CryptoManager) pour tenter de déchiffrer la première couche.
3. L'Instruction de Routage : Une fois déchiffré, le routeur découvre un en-tête caché. Deux cas se présentent alors :
    - Cas NEXT_IP : Le message contient l'adresse du prochain saut. Le routeur se connecte alors à ce voisin et lui transmet le reste de l'oignon.
    - Cas DEST:FINAL : Le routeur réalise qu'il est le dernier maillon. Il livre alors le message au destinataire final (ou l'affiche s'il est lui-même la destination).

```python
def _analyser_paquet(self, donnees_chiffrees):
    try:
        message_str = donnees_chiffrees.decode('utf-8', errors='ignore').strip()
        message_clair = self.crypto.dechiffrer(message_str)

        if not message_clair:
            print(f"[{self.nom_log}] [DEBUG] ❌ Échec déchiffrement (Clé incorrecte ou paquet altéré)")
            return

        if "|" not in message_clair: return

        commande, reste_du_paquet = message_clair.split("|", 1)
        journalisation_log(self.nom_log, "CRYPTO", "Couche d'oignon retirée avec succès.")

        if "NEXT_IP" in commande:
            infos = self._parser_headers(commande)
            journalisation_log(self.nom_log, "ROUTAGE", f"Relayage vers -> {infos['NEXT_IP']}:{infos['NEXT_PORT']}")
            self._envoyer_socket(infos['NEXT_IP'], int(infos['NEXT_PORT']), reste_du_paquet)

        elif "RELAY:CLIENT" in commande:
            infos = self._parser_headers(commande)
            dest_ip = infos['IP']
            dest_port = int(infos['PORT'])

            if dest_port == self.port_local:
                journalisation_log(self.nom_log, "ARRIVÉ", f"Message reçu pour moi : {reste_du_paquet}")
            else:
                journalisation_log(self.nom_log, "SORTIE", f"Livraison finale à {dest_ip}:{dest_port}")
                self._envoyer_socket(dest_ip, dest_port, reste_du_paquet)

        elif "DEST:FINAL" in commande:
            journalisation_log(self.nom_log, "ARRIVEE", f"Message final reçu : {reste_du_paquet}")

    except Exception as e:
        journalisation_log(self.nom_log, "ERREUR", f"Analyse paquet : {e}")
```

Le main du routeur :
Le fichier main_routeur.py orchestre la double nature du nœud (Relais et Client). Son exécution en ligne de commande permet une gestion souple des ressources et une compatibilité maximale avec les environnements de serveurs (VM).

1. Lancement dynamique via les arguments (sys.argv) :
   Pour faciliter les tests, le script utilise les arguments système.
   - Fonctionnement : En lançant la commande python3 script_routeur.py 8080, l'utilisateur définit instantanément le port d'écoute.
   - Avantage : Cela permet de lancer 5 ou 10 routeurs différents sur la même machine physique, simplement en changeant le numéro de port à chaque lancement.

```python
def main():
    if len(sys.argv) < 2:
        print("\n[!] Port manquant. Usage: python3 script_routeur.py <PORT>")
        sys.exit(1)

    try:
        port_local = int(sys.argv[1])
    except ValueError:
        print("\n[!] Le port doit être un nombre entier.")
        sys.exit(1)

    nom_log = f"ROUTEUR_{port_local}"
```

2. Le cœur du relais : Un Threading asynchrone :
   Dès son démarrage, le routeur réalise une opération critique :
   - Fil d'exécution séparé : Il lance la méthode mon_routeur.demarrer dans un Thread avec l'option daemon=True.

```python
try:
    mon_routeur = Routeur(port_local, ip_master, port_master)

    journalisation_log(nom_log, "SCRIPT", "Initialisation du thread d'écoute réseau...")
    thread_serveur = threading.Thread(target=mon_routeur.demarrer, daemon=True) # Utilise demarrer() qui lance l'écoute
    thread_serveur.start()
```

   - Conséquence : Le routeur peut "écouter" et "relayer" des paquets chiffrés en tâche de fond (partie serveur), pendant que le menu principal reste affiché à l'écran pour l'utilisateur (partie client).

