Documentation Technique SAE 3.02

Date: 30/12/2025	Durée conseillée: 1h	Durée du travail: ~60h

Champs d'activité:

CA-1.0: Gestion de l'Administration:
- ***CA-1.0.1:*** *Les documentations et procédures*
- ***CA-1.0.3:*** *L'analyse et la recherche*

CA-1.1: Technologies de l'information:
- ***CA-1.1.2:*** *Télécommunications et réseaux*
- ***CA-1.1.4:*** *Systèmes divers*

CA-1.2: Informatique Appliquée:
- ***CA-1.2.2:*** *Développement logiciel (Web, mobile, desktop)*

CA-1.3: Cybersécurité:
- ***CA-1.3.1:*** *Sécurité informatique et cryptographie*

Définition des savoirs:

DS-2.0 Gestion de l'Administration:
- ***DS-2.0.1:*** *Méthodologie d'apprentissage des procédures et documentation*

DS-2.1 Technologies de l'information:
- ***DS-2.1.2:*** *Appréhension des réseaux et des télécommunications*
- ***DS-2.1.3:*** *Compréhension de l'électronique et des matériels*

DS-2.2 Informatique Appliquée:
- ***DS-2.2.2:*** *Connaissance des langages de programmation (Python, C/C++, SQL, NoSQL)*
- ***DS-2.2.3:*** *Gestion de la donnée (relationnelle et non relationnelle)*

DS-2.3: Cybersécurité:
- **DS-2.3.3:** *Méthodes de protection des systèmes et des données (pare-feu, cryptographie, IAM)*

Niveaux Taxonomique:

N3: Maîtrise de l'outil (je sais l'utiliser, je sais le manipulé, je comprend son principe, je sais l'expérimenté)

Compétences terminales

CT-5.0: Gestion de l'Administration:
CT-5.0.1: Capacité à créer, organiser et maintenir des documents administratifs selon les normes établies.
CT-5.0.3: Maîtrise des méthodologies de recherche et d'analyse pour la collecte d'informations pertinentes.

CT-5.1: Technologies de l'information:
CT-5.1.4: Compétence pour gérer et intégrer des systèmes variés.

CT-5.2: Informatique Appliquée:
CT-5.2.1 Compétence pour concevoir et implémenter des algorithmes efficaces et adaptés à des problématiques complexes.
CT-5.2.2: Capacité à développer des applications robustes et performantes pour les environnements web, mobiles, et bureaux.
CT-5.2.6: Aptitude à manipuler et à interroger des bases de données, relationnelles et non relationnelles.

CT-5.3: Cybersécurité:
CT-5.3.2 Expertise en gestion des accès (IAM) et en cryptographie pour assurer la confidentialité et l'intégrité des informations.

Critère d'évaluation :

CE-4.0: Gestion de l'Administration:
CE-4.0.1: Qualité et rigueur des documents administratifs créés et maintenus.
CE-4.0.3: Autonomie, rigueur et pertinence dans la recherche et l'analyse de l'information.

CE-4.2: Informatique Appliquée:
CE-4.2.2: Qualité et maintenabilité du code source produit.
CE-4.2.3: Compétence dans le développement d'applications (web, mobile, bureau).

CE-4.3: Cybersécurité:
CE-4.3.2 Compétence dans l'application de mesures cryptographiques

Légende:
Rouge = Mot définis dans le **glossaire** ou directement.
Bleu clair = Lien, cliquez dessus pour plus d'information
Gras = Important
Gras Sous-ligné = Très important

TABLE DES MATIERES

1. INTRODUCTION:

1.1 Contexte et Enjeux:

Fait dans le cadre de la SAÉ 3.02, ce projet vise à concevoir une application répartie simulant un réseau d'anonymisation de type "**Tor**" (**The Onion Router**). L'enjeu technique principal est de garantir la confidentialité des échanges et l'anonymat de l'émetteur.

1.2 Fonctions:

Le livrable couvre les fonctionnalités suivantes:

- **Administration**: Gestion centralisée de la topologie via un Serveur Maître (Master).
- **Cryptographie**: Implémentation native (sans librairie tierce de haut niveau) de l'algorithme **RSA**.
- **Communication**: Échanges TCP/IP asynchrones via **Sockets**.
- **Interface Homme-Machine**: interfaces graphiques (GUI) basées sur **PyQt6** pour l'utilisation des interfaces client.

2. ANALYSE METHODOLOGIE ET PROTOCOLAIRE:

2.1 Méthodologie de Développement:

Le développement du Projet suit une approche **évènementielle** et **multithreadée**, le tout à été fait en suivant le principe de la programmation orienté objet.

