Programmation Réseau
Rapport EnSkred

KAOUANE Marwane, MAIBECHE Massiouane

Introduction :

Le projet **EnSkred** consiste à développer un protocole de communication sécurisé reposant sur TCP, permettant à des applications identifiées par des clés RSA 2048 bits d'échanger des messages textuels de manière confidentielle et anonyme. L'objectif est de garantir la confidentialité des échanges, même en cas de surveillance du réseau ou de compromission d'une application. Ce rapport décrit la conception du protocole, les choix techniques réalisés et l'implémentation qui en découle.

Etat du projet:

Les fonctionnalités du projet demandé sont toutes opérationnelles.
Pour lancer l'application, on se déplace dans le répertoire du projet et on lance les commandes suivantes :

Première Connexion :

java [path jusqu'au fichier binaire Application] [numéro de port]
java -jar [path jusqu'au fichier jar de l'application] [numéro de port]

ex : java bin/fr/uge/enskred/application/Application 7777
ex : java -jar target/enskred.jar 7777

NB: le fait de saisir un port fait que l'on crée notre propre réseau

Deuxième connexion :

java [path jusqu'au fichier binaire Application/ jar de l'application] [port 1] [port 2]
java -jar [path jusqu'au fichier jar de l'application] [port 1] [port 2]
Le **port 1** étant nous même et le **port 2** étant le client à qui on veut se connecter.

ex : java bin/fr/uge/enskred/application/Application 7778 7777
ex : java -jar target/enskred.jar 7778 7777

Déconnexion :

Pour se déconnecter, il faut lancer la commande **-d**. L'application va alors se déconnecter et déléguer tous ses voisins à un des voisins choisis aléatoirement.

Communication / messagerie ouverte :

Pour la communication ouverte il faut entrer la commande :
-mp [application 2] [Message]

ex : -mp 7778 Salut !!!

Pour distribuer ce message, le plus court chemin allant de l'application qui envoie à l'application receveuse est choisi.

Communication / messagerie fermé :

Pour la communication fermé il faut entrer la commande :
-mc [application 2] [Message]

ex : -mc 7778 Salut !!!

Pour distribuer ce message, un chemin tiré aléatoirement dans le réseau est sélectionné pour envoyer ce dernier, de l'application qui envoie à l'application receveuse.
Le message est alors encodé en onion routing.

Demande de la liste des applications connectés :

Pour la demande de liste des applications connectées, il faut saisir **-l.** L'application affichera alors un entier différent à chaque clé publique associée à une application connectée.
Il est aussi possible de demander les connexions de chaque application (topologie du réseau), à partir de la commande **-r.**

Help / Commande possible :

Pour la demande des commandes disponibles, il faut entrer **-h.** L'application affichera alors un menu d'aide récapitulant toutes les commandes possibles avec un exemple.

Pour une démonstration de chaque fonctionnalité et la génération du fichier JAR, veuillez vous référer au fichier README.

Correction depuis la RFC :

Suite aux retours de la soutenance intermédiaire, plusieurs ajustements ont été effectués sur la conception du protocole et l'implémentation :

- **RFC modifiée** : nous avons légèrement ajusté le comportement de la seconde connexion d'une application en introduisant deux nouveaux paquets : SECOND_JOIN() et CHALLENGE_OK(), afin de sécuriser et valider cette étape.

- **Recommandations appliquées** : conformément aux conseils de l'enseignant encadrant, nous avons :

    - Implémenté un **reader générique** pour les différents types de Reader, ce qui a considérablement amélioré la clarté et la robustesse du code. [annexe img 1]

    - **Allégé l'interface principale**, en la réduisant à deux méthodes stratégiques, puis en la rendant immuable (scellée) pour renforcer l'encapsulation. [annexe img 2]

    - **Fusionner les contextes client et serveur** en une seule classe Context, unifiée et cohérente, afin d'éviter tout problème de programmation et de maintenance.

    - **Supprimer les boucles d'attente active** oubliées dans certains Reader, améliorant ainsi l'efficacité et la non-bloquance du programme. [annexe img 3]

    - **Isolé la gestion des correspondances contextes / clés RSA** dans une classe dédiée, spécialisée et thread-safe 'infoUsers'. [annexe img 4]

    - Créer une **structure Graphe thread-safe** pour gérer dynamiquement les voisins les plus proches (messages publics) et construire des chemins anonymes (messages privés), comme demandé à la soutenance bêta, dans une classe dédiée. [annexe img 5]
    - Mettre en place une **sécurité anti-casse** pour pas qu'un nouveau connecté outrepasse la préconnexion, et pète le réseau en envoyant un broadcast. [annexe img 6]

    - Remplacer tous les System.out.println par un **système de log structuré**, plus adapté à une application en production.