3. Menu Interactif : La puissance du Nœud Hybride :
   Le while True propose un menu qui illustre parfaitement la nature hybride du nœud :
   - Option 1 (Inscription) : Le routeur s'annonce au Master et synchronise son annuaire. C'est sa phase d'initialisation en tant que membre du réseau.
   - Option 2 (Visualisation) : Permet de vérifier l'état du réseau global (nœuds connus).
   - Option 3 (Expédition) : Ici, le routeur utilise ses fonctions de Client. Il peut créer un oignon et l'injecter dans le réseau vers une cible, prouvant qu'il est capable d'être à la fois le messager et le relais.

```
Menu principal
1. S'inscrire & Sync Annuaire (Auto)
2. Afficher l'Annuaire local
3. Envoyer un message (Client)
0. Quitter

Action >
```

4. Sécurité et Gestion des Erreurs :
   Le script est protégé contre les mauvaises manipulations :
   - Validation des entrées* : Il vérifie que les ports sont des entiers et empêche l'utilisateur de s'envoyer un message à lui-même (boucle infinie).

```
try:
    target_port = int(input("Port de la cible : "))

    if target_port == port_local:
        print(f"\nERREUR : Le port {target_port} est le vôtre !")
        print("Impossible de s'envoyer un message à soi-même.")
        continue
```

   - Arrêt propre : En capturant le KeyboardInterrupt (Ctrl+C) ou le choix 0, le script ferme proprement ses connexions et journalise l'arrêt du service dans les fichiers de logs.

```
except KeyboardInterrupt:
    journalisation_log(nom_log, "STOP", "Interruption par l'utilisateur (Ctrl+C).")
    print("\n[!] Arrêt demandé.")
```

6. Tests et Validation :

Cette section présente les résultats des essais réalisés sur le réseau en utilisant des machines virtuelles (VM) pour simuler des nœuds distincts.

6.1. Scénarios de tests* : 1, 2, 3 et 4 sauts :

L'objectif était de vérifier que le message arrive à destination peu importe la complexité du circuit, et que chaque couche de chiffrement est correctement retirée

- Test 1 (1 saut - Route directe) : Le client chiffre le message pour le routeur de sortie. Succès immédiat. Temps de latence* quasi nul.
- Test 2 (2 sauts) : Ajout d'un relais intermédiaire. Le premier routeur "épluche" la couche externe et transmet au second. Vérification faite : le premier routeur n'a jamais vu le message final.
- Test 3 (3 sauts - Standard Tor*) : Configuration recommandée. Le message transite par trois entités. L'anonymat est optimal.
- test 4 (4 sauts - Test de stress) : Test de la robustesse du code. Le circuit devient long. Le système reste stable, prouvant que ma fonction récursive* (ou itérative*) de construction d'oignon n'a pas de limite théorique de profondeur*.

6.2. Analyse des résultats et performances (Le paradoxe de stabilité Routeur-Client) :

Lors des tests, un phénomène intéressant a été observé, que j'ai nommé le "Paradoxe de stabilité Routeur-Client" :

- Le constat : Plus on ajoute de sauts, plus la sécurité augmente, mais plus le risque de "rupture" d'un socket TCP augmente statistiquement (si un nœud tombe, tout le circuit échoue).
- La performance : La latence n'est pas causée par le chiffrement RSA (très rapide pour de petits messages), mais par l'ouverture successive des sockets TCP.
- Stabilité Hybride : Le fait que les nœuds soient hybrides (Routeur + Client) permet une meilleure résilience. Si un routeur doit lui-même envoyer un message, il utilise les mêmes ressources que pour relayer, évitant ainsi les conflits de ports ou les saturations de mémoire.
- Cependant j'ai constaté qu'il est plus dur de faire un échange de message entre 2 routeurs que entre un routeur et un client, car si un routeur envoie un message a un routeur il n'est pas sûr que le routeur a qui il envoie le reçoivent alors que si un routeur envoie un message a un client le message est sûr d'arriver ces la que intervient le paradoxe.

6.3. Journalisation* (Logs) et traçabilité :

Pour valider le routage, le système de logs (journalisation) a été mon meilleur outil de débogage*.

- Transparence : Chaque action (chiffrement, réception, déchiffrement, saut suivant) est horodatée* et classée par type (INFO, OIGNON, ERREUR).
- Preuve de l'anonymat : En comparant les fichiers journal_client.log et les logs des routeurs, on constate qu'un routeur intermédiaire ne possède aucune trace de l'IP de l'expéditeur originel (si le saut > 1), validant ainsi l'efficacité de l'architecture.
- Audit* : Les logs permettent de reconstruire le trajet d'un paquet après coup pour vérifier que l'algorithme de choix aléatoire* des nœuds du Master fonctionne de manière équitable.

Glossaire :

1. Introduction :

1.1. Présentation du projet :

- Routage en oignon* : Technique de communication anonyme consistant à encapsuler un message dans plusieurs couches de chiffrement. Chaque couche est "épluchée" par un nœud intermédiaire, garantissant que personne, à part l'émetteur, ne connaît le trajet complet du paquet.