- **Programmation orienté objet (POO)**: L'utilisation de classes (MasterServer, MasterWindow, RSA, Routeur…) permet une séparation nette des responsabilités (SOC - *Separation of Concerns*)**.** Par exemple, le Master gère la logique métier tandis que la fenêtre gère la présentation. Même principe pour l'application client.
- **Héritage et Spécialisation**:
  - L'extension de **threading.Thread** permet d'encapsuler tout le cycle de vie du serveur (init, run, stop) dans une entité autonome.
  - L'extension de **QMainWindow** offre un accès complet aux signaux et slots de PyQt6.
- **Gestion des Ressources**: L'implémentation de la méthode **stop()** et l'interception de QCloseEvent pour libérer les sockets et fermer les connexions SQL, évitant ainsi les erreurs de type "Port already in use" (Ce qui fut un problème conséquent en premier lieu).

2.2 Architecture Logique:

Le système repose sur une architecture hybride **Centralisée/Distribuée** :

1. **Plan de Contrôle (Centralisé)**: Le **Master** agit comme une autorité de certification et un annuaire. Il détient la liste des routeurs actifs, leurs adresses IP/Ports et leurs clés publiques **(N, E). (Voir 2.5 pour la partie RSA)**
2. **Plan de Données (Distribué)**: Les **Routeurs** forment un maillage. Ils relayent les messages de proche en proche sans inspecter le contenu final.
3. **Agents (Clients)**: Ils sont responsables de la construction du circuit (choix des routes) et de l'encapsulation cryptographique (couches en « Onion »).

2.3 Spécification du Protocole Applicatif:

Le protocole de couche 7 (Application) développé pour ce projet est un protocole texte encodé en **UTF-8**, utilisant le caractère « | » (pipe) comme délimiteur de champs et « ; » pour les listes.

Commande	Structure	Description
Enregistrement routeur	ENREGISTREMENT_ROUTEUR | ID_routeur | ip | port|clé_publique_n | clé_publique_e	Enregistrement du routeur pour l'envoie des données dans la BDD
Dé-enregistrement routeur	DEENREGISTREMENT_ROUTEUR | ID_routeur	Dé-enregistrement pour la synchronisation des données vers la BDD si un routeur disparaît
Enregistrement client	ENREGISTREMENT_CLIENT|nom_hôte	Log qu'un client c'est connecté au réseau
Demande topologique	LISTE_ROUTEURS	Demande l'annuaire complet des routeurs
Réponse topologique	ROUTEURS | ID_ROUTEUR:IP:PORT:N:E ; … ; ID_X:IP:PORT_X:N:E;	La réponse topologique que le master envoie vers les clients.
Aquittement	ACK	Réponse du master sur les phases d'initialisation des composants du système

2.3.1 Cycle de vie d'un enregistrement (Handshake)

1. **Requête du Nœud**: Le routeur envoie ENREGISTREMENT_ROUTEUR|ROUTEUR_ID|IP|PORT|N|E.
2. **Validation & Persistance**: Le Master analyse la trame, vérifie l'intégrité des champs et écrit dans la base MariaDB.
3. **Acquittement**: Le Master renvoie ACK au nœud pour confirmer son intégration au réseau. En cas d'erreur (format invalide), il renvoie un message d'erreur explicite.
4. **Arrêt d'un Nœud**: Quand un routeur s'arrête, il envoie un paquet de dé-enregistrement au master, ce choix à été fait pour mitiger le fait que si les routeurs sont arrêtés normalement, les entrés dans la base de données reste, il fallait donc que le master sache que le routeur est à l'arrêt et que donc il doit enlever son entré de la table « routeurs ».
5. **Acquittement**: Le Master renvoie après cela, un ACK, pour confirmer sa suppression du réseau. Si le routeur n'est pas dans la table alors le master log un message d'erreur et renvoie une erreur « Routeur inconnu ».

2.3.2 Diffusion de la Topologie:

Lorsqu'un client demande la liste des routeurs via LISTE_ROUTEURS, le Master transforme le jeu de données SQL en un flux série:

- **Format de sortie** : ROUTEURS|router_id1:ip1:port1:n1:e1;router_id2:ip2:port2:n2:e2
- **Choix méthodologique** : L'envoi des clés publiques (n et e) permet au client de construire son chiffrement en couche immédiatement sans avoir à requêter chaque routeur individuellement, réduisant ainsi la latence et les échanges réseaux superflus.

2.3.3 Intéraction avec la Couche de données:

La méthodologie de gestion des données repose sur un modèle **stateless (sans état) au niveau du code** mais **stateful (avec état) au niveau SQL (Voir Glossaire).**

- **Atomicité**: Chaque transaction (insertion de log ou enregistrement) ouvre et ferme une connexion (Avec mysql.connector.connect). Bien que cela soit plus coûteux en ressources qu'une connexion permanente, cela garantit une robustesse face aux micro-coupures réseau et évite les "Timeouts" de base de données sur de longues périodes d'inactivité.
- **Nettoyage au Démarrage**: La méthode **init_bdd** exécute un DELETE FROM routeurs. C'est un choix méthodologique que j'ai souhaité faire car je considère que tout routeur non actif au moment du lancement du Master est invalide. Cela évite d'envoyer aux clients des adresses de routeurs "fantômes" qui auraient crashé sans se désenregistrer.