Ces améliorations ont permis de renforcer la structure du code, sa lisibilité ainsi que sa conformité aux exigences de sécurité et de performance du projet.

En ce qui concerne les readers, il existe 3 grandes familles qui permettent de lire l'intégralité des paquets qui s'échange sur le réseau :

- le **PrimaryReader**, qui va lire l'ensemble des paquets de l'application.
- le **PrimaryPayloadReader**, qui va s'occuper de lire uniquement les payloads associées à un broadcast.
- le **PrimaryInstructionReader**, qui va s'occuper de lire les instructions associées à la messagerie cachée.

L'ensemble des readers implémente l'interface scellée Reader pour une meilleure cohérence des readers.

Pour ce qui est des paquets, l'ensemble des paquets du projet implémente l'interface scellée Paquet. Pour une meilleure lisibilité et reconnaissance par Pattern Matching, nous avons utilisé 2 autres interfaces scellées:

- **Instruction** pour les paquets spécialisés dans les instructions de la messagerie cachée.
- **Payload**, pour les paquets spécialisés attachée au broadcast nommé payload.

Pour ce qui est des connexions entre les applications, la classe infoUsers est à la fois propre et commune pour certaines parties à chaque application. Il s'agit d'une classe threadSafe qui contient toutes les informations concernant le réseau dans lequel une application se trouve.

Cette dernière partage certaines informations avec les classes Graphe, et DeconnexionManager (qui sont également threadSafe).

Pour Graphe, elle permet d'établir le routage dans chaque application du réseau, notamment utile pour établir des chemins sécurisés pour la messagerie cachée, ou pour trouver le prochain nœud d'un réseau en communication ouverte.

Et pour le **DeconnexionManager**, pour gérer l'état actuel des connexions directes d'une application.

CommandQueue est également une structure de données threadSafe comportant 2 queues, une pour traiter les commandes saisies par l'utilisateur d'une application, et l'autre spécialisée dans les éventuels messages d'informations internes d'une application.

Chaque contexte est différent. La première distinction vient de l'enum:

- **ContextMode**, pour distinguer la connexion initiée par nous-même ou venant d'une autre appli.

- **ContextProgressStatus**, pour distinguer à quelle étape se trouve un context, afin d'éviter les casses d'une appli qui se connectera et enverra un broadcast par ex au lieu d'un preJoin.
- un booléen, **isFirstConnexions**, pour déterminer s'il s'agit d'une première ou seconde connexion.

Nous utilisons également une classe **Utils**, qui ne contient que des méthodes statiques, dans la majorité des classes du programme. Cela permet d'éviter certaines redondances.

Conclusion :

Ce projet nous a permis de mieux comprendre le fonctionnement concret de certaines applications que nous utilisons au quotidien, comme Discord. Il a marqué une étape importante dans notre apprentissage de la programmation réseau, en nous donnant l'opportunité de mettre en pratique les concepts étudiés en cours, tout en élargissant notre champ de compétences.

La rédaction de notre propre RFC nous a notamment appris à concevoir une application de manière structurée, en réfléchissant en amont aux échanges réseau et à la sécurité, avant même d'écrire une ligne de code. Enfin, ce travail en équipe a renforcé notre capacité à collaborer efficacement, à travers des discussions approfondies, des échanges d'idées constructifs, et une répartition claire des tâches.

Pour finir, en ce qui concerne la répartition du travail, celle-ci était équivalente: lorsque quelqu'un travaillait sur la déconnexion, l'autre gérait la messagerie privée et inversement, par exemple, et, lorsque c'était possible, nous travaillions sur le même poste afin d'avoir 2 personnes concentrées sur la même tâche, ce qui à été le cas de la messagerie cachée, publique, et de la déconnexion.

Cette inversion des tâches permet dans un premier temps de se contrôler mutuellement pour éviter certaines erreurs ou oublis et surtout de garder le fil sur le projet afin de savoir où l'autre en est (c'est quelque chose qui vient compléter les propos verbaux que l'on peut avoir par message).