- Routeur* : Dans le contexte de ce projet, il s'agit d'un nœud de relais informatique. Sa fonction est de recevoir un paquet chiffré, d'en retirer une couche de protection et de le transmettre à l'étape suivante du circuit.

- Chiffrement asymétrique* : Méthode de cryptographie qui utilise une paire de clés mathématiquement liées : une clé publique (pour chiffrer) et une clé privée (pour déchiffrer). Cela permet d'échanger des données de manière sécurisée sans avoir à partager un secret commun au préalable.

1.2 Le fonctionnement de l'oignon et des nœuds hybrides :

- Nœud hybride* (ou Hybride*) : Concept d'architecture où une même entité logicielle combine les fonctions de client (émetteur/récepteur) et de routeur (relais). Cette polyvalence renforce l'anonymat du réseau en rendant difficile la distinction entre l'auteur d'un message et un simple intermédiaire.

- Relais* : Fonction spécifique d'un nœud consistant à assurer le transit d'un paquet de données. Le relais déchiffre une couche de l'oignon pour obtenir l'instruction de saut suivant, puis transmet le reste du message sans en connaître le contenu final.

- Master* : Programme central faisant office d'annuaire et de coordinateur. Il gère l'inscription des routeurs et distribue la liste des nœuds actifs (IP, ports et clés publiques) aux clients qui souhaitent construire un circuit sécurisé.

2. Architecture Système :

2.1. Architecture réseaux (Schéma) :

- Interconnecter* : Action de relier plusieurs équipements ou sous-réseaux afin de permettre une communication dans les 2 sens. Dans ce projet, il s'agit de faire communiquer l'hôte physique avec les nœuds virtuels.
- Hôte* (ou Hôte physique) : L'ordinateur réel (votre PC) qui exécute le logiciel de virtualisation. Il sert généralement de point d'ancrage pour le Master et de passerelle vers le réseau extérieur.

- Machine Virtuelle (VM)* : Système informatique complet simuler par logiciel. Cela permet de simuler un réseau de plusieurs routeurs indépendants sur une seule machine physique.

- Segment réseau* : Portion logique d'un réseau informatique permettant d'isoler ou de regrouper des flux de données.

- Réseau Bridge* (Pont) : Type de connexion réseau permettant à une machine virtuelle de se comporter comme un appareil physique à part entière sur le réseau local. Elle obtient une adresse IP de la même plage que l'hôte (ex: 192.168.x.x).

- Réseau Intnet* (Interne) : Réseau virtuel privé géré par le logiciel de machine virtuel, permettant uniquement aux machines virtuelles de communiquer entre elles, sans accès direct à l'hôte physique ou à Internet. C'est idéal pour simuler un réseau local "caché".

2.2. Segmentation Réseau et Adressage :

- Segmentation* : Technique consistant à diviser un réseau informatique en plusieurs sous-réseaux (segments) isolés les uns des autres. Cela permet d'augmenter la sécurité en limitant la propagation des données et en contrôlant les accès entre zones (ex: entre le Bridge et l'Intnet).

- Adressage (IP)* : Système permettant d'attribuer une identité numérique unique (adresse IP) à chaque interface d'un appareil sur un réseau. Dans mon projet, l'adressage permet de distinguer les machines sur le segment 192.168.x.x de celles sur le segment 10.0.0.x.

- Interface* : Point de connexion (physique ou virtuel) entre un appareil et un réseau. Un routeur possède ici plusieurs interfaces : une pour écouter le "monde extérieur" (Bridge) et une pour communiquer avec le réseau interne (Intnet).

- Réseau Privé (vs Public)* : Un réseau privé est un espace d'adressage non routable directement sur Internet (comme ton Intnet). Le passage de l'un à l'autre nécessite souvent un équipement faisant office de passerelle ou de routeur, ce qui correspond exactement au rôle de tes nœuds hybrides.

2.3. Les Nœuds Hybrides et la Double Interface :

- Passerelle (Gateway)* : Dispositif (matériel ou logiciel) permettant de relier deux réseaux de natures différentes. Dans mon projet, le routeur sert de passerelle entre le réseau physique de l'hôte (Bridge) et le réseau virtuel isolé (Intnet).

- Routeur hybride* : Concept architectural propre à ce projet où chaque nœud dispose des bibliothèques logicielles nécessaires pour être à la fois un Client (émetteur/récepteur final) et un Routeur (relais intermédiaire). Cela permet une plus grande flexibilité et un meilleur anonymat.