2.4 Trames de Transport (Client <> Routeur <> Routeur):

Le message circulant sur le réseau est une chaîne chiffrée. Une fois déchiffrée par la clé privée d'un routeur, la structure interne apparaît comme suit:

- **Cas Relais (Nœud Intermédiaire)**: IP_SUIVANTE | PORT_SUIVANT | PAYLOAD_ENCORE_CHIFFRÉ
  Le routeur lit l'IP suivante et transmet le payload tel quel.
- **Cas Terminal (Dernier Nœud)** : FINALE | 0
  Indique que la couche suivante contient le message clair. La charge utile associé est: IP_DEST | PORT_DEST | MESSAGE_CLAIR

2.5 Algorithme de Cryptographie (RSA):

L'algorithme RSA est implémenté dans la classe RSA du fichier **Algorithme_de_chiffrage.py**.

- **Génération de clés**: Utilisation de **sympy.randprime** pour obtenir deux grands nombres premiers p et q.
  - Module $n = p \times q$.
  - Exposant public $e = 65537$. (Nombre de Fermat)
  - Exposant privé $d \equiv e^{-1} (\text{mod } (p - 1) \times (q - 1))$.
- **Segmentation (Chunking)**: RSA ne pouvant chiffrer des données plus grandes que le module n, le message est découpé en blocs de taille T = taille_clé ⁄ 8 – 11 octets. (Padding lié à la norme PKCS#1 v1.5, voir si dessous)
  Note: La décision de segmenté les messages chiffrés à été fait après des dizaines de tests de l'algorithme créer et des heures d'analyses:

La structure d'un bloc PKCS#1 v1.5 est la suivante:

00 02	PS	00	M
2	k-3-L		L

Où:

- 00 02: 2 Octets fixes indiquant le chiffrement avec padding.
- PS: Octets de remplissage aléatoire non nuls
- 00: Séparateur d'octet nul
- M: Le message original
- L: La longueur du message
- k: La taille du module (Ce qui fait taille_clé // 8)

Soit k la taille totale du bloc, mais l'espace réel pour le message est égal à k-3 (octets fixes) + 8 octets (minimum selon la RFC 8017 Section 7.2.1)

2.5.1 Analyse de la fonction de chiffrage:

```python
def encrypt(self, message: str, clé_publique: Tuple[int, int]) -> str:
        """
        Retourne une string de nombres séparés par des virgules

        Args:
            message (str): Le message à chiffrer
            clé_publique (Tuple[int, int]): La clé publique (n, e)
        """
        n: int
        e: int
        n, e = clé_publique
        taille_bloque: int = (self.taille_clé // 8) - 11
        entiers_chiffré: list[int] = []
        octets_message = message.encode('utf-8')


        for i in range(0, len(octets_message), taille_bloque):
            bloc: bytes = octets_message[i:i+taille_bloque]
            m_int = int.from_bytes(bloc, 'big')
            c_int = pow(m_int, e, n)
            entiers_chiffré.append(str(c_int))

        return ",".join(entiers_chiffré)
```

La fonction encrypt de la classe RSA extrait les composants de la clé publique des routeurs.

La clé publique est un Tuple composé de deux entier, « n » et « e » ou:
- n est le Module RSA (produit de p × q)
- e est l'exposant publique (2^{16}+1).

La taille du bloque, k est calculé à partir de la taille de la clé RSA (Par défaut 1024 bits) divisé par la division entière avec 8 (Soit 128 octets pour 1024 bits) – 11 qui est l'espace réservé pour le padding PKCS#1 v1.5.
Soit 117 octets maximum par bloc.

Une fois la taille des blocs calculé, on convertie le message Unicode en séquence d'octets (Par exemple « Hello » devient « \x48\x65\x6c\x6c\x6f ». Le choix d'utilisé UTF-8 a été fait car il gère tout les caractères Unicode, il est compatible avec ASCII et il ne fait que 1-4 octets par caractères selon le caractère.

« entiers_chiffré » est une liste qui permettra de stocker les nombres chiffrés sous formes de chaînes.

La boucle de traitement des blocs commence au bloc 0 (range(0…), elle va jusqu'à la fin du message (range(0, len(octets_message)…), et avance par blocs de la taille de la valeur de « taille_bloque » en octets.

(Pour un message de 300 octets et une clé de 1024 bits, il y a 3 itérations, (i = 0 (octets 0-116), i = 117 (octets 117-233), i = 234 (octets 234-299).

« bloc = octets_message[i:i+taille_bloque] » extrait taille_bloque d'octets depuis la valeur de la variable « i » (Par exemple octets_message[0:117] est égal au 117 premier octets.

