ROMAIN CYRILLE – VIVIEN GALUCHOT – LUDOVIC POUJOL

PROTOCOLE DE

COMUNICATION SVC

«Secure Virtual Connectors»

RAPPORT FINAL

PROJET DE FIN D’ANNÉE

2015-2016

INFO 4 – POLYTECH MARSEILLE

**Table des matières**

[1. Introduction 3](#_Toc451518401)

[2. Présentation du protocole SVC 3](#_Toc451518402)

[3. Propriétés du protocole et méthodes cryptographiques utilisées 4](#_Toc451518403)

[4. Interface d’authentification 4](#_Toc451518404)

[5. Conception et implémentation du protocole SVC 5](#_Toc451518405)

[5.1. Décomposition de la librairie 6](#_Toc451518406)

[5.2. Diagramme logiciel 7](#_Toc451518407)

[5.2.1. Module Socket 8](#_Toc451518408)

[5.2.2. Module Diffie Hellman 8](#_Toc451518409)

[5.2.3. Le module CryptoSym 8](#_Toc451518410)

[5.2.4. Le module SVC 8](#_Toc451518411)

[5.3. Fonctions principales d’utilisation du protocole. 11](#_Toc451518412)

[5.4. Difficultés rencontrées 12](#_Toc451518413)

[6. Comparaison entre les protocoles TLS et SVC 12](#_Toc451518414)

[7. Conclusion 14](#_Toc451518415)

# Introduction

Ce   « projet de synthèse » s’effectue dans le cadre de notre 4ème année d’école d’ingénieurs et permet de solliciter et de mettre en pratique les différentes connaissances acquises au cours de l’année.

Les objectifs de ce projet comprennent l’implémentation du protocole SVC (Secure Virtual Connectors) au travers d’une librairie afin de faire une démonstration de son fonctionnement et de son utilisation. Il a aussi pour objectif la comparaison de ce protocole avec le protocole TLS afin de faire ressortir les avantages de l’utilisation d’un tel protocole qui s’articule autour de la problématique des communications sécurisées de bout-en-bout pour les besoins des applications multi-réseaux.

Le protocole est conçu pour répondre à la problématique des communications internes des applications. Il doit avoir un coût minimum en termes de bande passante et de temps de calcul, tout en conservant un niveau de sécurité maximum et pouvant aussi bien s’intégrer à des applications existantes que servir de base à la construction de nouvelles applications.

Le protocole minimise les échanges pour les besoins en performances et réduire ainsi le surcoût pour assurer la sécurité des échanges. Il ne met pas en place d’interopérabilité entre applications et limite les messages échangés. Il a notamment pour vocation d’être utilisé sur des réseaux mobiles avec des pertes de paquets et de connexions régulières.

La conception de ce protocole s’est faite autour de l’article « A New Secure Virtual Connector Approach for Communication within Large Distributed Systems » écrit par Gabriel RISTERUCCI, Traian MUNTEAN et Léon MUGWANEZA.

Le projet s’est déroulé en trois phases, une première phase de spécification et de conception en suivant une démarche de génie logiciel, une phase d’implémentation et enfin une phase de test, évaluation des performances et validation du code produit.

# Présentation du protocole SVC

Le protocole SVC permet d’établir une connexion sécurisée M2M entres deux machines connectées par des réseaux, éventuellement hétérogènes, afin d’échanger des données. Ce protocole garantit les mêmes propriétés classiques d’un protocole tel que TLS, c’est à dire la confidentialité persistante des données, l’authentification du serveur et du client ainsi que l’anonymat du client. De plus, il se veut plus simple afin de s’intégrer facilement dans la conception des applications et de faire une négociation de la connexion plus efficace et économe en bande passante.

Cependant il vise seulement un certain type de communications où l’utilisateur possède le contrôle sur les deux machines communicantes. Il ne propose pas de mécanisme d’interopérabilité et deux machines n’ayant pas la même version du protocole ne peuvent pas communiquer ensemble. C’est un protocole voué à être utilisé dans des communications internes aux applications, il répond à des besoins étendus des protocoles existants pour des applications sécurisées.