2.4. Flux de Contrôle et Annuaire (Le Master) :

- Flux de Contrôle* : Désigne les échanges de données destinés à la gestion et à la coordination du réseau (comme l'envoi de l'annuaire ou l'inscription d'un routeur), par opposition au flux de données (le message réel). C'est la "signalisation" du système.

- MariaDB* : Système de gestion de base de données (SGBD) relationnel, open-source et performant. Il est utilisé ici par le Master pour stocker de façon structurée et persistante la liste des nœuds actifs.

- Clé publique* : Composant du chiffrement asymétrique utilisé pour "verrouiller" les données. Elle est stockée dans l'annuaire du Master pour que n'importe quel client puisse l'utiliser afin de chiffrer une couche de l'oignon destinée à un routeur précis.

2.5. Cheminement des Messages :

- Nœud* : Terme générique désignant tout équipement ou logiciel connecté au réseau capable de recevoir, traiter ou transmettre des informations. Dans mon architecture, les Clients, les Routeurs et le Master sont tous des nœuds possédant leur propre identité réseau (IP).

- Paquet* : Unité fondamentale de transfert de données sur un réseau informatique. Dans mon projet, le paquet correspond à l'oignon : il contient les données utiles (le message) enveloppées dans des en-têtes de routage et des couches de chiffrement.

3 Protocoles et Communication :

3.1. Gestion des Sockets, des Protocoles et des Threads :

A. Communication et Transport :

- Socket* : Interface logicielle (une "prise") qui permet à deux programmes de communiquer à travers un réseau. Un socket est défini par la combinaison d'une adresse IP et d'un numéro de port.

- Protocole* : Ensemble de règles et de conventions qui définissent comment les données doivent être formatées et transmises sur un réseau pour que tous les nœuds se comprennent.

- Transport* : Dans le modèle réseau, il s'agit de la manière dont les données sont acheminées d'un point A à un point B. Le transport gère soit la fiabilité (TCP), soit la rapidité (UDP).

- Port* : Identifiant numérique (de 0 à 65535) associé à un service spécifique sur une machine. Par exemple, le Master écoute sur un port, et chaque Routeur sur un autre, permettant d'aiguiller les messages vers le bon programme.

- TCP (Transmission Control Protocol)* : Protocole de transport orienté connexion. Il assure que les données arrivent sans erreur et dans l'ordre (envoi d'accusés de réception). Indispensable ici pour que le déchiffrement de l'oignon ne soit pas corrompu.

- UDP (User Datagram Protocol)* : Protocole de transport "sans connexion" et beaucoup plus rapide que le TCP. Il n'y a pas de garantie de livraison, ce qui est parfait pour les notifications légères de présence des nœuds.

B. Gestion de l'Exécution :

- Thread (ou Fil d'exécution)* : Petite unité d'exécution au sein d'un même programme. Contrairement à un processus complet, les threads partagent la même mémoire, ce qui permet de réaliser plusieurs tâches simultanément de manière efficace.

- Opération Bloquante* : Se dit d'une fonction (comme recv pour les sockets) qui suspend l'exécution du programme tant qu'un événement attendu (l'arrivée d'un message) ne s'est pas produit.

- Multi-threading* : Capacité d'un logiciel à exécuter plusieurs threads en parallèle. C'est ce qui permet au nœuds de rester "à l'écoute" du réseau tout en effectuant d'autres calculs ou en gérant l'interface utilisateur.

- Processus* : Instance d'un programme en cours d'exécution par le système d'exploitation. Un processus peut contenir un ou plusieurs threads.

3.2. Format des paquets : Structure de l'oignon :

A. Formatage et Structure des données :

- Concaténation* : Opération consistant à relier plusieurs chaînes de caractères les unes à la suite des autres pour n'en former qu'une seule. C'est la base de mon protocole de communication personnalisé.

- Délimiteur* : Caractère spécial (ici le pipe |) utilisé pour séparer visuellement et logiquement différentes informations au sein d'un même message brut. Il permet au programme de savoir où s'arrête l'adresse IP et où commencent les données.

- Encapsulation* : Procédé consistant à inclure des données dans une "enveloppe" contenant des informations supplémentaires (comme des instructions de routage). Dans mon projet, chaque couche de l'oignon est une encapsulation chiffrée de la couche suivante.

- Empilement* : Logique de construction de l'oignon où le client superpose les couches de chiffrement dans l'ordre inverse de leur ouverture. C'est ce qui garantit que seul le premier routeur peut ouvrir la première couche.