« m_int = int.from_bytes(chunk, 'big') » convertie les octets vers des entiers, 'big' représente « big endian » qui est l'ordre des octets (MSB en premier). Soit: b'\x01\x02' → 1 × 256 + 2 = 258

Pour les matheux:
octets = $[b_0, b_1, b_2, ..., b_{k-1}]$
entier = $b_0 \times 256^{k-1} + b_1 \times 256^{k-2} + ... + b_{k-1} \times 256^0$

Le cœur de l'algorithme est la dernière partie, c'est là que le chiffrement fonctionne.

La formule d'encryption RSA est égal à $C \equiv m^e \pmod n$. (Voir Lien RSA pour plus de détail).
Pour cela j'ai donc utilisé la fonction « pow » prenent en argument « m_int » qui est le message sous forme d'entier, l'exposent public « e » et le module RSA « n ».

Techniquement, une autre façon de le faire est `(m_int ** e) % n` en revanche, ceci est bien trop lent car il calcule un nombre énorme en premier lieu, la fonction « pow() » est beaucoup plus efficace car elle n'utilise pas d'entiers intermédiaire dans son calcule, elle utilise l'exponentiation modulaire pour cela donc elle est plus efficace en terme de mémoire.

A la fin on stock le résultat dans « entiers_chiffré » après convrsion en chaîne de caractères (fonction « str(…) »). Nous les stockons en chaîne pour qu'après on puisse les joindres avec des virgules.

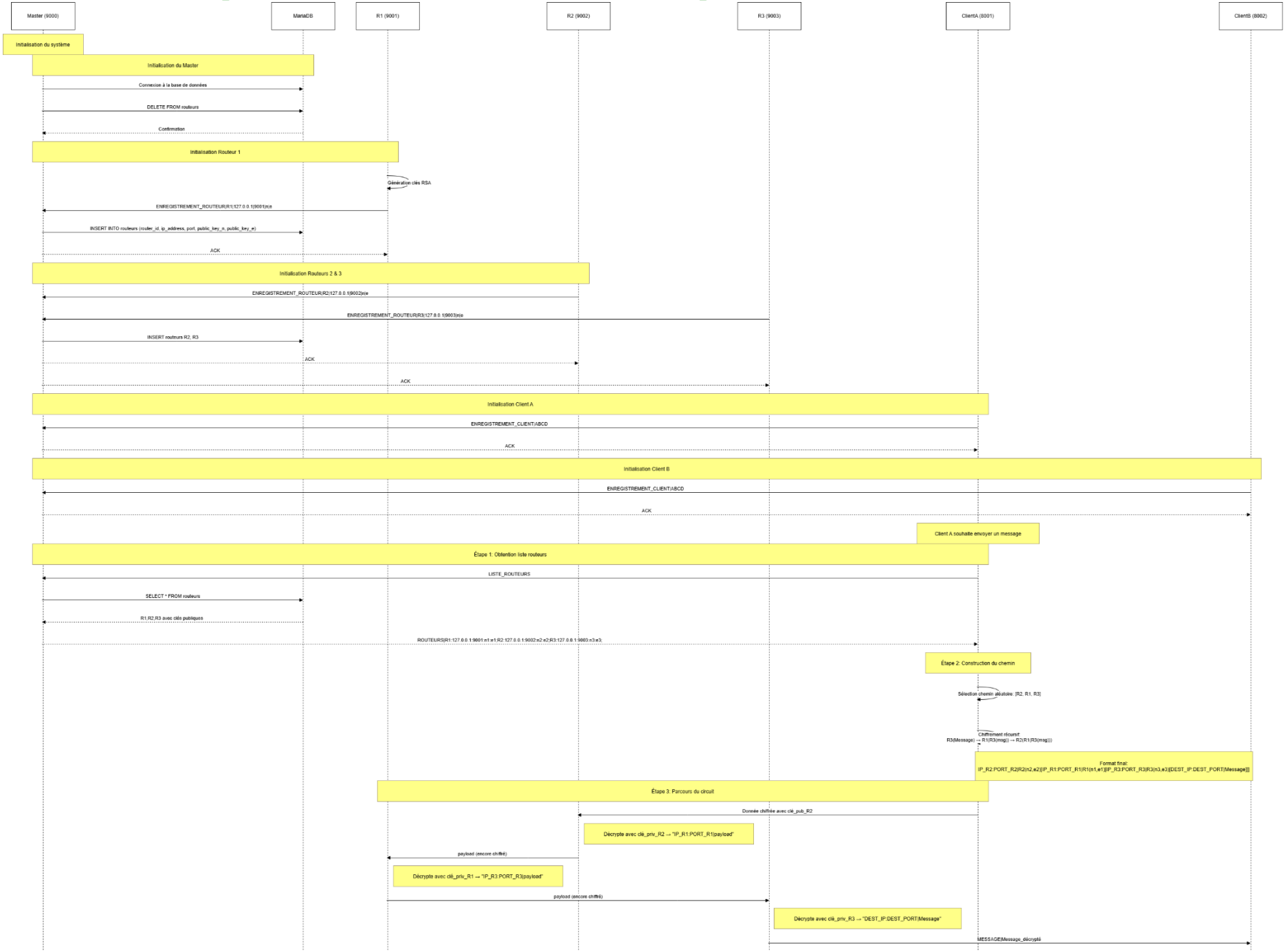
Soit le formattage final devient ['855', '1234', '12'] → « 855,1234,567 »

Mon algorithme RSA à bien sûr des limites, selon mes tests, après 6 itérations (Donc 6 routeurs), les messages prennent beaucoup de temps à s'envoyé.