Annexes:

annexe 1: nous avons séparé les readers en trois catégories: ceux capables lire les paquets de l'interface Paquet (non spécialisés dans la payload/instruction), et ceux capables de lire les buffers associés à des paquets tels que la payload d'interface Payload et l'instruction d'interface Instruction. L'image est associée au reader de paquets, ceux spécialisés dans les payloads et instructions fonctionnent de manière similaire. Et fonctionne sous l'interface scellée Reader.

```
public final class PrimaryReader implements Reader<Paquet> {
    private enum State {
        DONE, WAITING_OPCODE, WAITING_PAQUET, ERROR
    }

    private static final Logger logger = Logger.getLogger(PrimaryReader.class.getName());

    private ProcessStatus localStatus = ProcessStatus.REFILL;
    private State state = State.WAITING_OPCODE;
    private OpCode opCode;
    private Paquet paquet;
    //readers
    private final RSAReader rsaReader = new RSAReader();
    private final ByteReader byteReader = new ByteReader();
    private final LongReader longReader = new LongReader();
    private final NodeReader nodeReader = new NodeReader();
    private final BroadcastReader broadcastReader = new BroadcastReader();
    private final PublicKeyReader publicKeyReader = new PublicKeyReader();
    private final JoinResponseReader joinResponseReader = new JoinResponseReader();
    private final ListReader<Node> listReader = new ListReader<>(new NodeReader());
    private final MessagePublicReader messagePublicReader = new MessagePublicReader();

    public PrimaryReader(Level level) {
        logger.setLevel(level == null ? Level.SEVERE : level);
    }

    @Override
    public ProcessStatus process(ByteBuffer buffer) {
        if (state == State.DONE || state == State.ERROR) {
            throw new IllegalStateException();
        }
        switch(state) {
        //On va récuperer l'opcode du paquet !
        case WAITING_OPCODE:
            localStatus = byteReader.process(buffer);
            if(localStatus != ProcessStatus.DONE) {
                return localStatus;
            }
            opCode = OpCode.intToOpCode(byteReader.get());
            byteReader.reset();
            state = State.WAITING_PAQUET;
        case WAITING_PAQUET:
            switch(opCode) {
                /**************************************
                 ******** GESTION DE LA CONNEXION ********
                 **************************************/
                case PRE_JOIN               -> { paquet = readPreJoin(buffer); }
                case SECOND_JOIN            -> { paquet = readSecondJoin(buffer); }
                case CHALLENGE_PUBLIC_KEY   -> { paquet = readBufferChallengePublicKey(buffer); }
                case RESPONSE_CHALLENGE     -> { paquet = readLongChallengeResponse(buffer); }
                case CHALLENGE_OK           -> { paquet = readChallengeOk(buffer); }
                case JOIN_RESPONSE          -> { paquet = readBufferJoinResponse(buffer); }
```

annexe 2: En complément de l'annexe 1, les 3 principales interfaces (Paquet, Payload, Instruction) sont scellées.
Elles disposent toutes des méthodes '`getWriteModeBuffer()`' et '`getOpCode()`' que nous avons jugées importantes d'intégrer.

```java
public sealed interface Paquet permits
    //Paquet
    Broadcast, PreJoin, SecondJoin, ChallengePublicKey,
    ChallengeLongResponse, ResponseChallenge, ChallengeOk,
    JoinResponse, Connexion, EncodedRSABuffers, ListConnected,
    Message, MessagePublic, Node, MessageToSecure,
    //déconnexion
    LeaveNetworkAsk, LeaveNetworkResponse, LeaveNetworkCancel,
    LeaveNetworkConfirm, LeaveNetworkDone,
    //Payload
    NewNode, NewConnection, RemoveNode,
    //Instruction
    PassForward, SecureMessage
    {

    /**
     * Sérialise le contenu du paquet dans un buffer binaire pour l'envoi.
     * <p>
     * Cette méthode transforme les données internes du paquet en un format binaire (ByteBuffer) qui peu
     * transmis sur le réseau. Elle est utilisée lors de la préparation des données à envoyer à un autre
     * </p>
     *
     * @return Un {@link ByteBuffer} contenant les données sérialisées du paquet prêtes à être envoyées.
     */
    ByteBuffer getWriteModeBuffer();

    /**
     * Retourne le code d'opération associé à ce paquet.
     * <p>
     * Chaque paquet est associé à un code d'opération unique, qui permet de l'identifier et de détermin
     * traitement lors de la réception. Ce code est utilisé pour comprendre le type du paquet et la mani
     * le traiter dans le système.
     * </p>
     *
     * @return Le {@link OpCode} représentant le code d'opération associé à ce paquet.
     */
    OpCode getOpCode();
}
```