B. Traitement et Cryptographie :

- .split()* : Méthode de programmation (en Python) utilisée pour découper une chaîne de caractères en plusieurs morceaux à chaque fois qu'un délimiteur spécifique est rencontré.

- RSA* : Algorithme de cryptographie asymétrique utilisé pour sécuriser chaque couche de l'oignon. Son nom vient de ses inventeurs (Rivest, Shamir et Adleman).

- Clé privée* : Clé secrète détenue uniquement par le propriétaire d'un nœud. Elle est indispensable pour déchiffrer les données qui ont été verrouillées avec la clé publique correspondante.

- Payload (ou Charge utile)* : Désigne la partie d'un paquet de données qui contient le message réel transmis. Par opposition aux en-têtes (IP/Port), le payload est la partie chiffrée que le routeur doit traiter.

3.3. Cycle de vie d’un message : De la création à la livraison finale :

A. Architecture du Circuit :

- Maillon* : Désigne chaque nœud intermédiaire (routeur) faisant partie du circuit de communication. L'expéditeur n'est en contact direct qu'avec le "premier maillon", ce qui empêche les maillons suivants de connaître l'identité de l'émetteur originel.

- Circuit* : Chemin logique et éphémère composé d'une suite de routeurs choisis aléatoirement par le client pour acheminer un message. La sécurité du système repose sur le fait que le circuit change régulièrement.

- Saut (Hop)* : Action de transmettre le message d'un nœud au suivant. Un circuit à "3 sauts" signifie que le message transite par trois routeurs avant d'atteindre sa destination.

B. Mécanique de l'Oignon :

- Couche (ou Pelure)* : Niveau de chiffrement individuel enveloppant le message. Chaque couche est verrouillée avec la clé publique d'un routeur spécifique du circuit. Le retrait d'une couche est l'opération de déchiffrement effectuée par un relais.

- Chaîne de caractères brute* : Format de donnée non structuré (contrairement au JSON) envoyé via le socket. Elle contient l'ensemble des octets concaténés (IP | Port | Payload) que le routeur doit découper manuellement.

4. Sécurité et Cryptographie :

4.1 Implémentation de l'algorithme RSA :

- Inverse modulaire* : En cryptographie RSA, c'est le nombre d qui permet d'"annuler" l'effet de l'exposant de chiffrement e. Mathématiquement, on cherche d tel que $(e \times d) \equiv 1 \left(mod\ \phi(n)\right)$ C'est la pièce maîtresse qui constitue la clé privée.

- Algorithme d'Euclide (étendu)* : Une méthode mathématique très ancienne utilisée ici pour deux raisons : vérifier que deux nombres n'ont aucun diviseur commun (PGCD) et calculer l'inverse modulaire. C'est cet algorithme qui rend la génération de la clé privée possible.

- Modulo (n)* : C'est le nombre issu du produit de deux grands nombres premiers $(p \times q)$. Il définit la "taille" de la boîte de calcul : tous les

résultats des chiffres et déchiffres sont les restes d'une division par ce modulo.

- PGCD (Plus Grand Commun Diviseur)* : Le plus grand entier qui divise deux nombres sans reste. Pour que la clé RSA soit valide, l'exposant e et la valeur $\phi(n)$ doivent être "premiers entre eux", c'est-à-dire que leur PGCD doit être égal à 1.

- Nombre Premier* : Un nombre entier qui n'est divisible que par 1 et par lui-même. La solidité du RSA repose sur le fait qu'il est très facile de multiplier deux nombres premiers, mais extrêmement difficile pour un ordinateur de retrouver ces deux nombres à partir de leur produit (n).

4.2. La classe CryptoManager : Gestion des clés :

- Persistance* : Capacité des données à être conservées de manière durable, même après l'arrêt du programme. Dans mon projet, elle est assurée par l'écriture des clés RSA dans des fichiers .txt.

- Chaîne de caractères (String)* : Suite de symboles ou de lettres traitée par l'ordinateur. Ici, les clés numériques sont transformées en chaînes de caractères pour pouvoir être transmises via les sockets et stockées dans la base de données MariaDB.

- Exportation* : Action de préparer une donnée interne au programme (la clé publique) pour la rendre disponible à l'extérieur (au Master). Cela permet aux autres nœuds du réseau de récupérer mon "cadenas" pour chiffrer des messages.

4.3. Transformation et Chiffrement des données :

- Méthode / Fonction* : Bloc de code réutilisable qui effectue une tâche spécifique. Dans mon projet, les méthodes chiffrer et dechiffrer encapsulent toute la logique de transformation des données.