2.6 Modélisation du chiffrement:

1. **Initialisation:** Le Master démarre et vide la table routeurs de la BDD.
2. **Enregistrement:** Le Routeur R1 démarre, génère (Pub_{R1}, $Priv_{R1}$) et envoie ENREGISTREMENT_ROUTEUR au Master.
3. **Découverte:** Le Client A demandé LISTE_ROUTEURS au Master.
4. **Choix:** Le client A choisi un chemin aléatoire (En utilisant random.samples) en se basant sur le nombre de sauts que l'utilisateur choisie.
5. **Construction de l'onion (Client A):**
   - Couche 3 (Pour R_{Final}): M_3 = Encrypt(Payload, Pub_{Final})
   - Couche 2 (Pour R_{Inter}): M_2 = Encrypt(IP_{Final}|Port|M_3, Pub_{Inter})
   - Couche 1 (Pour $R_{Entrée}$): M_1 = Encrypt(IP_{Inter}|Port|M_2, $Pub_{Entrée}$)
6. **Acheminement:** A→ $R_{Entrée}$ → R_{Inter} → R_{Final} → B

2.7 Schéma Séquentielle d'une communication utilisant ce protocole:



3. ANALYSE TECHNIQUE ET IMPLEMENTATION:

3.1 Serveur Maître (master.py):

3.1.1. Rôle et Architecture:

Le Serveur Maître agit comme l'**Autorité Centrale** du réseau d'anonymisation. Son architecture repose sur un modèle hybride:

- **Back-end Réseau**: Un serveur TCP multi-threadé gérant les requêtes asynchrones.
- **Couche de Données**: Une base de données MariaDB/MySQL pour la persistance de l'annuaire et des traces.
- **Front-end (UI)**: Une interface d'administration temps réel sous PyQt6.

3.1.2. Analyse des Composants Logiciels

3.1.2.1 Gestion de la Configuration (chargement_conf_bdd):

Le serveur utilise un mécanisme d'auto-configuration par fichier (Configuration/config.conf).

- **Sécurité**: L'isolation des identifiants SQL du code source permet une portabilité sécurisée sans identifiants dans le code.
- **Robustesse**: Une gestion d'exception FileNotFoundError empêche le lancement du serveur dans un état instable.

3.1.2.2 Le Cœur Réseau (MasterServer):

La classe **MasterServer** hérite de **threading.Thread**, permettant au moteur réseau de fonctionner sans bloquer l'interface graphique.

- **Initialisation Critique (init_bdd())**: À chaque lancement, le Master purge la table « **routeurs** ». Cette étape est cruciale pour garantir que la liste des nœuds disponibles est **fraîche** et exempte de routeurs déconnectés lors de sessions précédentes.
- **Mécanisme d'Écoute (run())**:
  - Utilise socket.listen() pour mettre les requêtes en file d'attente.
  - Pour chaque client accepté, un thread "jetable" (threading.Thread) est instancié pour exécuter **gère_client**, permettant un traitement **concurrent** des demandes.
- **Arrêt de Force (stop())**: Le serveur utilise une technique de "Self-Connect". Pour débloquer l'appel **accept()** qui est bloquant, le serveur se connecte à lui-même sur 127.0.0.1 après avoir passé son flag **en_cours** à False.

  Note: La raison pour cela est que la fonction accept() reste en attente infiniment tant qu'aucun client ne tente de se connecter, sans ce mécanisme, accept() reste actif et donc le serveur ne peut pas fermer son socket ou terminer son thread.

3.1.2.3 Protocole d'Enregistrement et Annuaire (gère_client)

Cette méthode implémente la logique métier du protocole:

Commande	*Action SQL / Logique*
ENREGISTREMENT_ROUTEUR	Insertion du router_id, de l'IP, du port et du couple RSA (n, e) dans la base de donnée.
DEENREGISTREMENT_ROUTEUR	Suppression par router_id. Utilisé pour l'arrêt propre des nœuds.
LISTE_ROUTEURS	Extraction complète de la table routeurs et sérialisation en format ID:IP:PORT:N:E;

3.1.2.4 Journalisation (sauvegarde_log):

Le Master assure une double journalisation:

1. **Persistante**: Insertion dans la table logs de la base de données pour audit « post-mortem ».
2. **Volatile**: Appel à l'attribut **log_callback** pour mise à jour de l'UI avec code couleur (Vert pour DB, Rouge pour Error, Bleu pour Enregistrement, violet pour informatif, orange pour les warnings).