Le protocole a été implémenté sous forme d’une libraire C++ avec peu de fonctions offertes à l’utilisateur afin de faciliter son utilisation. La conception a été faite de sorte que son utilisation diffère peu de l’utilisation d’une simple librairie de sockets afin de pouvoir l’utiliser facilement et même de l’ajouter à une application existante en remplaçant un autre protocole déjà utilisé.

La librairie produite possède les mêmes fonctions d’envoi et de réception qu’une simple communication via des sockets, prenant en paramètre un pointeur vers un espace mémoire, et le nombre d’octets de données à envoyer ou recevoir. La phase de communication est précédée d’une phase de négociation qui permet d’établir la connexion authentifiée et d’échanger les paramètres de sécurité. Cette phase de négociation ne comprend que l’appel d’une seule fonction.

Ainsi ce protocole se présente comme une couche supplémentaire presque transparente pour l’utilisateur dans son utilisation et apportant les propriétés nécessaires pour une communication sécurisée.

Bien que le protocole ne soit pas destiné à être forcément utilisé dans un modèle type client / serveur, nous allons dans la suite du document considérer que la machine en écoute d’une connexion sera le serveur et la machine lançant la connexion sera le client.

# Propriétés du protocole et méthodes cryptographiques utilisées

SVC fournit plusieurs propriétés de sécurités grâces à différents outils cryptographiques. Le protocole commence la connexion par une phase de négociation dont l’objectif est de vérifier les identités à la fois du client et du serveur et de pouvoir procéder ensuite à un échange de clés.

La vérification des identités est laissée à la charge du client grâce à une interface proposée par la librairie, cela permet une plus grande liberté dans les méthodes d’authentification possibles, nous reviendrons sur ce point plus tard (voir § 4). L’échange de clés quant à lui se fait grâce à l’algorithme de Diffie-Hellman en utilisant les courbes elliptiques. Cela permet de partager un secret commun entre les deux entités communicantes sans que celui-ci passe sur le réseau. Grâce à ce secret commun une clé de chiffrement symétrique pourra être dérivée, dans notre cas avec l’algorithme de hachage SHA 256, et utilisée pour établir un lien sécurisé entre deux machines. De plus l’échange de Diffie-Hellman garantit une confidentialité persistante car les clés obtenues sont seulement temporaires et ne peuvent pas être retrouvées à partir de la clé privée des participants.

L’anonymat du client est possible car l’authentification de celui-ci ne s’effectue qu’après l’échange d’une clé de chiffrement symétrique afin d’envoyer son identité dans un canal chiffré.

Une fois la phase de négociation faite, la communication sécurisée entre les deux machines est assurée avec l’algorithme AES-GCM. C’est un algorithme de chiffrement authentifié qui permet en plus de la confidentialité grâce au chiffrement, l’authentification des données grâce à une signature symétrique ainsi que l’intégrité des échanges. Afin de garantir une sécurité optimale les clés de chiffrement sont régulièrement changées, soit après un certain délais ou après qu’un quota de données échangées ait été atteint.

L’ensemble des algorithmes cryptpgraphiques utilisés est formé de ceux recommandés dans l’article de recherche sur lequel se base le protocole. Ils sont implémentés par la bibliothèque cryptographique OpenSSL [ ].

# Interface d’authentification

Aucun algorithme d’authentification n’est propre au protocole. En effet, le choix des méthodes d’authentification du client et du serveur est laissé à la discrétion de l’utilisateur. La librairie propose seulement une interface qui possède les signatures des fonctions devant être implémentées par l’utilisateur pour le bon fonctionnement du protocole. Le fait de laisser libre le choix des méthodes d’authentification laisse une grande liberté à l’utilisateur, et permet une intégration facilitée au sein d’une application déjà existante avec de multiples possibilités. Par exemple, des parties différentes d’une même application développées par des utilisateurs multiples peuvent utiliser des méthodes d’authentification spécifiques.