- ASCII* : Norme de codage informatique qui attribue un numéro unique à chaque caractère (ex: 'A' = 65). Comme le RSA ne peut traiter que des chiffres, l'ASCII sert de "traducteur" universel entre le texte et le numérique.

- La méthode du reste (Exponentiation modulaire)* : Technique de calcul utilisée par la fonction pow(base, exposant, modulo). Elle permet de calculer des puissances extrêmement grandes sans saturer la mémoire de l'ordinateur, en ne conservant que le reste de la division à chaque étape.

- CSV personnalisé* : Format de données où les valeurs sont séparées par des virgules (Comma-Separated Values). Dans mon projet, cela permet de transformer une liste de grands nombres chiffrés en une seule chaîne de texte facile à transmettre sans corrompre les données.
- Joints (Join)* : Opération de programmation consistant à assembler les éléments d'une liste pour former une seule chaîne de caractères en insérant un séparateur (comme une virgule) entre chaque élément.

- Terminaux* : Interfaces textuelles (comme l'Invite de commande ou le Terminal Linux) utilisées pour afficher les logs et les messages. L'utilisation du format numérique évite que le terminal n'essaie d'afficher des caractères de contrôle "invisibles" qui pourraient faire planter l'affichage.

4.4. Forces et limites du système actuel :

- Modularité* : Propriété d'un système dont les composants (comme ta classe CryptoManager) sont indépendants et interchangeables. Cela permet d'utiliser le même code sur le Master, le Client ou le Routeur sans avoir à le réécrire.

- Script* : Fichier texte contenant une suite d'instructions (en Python ici) destinées à être exécutées par un interpréteur.

- Texte clair (Plaintext)* : Information qui n'est pas chiffrée et donc lisible par n'importe qui.

5. Explication des différente classe et leur main :

5.1. Explication de la classe Master et de son main :

- Main (Fonction principale)* : Le point d'entrée du script. C'est ici que tout commence : le programme initialise l'interface graphique, lance les fils d'exécution (Threads) pour l'écoute réseau et établit la connexion avec la base de données.

- Annuaire dynamique* : Liste vivante des routeurs disponibles à un instant T. Il est dit "dynamique" car il se met à jour automatiquement dès qu'un nouveau routeur s'enregistre ou se déconnecte, garantissant que les clients reçoivent toujours des informations fraîches.

- Gestionnaire de base de données* : Module logiciel (utilisant MariaDB) chargé d'enregistrer et de lire les informations des routeurs. Il transforme les actions du programme en requêtes SQL pour garantir que les données

ne soient pas perdues.

- Circuits de routage* : Chemins logiques formés par une succession de nœuds. Bien que le Master ne crée pas le circuit (c'est le client qui le fait), c'est lui qui fournit les "briques" (les routeurs) nécessaires à sa construction.

- Adresses virtuelles* : Adresses IP utilisées au sein de mon infrastructure de test (comme les IP en 10.0.0.X). Elles permettent de simuler un réseau privé et distinct de ton réseau physique réel.

- Logs (Journalisation)* : Historique chronologique de tous les événements survenus sur le Master (ex: "Connexion du Routeur 1", "Requête du Client A"). C'est la "boîte noire" qui permet de vérifier que tout fonctionne et de comprendre l'origine d'une erreur.

- Interface graphique (GUI)* : Espace visuel composé de fenêtres, de boutons et de champs de texte permettant à l'utilisateur d'interagir avec le programme. Dans ce projet, elle est réalisée avec la bibliothèque PyQt5 pour le Master et le Client.

- Traçabilité* : Capacité à conserver un historique précis et chronologique des événements (via les logs). Elle permet de comprendre le trajet d'une requête et de diagnostiquer les pannes dans le réseau.

- Constructeur (__init__)* : Méthode spéciale au sein d'une classe Python qui s'exécute automatiquement lors de la création d'un objet. Elle sert à configurer les paramètres de base, comme l'ouverture des sockets ou l'initialisation de l'interface graphique.

- Dictionnaire (dict)* : Structure de données Python stockant des informations sous forme de paires "Clé / Valeur". Le Master utilise des dictionnaires pour manipuler en mémoire les informations des routeurs (ex: {'IP': '10.0.0.1', 'Port': 5000}) avant de les envoyer au client.

- Identifiants* : Données uniques (comme le port ou un identifiant en base de données) permettant de distinguer chaque nœud du réseau sans risque de confusion.