3.1.3 Analyse de l'Interface d'Administration (MasterWindow)

L'interface a été conçue selon des principes de design web (style Dark Mode inspiré de Tailwind CSS/Slate).

- **Récupération de l'IP Locale**: La fonction **trouve_ip_local** tente une connexion vers les DNS de Google (8.8.8.8) pour identifier l'interface réseau active, facilitant ainsi le déploiement sur des machines distantes.
- **Fonctionnalités d'administration**:
  - **Exportation**: Une méthode **export_logs** permet d'extraire le journal de session au format .txt.
  - **Gestion des signaux systèmes**: L'intégration de signal.SIGINT assure que si le programme est tué par le terminal, les ressources réseau sont relâchées correctement.

3.1.4. Sécurité et Fiabilité

- **Contre-mesures d'Injection**: Le code vérifie explicitement si cmd fait partie d'une liste blanche avant tout traitement SQL.
- **Options de Socket**: SO_REUSEADDR est activé pour permettre un redémarrage instantané du service sans attendre la libération du port par le système d'exploitation.

3.2 Routeur (router.py):

3.2.1 Rôle et fonctions:

Le routeur est l'unité de base du réseau d'anonymisation. Son rôle est double:

1. **S'enregistrer** auprès de l'autorité centrale (le Master) pour être intégré aux circuits.
2. **Relayer** les paquets de manière opaque en retirant une couche de chiffrement (Il est l'éplucheur de l'Onion).

Le principe fondamental respecté ici est qu'un routeur ne connaît que son **prédécesseur** (celui qui lui donne le paquet) et son **successeur** (celui à qui il transmet), garantissant l'anonymat de la source initiale.

3.2.2 Architecture logicielle:

3.2.2.1 Cryptographie Asymétrique (RSA):

Chaque instance de Routeur génère son propre couple de clés RSA au démarrage via l'attribut cipher et la fonction **generate_keys()**:

- **Clé Publique**: Envoyée au Master lors de l'enregistrement. Elle est utilisée par les clients pour chiffrer la couche spécifique à ce routeur.
- **Clé Privée**: Conservée localement en mémoire pour déchiffrer les paquets reçus. JAMAIS TRANSMIS.

Cette approche garantit le principe **Perfect Forward Secrecy** (Confidentialité persistante) à l'échelle de la session: si un routeur est compromis, seules les données de la session actuelle sont vulnérables.

3.2.2.2 Analyse du Traitement des Paquets (gestionnaire_paquet(...)):

C'est ici que réside la logique de routage en onion. Le traitement suit un flux strict:

1. **Réception**: Le routeur reçoit un bloc de données chiffrées.

2. **Déchiffrement**: Utilisation de la clé privée locale. Le résultat est une chaîne formatée: IP_SUIVANTE|PORT_SUIVANT|PAYLOAD.
3. **Analyse de la destination**:
    - Si prochaine_ip == "FINALE": Le routeur identifie qu'il est le **nœud de sortie**. Il extrait le message clair et l'adresse de destination finale.
    - Sinon: Il identifie le prochain saut et transmet la charge utile (qui est elle-même une couche chiffrée pour le routeur suivant).

3.2.2.3 Gestion de la Concurrence:

Le routeur utilise un modèle **Multi-threaded asynchrone**:

- Le thread principal (**while self.en_cours**) accepte les connexions.
- Chaque paquet est traité dans un thread dédié (daemon=True), permettant au routeur de gérer plusieurs flux simultanés sans latence.
- **Auto-nettoyage**: Une routine de filtrage (**threads_vivants**) est appliquée à la liste **threads_actifs** pour éviter les fuites de mémoire.

3.3 Client (client.py)

3.3.1 Rôle et fonctions:

Le client constitue l'interface utilisateur et le point d'entrée du réseau d'anonymisation. Ses deux responsabilités critiques sont:

- **La Construction du Circuit**: Sélectionner des nœuds aléatoirement et encapsule le message dans plusieurs couches de chiffrement.
- **La Terminaison du Flux**: Agit comme serveur local pour recevoir et afficher les réponses provenant du réseau.

3.3.2 Architecture logicielle:

3.3.2.1 Multi-threading et Communication Asynchrone (La classe ÉcouteClient):

Pour éviter le gel de l'interface graphique (GUI) lors de l'attente de messages, le client utilise une architecture découplée:

- **Classe ÉcouteClient (Thread dédié)**: Hérite de QThread. Il gère un socket en mode écoute (bind/listen/accept) sur le port local du client.
- **Mécanisme de Signalisation**: Utilise pyqtSignal(str) pour transmettre les données reçues du thread réseau vers le thread principal de l'UI. Cela garantit une mise à jour de l'affichage sécurisée. (L'implémentation de ce mécanisme à été une des parties les plus difficiles de l'implémentation du client).