Les méthodes d’authentification doivent cependant permettre de réaliser une signature numérique et de la vérifier afin de s’assurer qu’il n’y a pas d’usurpation d’identité lors de l’échange des paramètres de sécurité.

# Conception et implémentation du protocole SVC

Le produit fini doit être une bibliothèque de fonctions pouvant être utilisé par n’importe quelle application comme une couche supplémentaire à la transmission de données sur le réseau. Elle a été développée dans le langage C++.

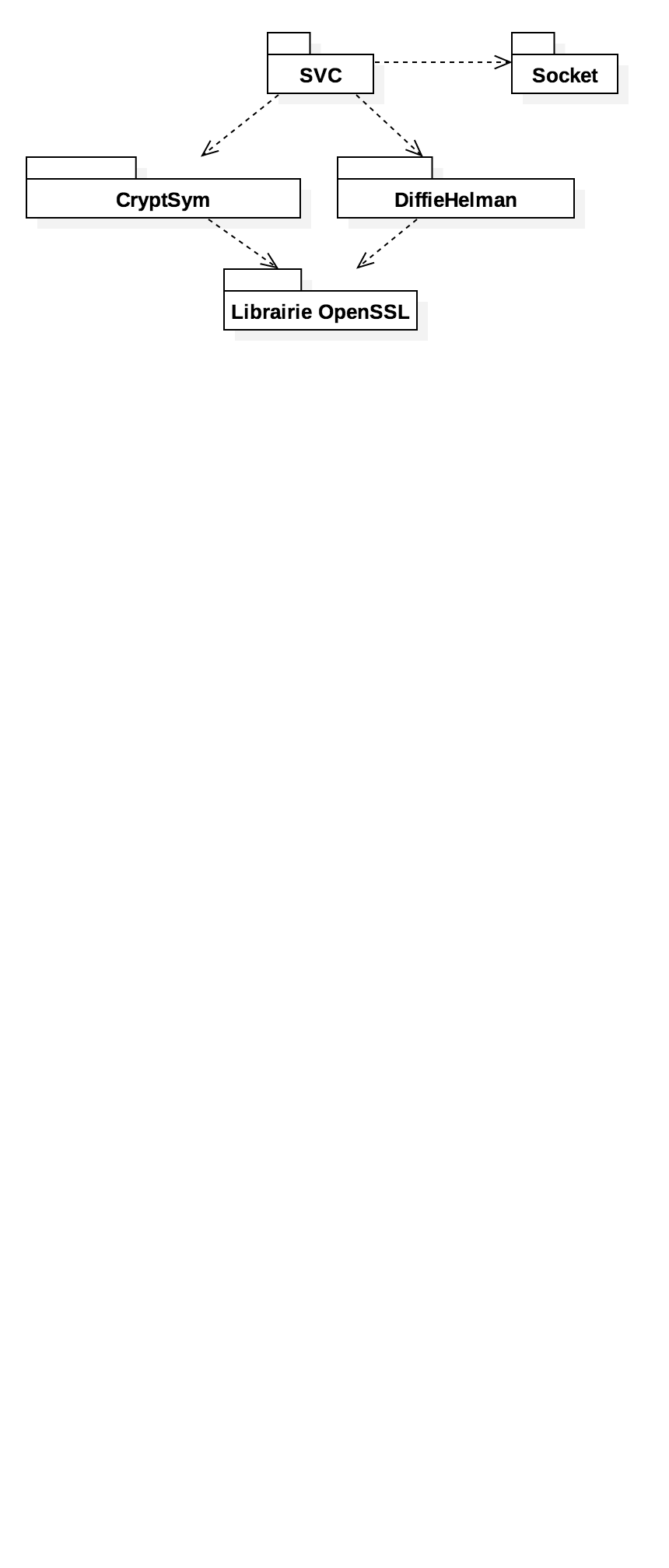
Dans un cas optimal le protocole SVC pourrait se baser sur le protocole UPD afin d’optimiser au mieux l’utilisation de la bande passante et de ne pas être dépendant d’une connexion TCP. Cependant cela implique une plus grande réflexion et une conception plus complexe afin de reproduire un mode semi-connecté. Dans le cas de ce projet et prenant en compte les contraintes de temps imposées pour le développement de la librairie, c’est l’utilisation de TCP qui a été faite. En effet cela permet notamment de garantir le bon acheminement et l’ordre des paquets sur le réseau. Le protocole fourni aussi la phase de connexion et de déconnexion.