annexe 3: Suppression des boucles infinies (laissées par oubli) dans la partie Reader.

```java
private Paquet readPreJoin(ByteBuffer buffer) {
    localStatus = nodeReader.process(buffer);
    switch(localStatus) {
        case REFILL ->  {/*REFILL*/}
        case DONE ->    { return new PreJoin(nodeReader.get());}
        case ERROR ->   { logger.info("Error with manageConnexion"); }
    }
    return null;
}
```

annexe 4: Nous avons isolé la gestion des contextes associés à leur clé publique, ainsi que toutes les informations du réseau dans une unique classe threadSafe nommée infoUsers. Elle permet ainsi de retrouver toutes les informations concernant les applications dans une seule et même structure de données, et non plusieurs.

```java
public final class InfoUsers {
    static private final Logger logger = Logger.getLogger(InfoUsers.class.getName());
    private final HashSet<PublicKeyRSA> publicKeys;
    private final Map<Integer, Node> cachedIndexedPublicKeys;
    private int cachedPublicKeyCount;

    private final PublicKeyRSA myPublicKeyRSA;
    private final Set<Long> myHiddenMessageID;
    private final HashMap<PublicKeyRSA, Set<Long>> lastMessageIDBroadcast;
    private final HashMap<PublicKeyRSA, SocketAddress> appToAddress;
    private final HashMap<PublicKeyRSA, HashSet<PublicKeyRSA>> routageConnexion;
    private final HashMap<SocketAddress, PublicKeyRSA> addressToApp;
    private final Map<Integer, SocketChannel> socketChannels;//<Port, Socket>
    private final HashMap<PublicKeyRSA, Context> appToContext;
    private final HashMap<Context, PublicKeyRSA> contextToApp;
    private final Lock lock;
    //structure de données pour les connexions direct !


    /**
     * Constructeur de la classe InfoUsers.
     * Initialise toutes les structures nécessaires au suivi du réseau.
     * ---
     */
    public InfoUsers(Level level, PublicKeyRSA publicKeyRSA) {
        myPublicKeyRSA = Objects.requireNonNull(publicKeyRSA);
        publicKeys = new HashSet<>();
        cachedIndexedPublicKeys = new HashMap<>();
        lastMessageIDBroadcast = new HashMap<>();
        appToAddress = new HashMap<>();
        myHiddenMessageID = new HashSet<>();
        routageConnexion = new HashMap<>();
        addressToApp = new HashMap<>();
        socketChannels = new HashMap<>();
        appToContext = new HashMap<>();
        contextToApp = new HashMap<>();
        lock = new ReentrantLock();
        logger.setLevel(level == null ? Level.SEVERE : level);

    }
```

annexe 5: Nous avons également utilisé une structure de données spécialisée dans la gestion des graphes pour la messagerie privée et publique.

```java
public final class Graphe {

    private static final int NOMBRE_CHEMIN_MAXIMUM = 20; //Limite de chemins explorés pour
    private static final Logger logger = Logger.getLogger(Graphe.class.getName());

    private final HashSet< PublicKeyRSA > keys = new HashSet<>();
    private final HashMap<PublicKeyRSA, HashSet<PublicKeyRSA>> network;
    private final HashMap<PublicKeyRSA, HashMap<PublicKeyRSA, PublicKeyRSA>> rootNetwork;

    private final ReentrantLock lock = new ReentrantLock();
```

annexe 6: Nous avons également mis en place une sécurité supplémentaire afin de ne permettre l'accès à certaines fonctionnalités telles que les messages privés/publics à d'autre personne qu'à soi-même, les broadcasts qu'aux personnes ayant passé le challenge auprès d'une application du réseau.

Cette mesure supplémentaire est possible grâce à un énumérateur qui évolue lorsque le contexte associé à une application change en passant le challenge.

```
private void handleSecureMessage(EncodedRSABuffers paquet) {
    if(!isConnexionVerified()) { return; }
    logger.info("Message privée de processIn !");
    var encodedBuffer = paquet.buffer();
    if(encodedBuffer == null) {
        logger.warning("Buffer encodé null, paquet invalide.");
        return;
    }
    var decoded = Utils.safeDecryptRSA(encodedBuffer.flip(), privateKeyIntern);
    var instruction = analyseInstruction(decoded);
    server.sendHiddenMessage(instruction);
}

private boolean isConnexionVerified() {
    if(progressStatus == ContextProgessStatus.UNVERIFIED_CHALLENGE
            || progressStatus == ContextProgessStatus.UNVERIFIED_PRE_JOIN) {
        logger.info("Error, you can't access to this protocole");
        return false;
    }
    return true;
}
```