3.3.2.2 Implémentation du Routage en Onion (envoie_message)

C'est le cœur algorithmique du projet. Le processus suit trois étapes clés:

1. **Récupération de la Topologie**: Le client interroge le Master (**recois_routeurs**) pour obtenir la liste des routeurs actifs et leurs **clés publiques RSA**.
2. **Sélection Aléatoire**: Utilisation de random.sample pour choisir un chemin de N sauts (défini par l'utilisateur via la QSpinBox), garantissant qu'un nœud ne peut pas prédire le chemin complet.
3. **Encapsulation (Chiffrement Récursif)** :
    - Le message est préparé de la fin vers le début (LIFO).
    - Chaque couche contient l'adresse du prochain saut et le message déjà chiffré pour le nœud suivant.
    - **Formule mathématique**:

      $$C = E_{K_1}(P_2|E_{K_2}(P_3 \mid E_{K_3}(\text{Destination}|\text{Message})))$$

      Où E est la fonction de chiffrement RSA, K_n la clé publique du routeur n, et P_n l'adresse du saut suivant.

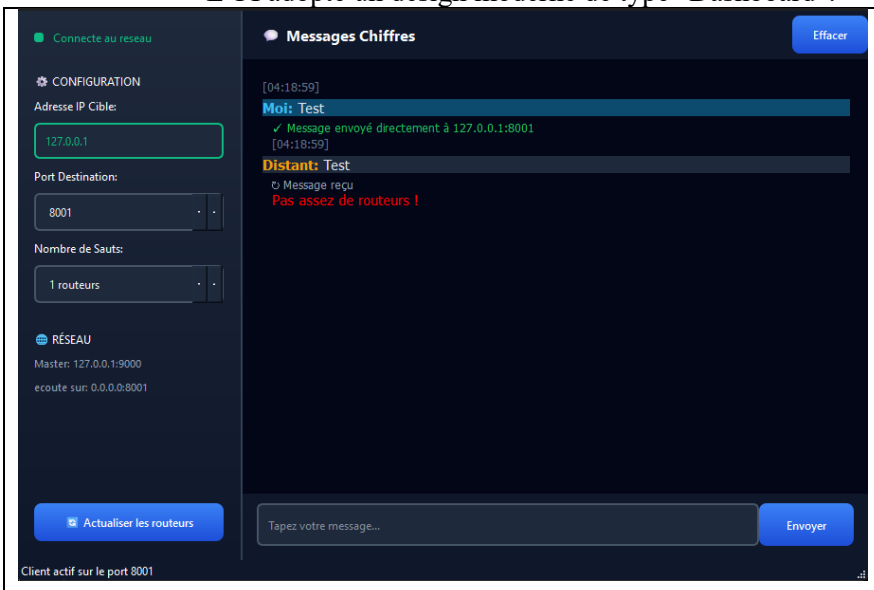
3.3.2.3 Validation et Robustesse (valide_ip)

Le client intègre une validation de l'adresse IP de la destination:

- Utilisation de socket.inet_aton() pour vérifier la conformité structurelle.
- Feedback visuel immédiat via le changement de style CSS (rouge/vert) et l'affichage de messages d'erreur (**label_erreur_ip**), améliorant l'expérience utilisateur.

3.3.3 Interface Graphique (PyQt6):

L'UI adopte un design moderne de type "Dashboard":



Barre Latérale de Configuration: Permet de modifier dynamiquement les paramètres du réseau (nombre de sauts, IP cible, port de destination).

Affichage (HTML): Le chat utilise du rendu HTML pour distinguer visuellement les messages envoyés (**bleu**), reçus (**ambre**) et les notifications système (**vert**/**rouge**).

Gestion des Paramètres de Lancement : Un système de gestion des arguments (**sys.argv**) permet de lancer plusieurs instances de clients sur une même machine en spécifiant des ports différents. Le système prend aussi en compte la gestion de clients sur le même port, pour éviter la mise en écoute, le premier client sur un port, est propriétaire de ce port.

3.4 Persistance des données:

Les données persistantes sont stockées dans la base de données relationnel centralisé MariaDB dans deux tables distinctes:

- La table routeurs(id, router_id, ip_address, port, public_key_n, public_key_e) qui sont des données volatiles mais persistante tant que le master est actif.
- La table logs(id, timestamp, event_type, details) qui est l'historique des évènements qui ne peut être supprimé seulement manuellement depuis la base de données. Ceci est un choix technique car j'estime que les logs sont importants.

4. ENVIRONNEMENT ET CONVENTION:

Le système utilise pour langage de développement Python. Il a été développé en Python 3.10+. Les bibliothèques externes utilisé ont été:

- PyQt6: Pour l'nterface Graphique.
- mysql-connector-python: Driver MySQL utilize pour MariaDB (Qui est basé sur MySQL)
- sympy: Qui est une librairie mathématiques utilisé pour la génération de nombres premiers.