De plus si des résultats satisfaisants de point de vue performance sont constatés en utilisant le protocole TCP il ne pourront n’être que meilleurs dans une implémentation utilisant UDP, dans la mesure ou le mode connecté reproduit serait aussi bon que celui de TCP.

Ensuite, la conception de la librairie s’est bien évidemment appuyé sur la définition du protocole SVC présentée dans l’article de recherche. La librairie a été conçue afin de se rapprocher au plus des spécifications données. Les algorithmes cryptographiques utilisés sont ceux proposés et les phases de négociation et de transfert de données correspondent à celle définies. Néanmoins, faute de temps le renouvellement des clés proposé n’a pas pu être implémenté. En effet il a été considéré que ce n’était pas une fonctionnalité essentielle pour la démonstration du protocole, permettant seulement de ne pas compromettre les propriétés de sécurité lors de longs échanges de données.

## Architecture de la librairie

La librairie suit la décomposition ci-dessous, le diagramme de paquetages représentant l’ensemble des modules du programme SVC. Ces modules utilisent la bibliothèque C OpenSSL, qui fournit un ensemble de fonctions cryptographiques, ainsi que les fonctions C standard permettant la manipulation de sockets.



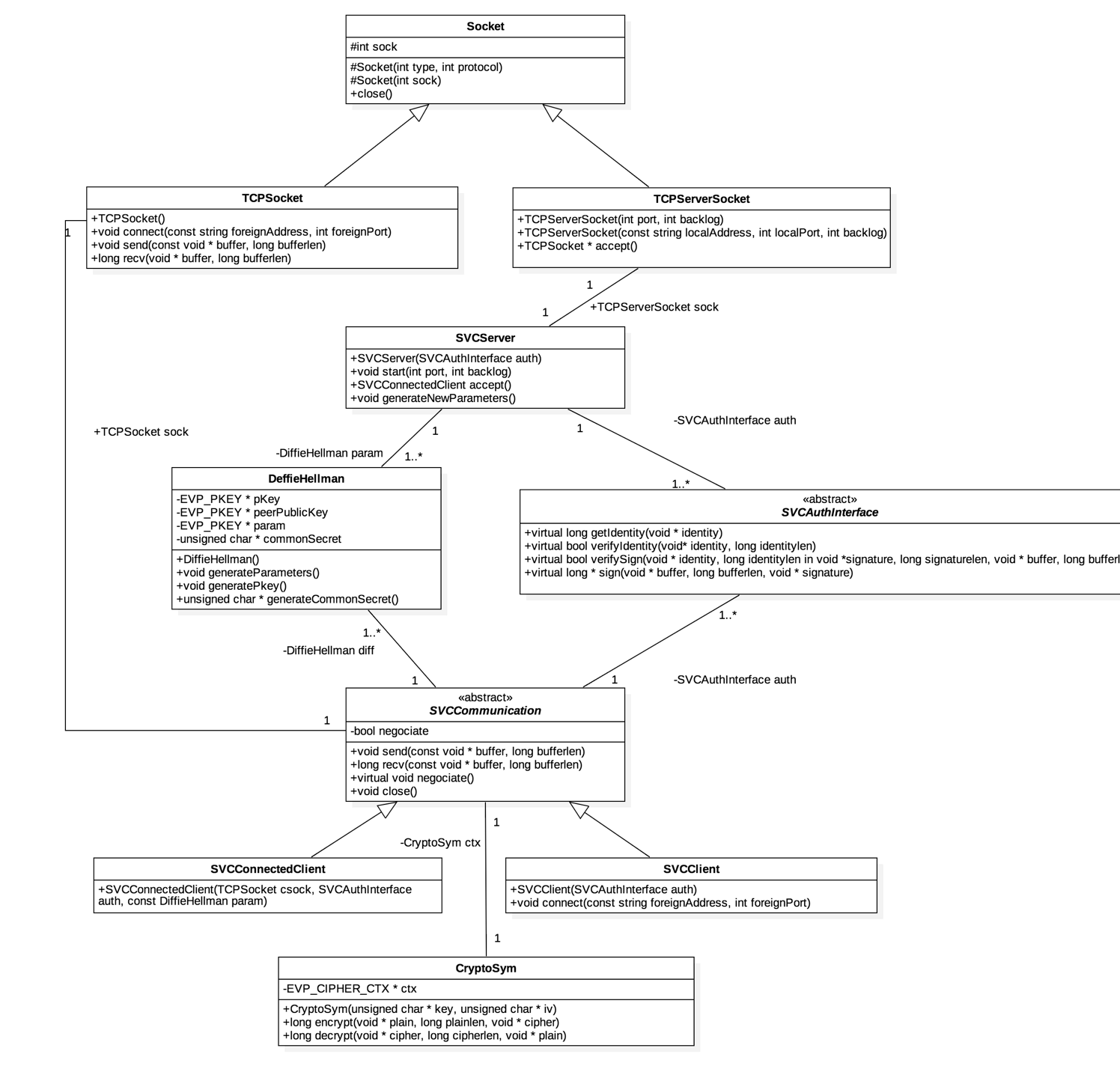
**Figure 1: Diagramme de package de la librairie SVC**