- Canaux de Communication* : Désigne les différentes voies par lesquelles les données circulent. Mon projet utilise deux canaux distincts : un pour le contrôle (annuaire via UDP) et un pour l'acheminement des messages (TCP).

- Injecter* : Action d'insérer des données dans un flux ou un système. Par exemple, le client "injecte" un oignon chiffré dans le premier routeur du circuit pour démarrer la transmission.

- Robustesse* : Capacité du système à continuer de fonctionner malgré des erreurs ou des conditions imprévues (ex: un routeur qui se déconnecte brusquement). Elle est assurée par une gestion rigoureuse des exceptions dans mon code.

- Intégrité* : Garantie que les données n'ont pas été modifiées ou corrompues pendant leur transport. Dans mon projet, c'est le protocole TCP qui assure l'intégrité en vérifiant que chaque octet de l'oignon arrive intact.

- lastrowid* : Propriété spécifique utilisée en programmation SQL (avec MariaDB) pour récupérer l'identifiant unique (ID) qui vient d'être généré automatiquement lors de l'insertion d'un nouveau routeur dans la base de données.

- Parsing (Analyse syntaxique)* : Action de découper et d'analyser une chaîne de caractères brute (comme ton oignon chiffré) pour en extraire des informations utiles. Mon utilisation de .split('|') est une forme de parsing manuel.

- Header (En-tête)* : Partie du paquet qui contient les instructions de routage (IP et Port du saut suivant). Chaque couche de l'oignon possède son propre en-tête, qui devient visible une fois la couche précédente "épluchée".

- ACK (Acknowledgment)* : Signal de confirmation renvoyé par un récepteur pour dire à l'émetteur : "Bien reçu, le message est arrivé sans erreur".

- NACK (Negative Acknowledgment)* : Signal d'erreur indiquant qu'un message n'est pas arrivé ou est corrompu, demandant souvent un renvoi de la donnée.

- Formatage* : Action de structurer les données d'une certaine manière avant de les envoyer (par exemple, transformer une liste de nombres en une chaîne de caractères séparée par des virgules).

- Bloc* : Segment de données de taille fixe ou variable traité comme une unité. Par exemple, la "charge utile" (payload) d'un oignon est traitée comme un bloc de données chiffrées.

- Plug & Play* : Concept de facilité d'utilisation où un nouveau routeur peut se connecter au réseau et être immédiatement opérationnel et reconnu par le Master sans configuration manuelle complexe.

- Boucle d'acceptation* : Boucle infinie (while True) située dans le Master ou les Routeurs, qui attend en permanence que de nouvelles connexions

de sockets arrivent pour les accepter et créer un nouveau thread de gestion.

- Lignes de commande (CLI)* : Interface textuelle où l'utilisateur interagit avec le programme en tapant des instructions. Contrairement au Master qui est graphique.

- Ergonomie* : Étude de la conception de l'interface pour qu'elle soit intuitive et facile à utiliser. Dans mon Master, cela se traduit par une visualisation claire des routeurs connectés et des logs en temps réel.

- Monitoring (Surveillance)* : Action de surveiller l'état du système. Le Master agit comme un outil de monitoring, permettant de voir instantanément quel nœud rejoint ou quitte le réseau.

- Signals (Signaux PyQt)* : Mécanisme spécifique à la bibliothèque PyQt5 permettant la communication entre différents composants. C'est indispensable pour que mes Threads (qui écoutent le réseau) puissent mettre à jour l'interface graphique sans la faire planter.

- Extraction dynamique* : Processus consistant à récupérer des informations en temps réel (comme les données d'un routeur dans la base de données) au moment précis où on en a besoin, plutôt que de stocker des informations figées.

- Navigation par pile (QStackedWidget)* : Technique de gestion d'interface où les différentes "pages" de l'application (ex: Écran de connexion vs Tableau de bord) sont empilées les unes sur les autres. On affiche celle du dessus tout en gardant les autres en mémoire.

- Déclencheur (Trigger)* : Événement (clic de souris, réception d'un paquet réseau) qui provoque l'exécution d'une action spécifique ou d'une fonction.

- Widget* : Brique de base d'une interface graphique (Bouton, Label, Zone de texte, Fenêtre). Mon application Master est un assemblage de widgets imbriqués.

- Bloc d'exécution* : Portion de code ou Thread qui réalise une tâche spécifique de manière isolée. Cela permet de segmenter le programme pour que l'interface ne gèle pas pendant qu'un traitement long est en cours.

5.2. Explication de la classe Client et de son main :

- Écoute passive* : État d'un programme (via son socket) qui attend l'arrivée de données sans solliciter activement un serveur. Pour le Client,

c'est indispensable pour rester prêt à recevoir un message à tout moment, même s'il n'est pas en train d'en envoyer.