Le système est compatible Windows/Linux (gestion des chemins via os.path).

5. GLOSSAIRE DU PROJET:

- **Architecture découplée**: Principe logiciel visant à autoriser les composants d'un système de faire des actions indépendamment l'un de l'autre.
- **Admininistration**: L'action de géré, manipulé ou utilisé le système.
- **Adresse IP**: Suite de chiffres permettant l'identification unique (Dans son contexte) d'un appareil connecté à un réseau.
- **Bind**: Action d'associer un socket à une adresse IP et un port local.
- **Classe**: En programmation orienté objet, un plan de création d'objet.
- **Clés**: Concept informatique abstrait représentant une valeur utilisée pour le contrôle d'opérations cryptographique (chiffrement, déchiffrement, génération de signature ou vérification). Elles peuvent être asymétriques ou symétrique. Une clé symétrique est utilisée pour les opérations de chiffrement et déchiffrement, alors qu'un pair de clés asymétrique (publique et privé) ont toutes les deux leurs propres opérations.
- **Communication**: Action d'envoyé ou recevoir des données.
- **Cryptographie**: Domaine de la cryptologie, permettant la protection de messages à partir de secrets ou, dans le cas du projet, de clés.
- **Évènementielle**: Architecture utilisant des événements pour déclencher des actions et assurer la communication entre services découplé.
- **GUI**: Graphical User Interface, ou encore, une interface graphique.
- **Héritage**: Principe de la POO permettant de créer une classe à partir d'une classe dites « parent » héritant ainsi des attributs et méthodes de cette classe.
- **Interface Homme-Machine (IHM)**: Terme faisant référence à l'interface graphique que voit l'utilisateur.
- **Multithread**: Capacité d'un processus à gérer son utilisation sur plusieurs demandes sans qu'il soit nécessaire de créer plusieurs instances du programme. (Comme un livreur qui livre 12 colis au lieu que 12 livreurs livrent chacun 1 colis).
- **Native:** Fait référence au niveau d'interaction avec le système. Ici, la création du système a été faite sans véritable bibliothèque de haut niveau, ce qui le rend plus proche du runtime Python.
- **Onion**: Structure de données non primitive constituée de couches de chiffrement successives. Chaque couche ne peut être déchiffrée que par un routeur spécifique.
- **POO**: Programmation orienté objet, un paradigme informatique constant en la création de composants logiciel appelé « objet » (Ex: Une voiture) possédant des attributs et pouvant intéragir avec d'autres composants dans un lien à différent niveau (Héritage, agrégation, composition).
- **Port**: Valeur logique assigné à un processus réseau agissant comme point terminal de communication pouvant allé de 1-65535
- **Routeur**: Appareil permettant le routage de paquets. Dans le contexte du projet, implémenté de façon logiciel et agissant comme relaye de paquet.
- **RSA**: Rivest-Shamir-Adleman est un système de cryptographie asymétrique reposant sur la difficulté de factoriser de grands nombres entiers, c'est un des systèmes de cryptographie les plus utilisé dans les communications sécurisées de données.
- **Signal**: Mécanisme de communication inter-objets utilisé par PyQt. Permet à un thread secondaire de signaler un événement au thread principal.
- **Socket**: Point de terminaison d'un flux de communication réseau.
- **SQL**: Structured Query Language (Langage de requêtes structurées), langage de programmation standardisé pour l'intéraction avec des bases de données relationnelles.
- **Structure de données**: Manière d'organiser les données pour les traiter plus facilement. Elles sont distinguées en deux sortes, les primitives, qui utilise des valeurs unique et immuable (non changeable) et les non primitifs, ceux-ci stockent souvent plusieurs valeurs dans un ou différent format. Une liste est un exemple d'une structure de donnée non primitive.
- **TCP/IP**: Suite de protocoles réseau pour l'intercommunication de données.
- **Thread (Fil d'exécution)**: Unité d'exécution au sein d'un processus. Permet le parallélisme (ex: écouter le réseau tout en gardant l'interface réactive).

6. BIBLIOGRAPHIE ET REFERENCES:

1. **Documentation Python 3 - Socket**: *Low-level networking interface*. Disponible à: lien vers la documentation de socket
2. **Documentation Python 3** – **Threading**: Thread-based parallelism. Disponible à: lien vers la documentation de Threading
3. PyQt6 Documentation as a whole
4. **PyQt6 Documentation**: *Signals & Slots*. Riverbank Computing.
5. **Tor Project**: *Tor Protocol Specification*
6. **RFC 793**: *Transmission Control Protocol (TCP)*.
7. **RFC 2313**: PKCS #1: RSA Encryption
8. **RFC 8017**: PKCS #1: RSA Cryptography Specifications Version 2.2
9. **SymPy Documentation**: *Number Theory*.
10. **MySQL Connector** Guide Python