* Le module CryptSym utilise les fonctions d’OpenSSL pour fournir des fonctions de chiffrement symétrique.
* Le module DiffieHelman utilise les fonction d’OpenSSL pour fournir une implémentation d’un algorithme d’échange de clé.
* Le module Socket utilise les fonctions standard C pour fournir des classes de manipulation de socket. Ce module n’est qu’un contenant des fonctions de socket C qui proposent des classes et la gestion des exceptions.
* Le module SVC utilise et dépend de ces différents modules afin d’implémenter les différentes fonctions du protocole SVC, pour la négociation de la connexion et l’échange de données. Ce module est le seul qui est réellement exposé à l’utilisateur, il contient toutes les fonctions nécessaires à l’utilisation du protocole.

## Diagramme logiciel

Le diagramme logiciel suivant présente le détail de ces différents modules avec leur dépendance.

Pour chaque classe sont spécifiés les attributs, les fonctions et leur visibilité. Les getters et les setters des attributs sont cependant omis. La classe abstraite **SVCAuthInterface** est une interface qui nécessite d’être implémentée par l’utilisateur pour fournir au protocole SVC l’interface d’authentification.

On retrouve les fonctions spécifiques – négociation, envoi et réception de données – du protocole défini dans la classe abstraite **SVCCommunication** qui représente le cœur du fonctionnement du protocole.

**Figure 2 : Diagramme logiciel de la librairie SVC**

### Module Socket

Le module Socket est constitué de trois classes :

* La classe **Socket** n’est qu’un simple conteneur d’un fichier descripteur de socket, elle peut aussi bien être utilisé pour faire du TCP que de l’UDP.
* La classe **TCPServer** permet la création d’un serveur en écoute sur un port et en attente de connexion.
* La classe **TCPSocket** est la classe qui permet à un client de se connecter à un serveur. Elle est aussi utilisée pour représenter un client connecté côté serveur.

Une dernière classe fait partie de ce module socket (elle n’apparaît pas sur le diagramme logiciel), c’est la classe **SocketException** qui hérite de la classe de base des exception en C++ std ::exception. Cette classe permet de faire remonter toutes erreurs apparaissant lors de l’utilisation des différentes méthodes de ces classes.