- Port local* : Numéro de port spécifique sur la machine de l'utilisateur réservé à l'application Client. Il permet au système d'exploitation d'aiguiller les messages arrivant du réseau vers mon logiciel plutôt qu'un autre.

- Couche de sortie* : Dans le contexte de l'oignon, il s'agit de la dernière couche de chiffrement (la plus interne). Elle ne peut être déchiffrée que par le destinataire final, garantissant que même le dernier routeur du circuit ne peut pas lire le contenu du message.

- Node (Nœud)* : Terme anglais souvent utilisé pour désigner n'importe quel point de connexion du réseau (Client, Routeur ou Master). Un "nœud" est une entité capable de traiter et de transmettre des paquets de données.

- Successeur immédiat* : Désigne le nœud situé juste après le nœud actuel dans le circuit de routage. Pour le client émetteur, le successeur immédiat est le premier routeur de la chaîne.

5.3. Explication de la classe Routeur et de son main :

- Point d'entrée (Entry Node)* : Désigne le tout premier routeur d'un circuit choisi par le client. C'est le "sas" par lequel le message quitte la machine de l'utilisateur pour entrer dans l'infrastructure d'anonymisation. Il connaît l'adresse IP de l'expéditeur, mais ignore tout du contenu et de la destination finale.

- Nœud de sortie (Exit Node)* : C'est le dernier routeur du circuit de l'oignon. Son rôle est de retirer l'avant-dernière couche de chiffrement pour découvrir les coordonnées du destinataire final (Client B) et lui délivrer le message. C'est le seul routeur qui "voit" le message en clair avant sa livraison.

- Entrées* : Mécanisme de sécurité et de contrôle des données saisies par l'utilisateur. Mon programme effectue un assainissement des données : il vérifie que les numéros de ports sont bien des entiers et bloque les tentatives d'auto-envoi (un client s'envoyant un message à lui-même), ce qui prévient les boucles infinies susceptibles de saturer le réseau.

6. Tests et Validation :

6.1. Scénarios de tests : 1, 2, 3 et 4 sauts :

- Scénario de test* : Suite d'étapes et de conditions définies à l'avance pour valider le bon fonctionnement d'une fonctionnalité. Mes tests de 1 à 4 sauts sont des scénarios qui valident la robustesse de l'algorithme d'encapsulation.

- Latence* : Temps écoulé entre l'envoi d'un message par le client et sa réception par le destinataire. Dans mon réseau, la latence est principalement due au temps de calcul du déchiffrement RSA sur chaque nœud et au transit réseau.

- Standard Tor* : Référence au réseau The Onion Router. Mon test à 3 sauts suit ce standard, qui est considéré comme l'équilibre parfait entre un anonymat fort et une vitesse de navigation acceptable.

- Fonction récursive* : En programmation, il s'agit d'une fonction qui s'appelle elle-même. C'est une méthode élégante pour construire un oignon : pour ajouter n couches, on ajoute une couche à un oignon qui en a déjà n-1.

- Fonction itérative* : Approche utilisant des boucles (comme for ou while) pour répéter une action un nombre précis de fois. C'est l'alternative à la récursivité pour empiler ou éplucher les couches une par une.

- Profondeur (du circuit)* : Nombre de routeurs (maillons) composant un circuit. Plus la profondeur est élevée, plus il est difficile pour un attaquant de remonter jusqu'à la source, mais plus le message met de temps à arriver.

6.3. Journalisation (Logs) et traçabilité.

- Débogage (Debugging)* : Processus d'identification, d'analyse et de correction des erreurs (bugs) dans un programme. Les logs sont l'outils principal pour comprendre pourquoi un paquet pouvait être bloqué ou mal déchiffré.

- Horodaté (Timestamped)* : Ajout automatique de la date et de l'heure précise à chaque ligne de log. Cela permet de suivre le parcours d'un message à la milliseconde près à travers les différents routeurs du réseau.

- Audit* : Examen approfondi des enregistrements (logs) pour vérifier la conformité, la sécurité ou les performances d'un système. L'audit des logs

permet de prouver mathématiquement et techniquement que l'anonymat est respecté.

- Journalisation (Logging)* : Enregistrement automatique des événements survenant durant l'exécution d'un logiciel dans un fichier texte dédié. C'est la "mémoire" de mon infrastructure.

- Aléatoire (Choix aléatoire)* : Sélection de nœuds sans motif prévisible.