### Module Diffie-Hellman

Ce module ne contient que deux classes, la classe **DiffieHellman** et la classe **DiffieHellmanException** qui de nouveau permet de renvoyer les différentes erreurs pouvant se produire lors de l’utilisation de la classe. Ce module utilise les fonctions d’OpenSSL afin de réaliser l’algorithme d’échange de clé. Lors de l’implémentation de cette classe deux méthodes ont été ajoutées, en effet nous avons fait face à la problématique de l’envoi des structures PKEY sur le réseau afin de partager les clés publiques lors de l’échange. Nous avons décidé de faire de la sérialisation de cette structure et donc d’ajouter une méthode de sérialisation.

### Le module CryptoSym

Ce module permet de faire le chiffrement et le déchiffrement de données avec l’algorithme AES-GCM. Il ne fait qu’utiliser les fonctions d’OpenSSL et facilite leur utilisation via la classe **CryptoSym**. Encore une fois il contient aussi une classe d’exception **CryptoSymException**.

### Le module SVC

Ce module est le cœur du fonctionnement du protocole, il utilise les différentes classes présentes dans les autres modules.

Une classe SVCException fait aussi partie de ce module et permet de prévenir l’utilisateur des différentes erreurs qui peuvent se produire. Lorsqu’une erreur se produit un numéro lui est attribué. Les numéros des erreurs ont été définis avec des macros pour une manipulation facilitée de ces exceptions par l’utilisateur.

En effet plusieurs exceptions peuvent se produire et l’utilisateur peut vouloir les récupérer pour en déduire l’action à faire. On a par exemple des exceptions si l’authentification d’une entité est invalide, ou si l’envoi ou la réception de données a échoué. Cela permet notamment à l’utilisateur d’être informé lors d’une déconnexion et de refaire une connexion au serveur et une négociation afin de continuer le transfert de données. Bien sûr il est à la charge de l’utilisateur de se reconnecter et d’assurer la reprise du service.

#### La classe SVCCommunication

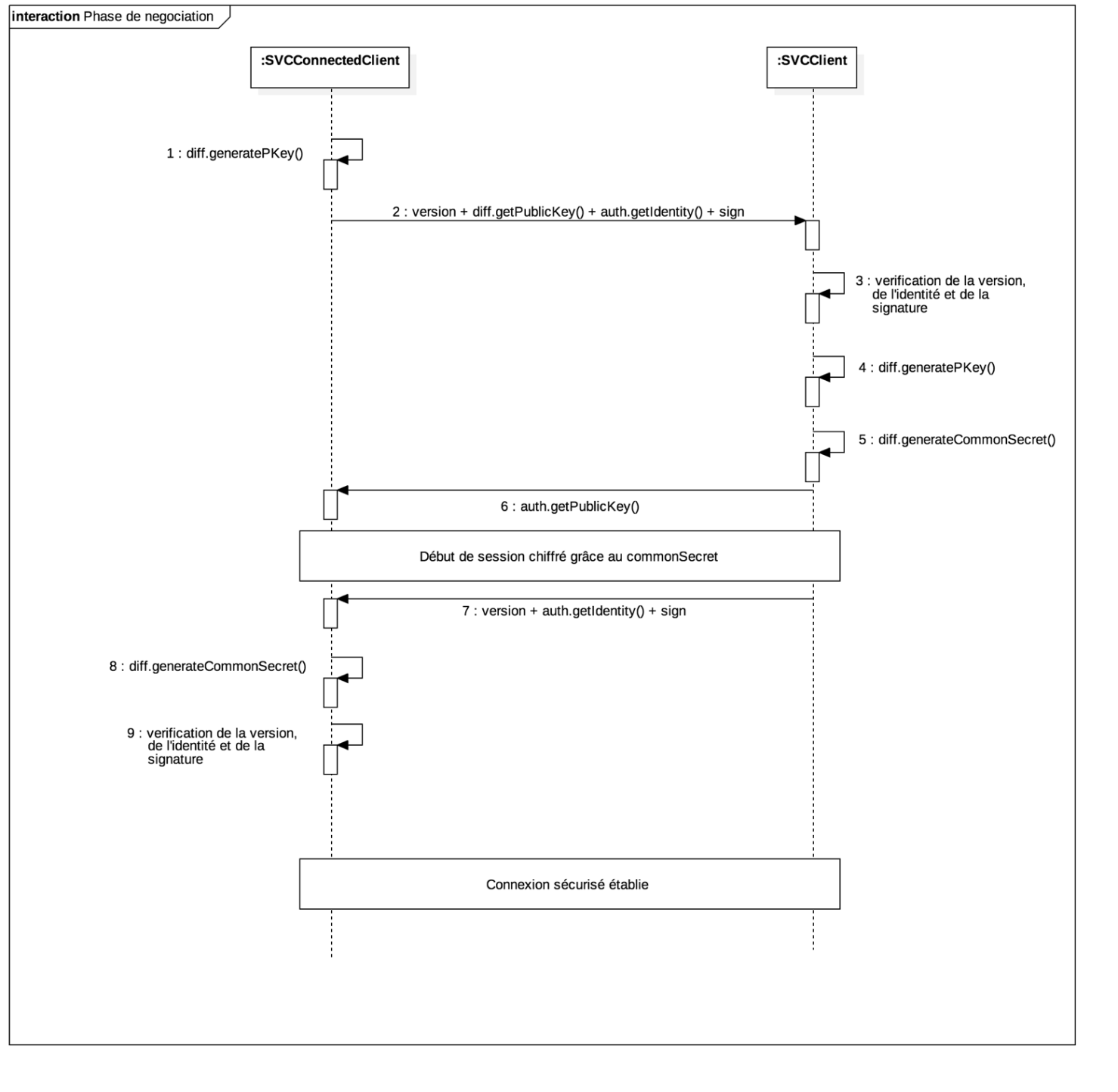
La classe **SVCCommunication** représente une entité de communication du protocole SVC qui peut aussi bien être un client ou la représentation d’un client sur une machine serveur. Cette classe implémente les deux méthodes d’envoi et de réception des données. Comme les données envoyées sont chiffrées par blocs et que lors du déchiffrement les données reçues doivent vérifier un tag d’authentification, il a été nécessaire de mettre en place un système de tampons.

Lorsque l’utilisateur utilise la méthode *send,* toutes les données sont directement envoyées avec le format de trame suivant, il n’y a pas de tampon lors de l’envoi pour ne pas ralentir l’envoi des données :

|  |  |  |
| --- | --- | --- |
| Packet Length | Data | Tag |

Par contre lors de la réception des données, l’utilisateur peut demander de recevoir *n* octects alors que la taille des données chiffrées envoyées est de *m* octects avec *m>n.*  Dans ce cas il est impossible de ne recevoir que les *n* premiers octets car l’authenticité de ces données ne pourrait être vérifiée tant que le tag n’est pas reçu. Il est donc nécessaire de mettre en place un tampon qui reçoit le bloc entier de données envoyées, les déchiffre et permet à l’utilisateur de les récupérer au fur et à mesure comme cela est fait avec des communications sur des sockets simples. Dans l’implémentation actuelle la mise en tampon pourrait être optimisée et le découpage par bloc lors de l’envoi important de données est laissé à la charge de l’utilisateur.

#### La classe SVCConnectedClient et la classe SVCClient

Ces deux classes héritent de la classe **SVCCommunication** et représentent un client et son entité sur le serveur, elles implémentent toutes les deux les fonctions de négociation nécessaires pour l’établissement d’une connexion sécurisé. La négociation a été implémentée selon le diagramme de séquence suivant :

**Figure 3 : Diagramme de séquence de la négociation d'une connexion**

C’est le serveur qui peut commencer l’envoi car la communication est basée sur le protocole TCP il est ainsi possible de connaître la connexion d’un client au serveur et ne nécessite donc pas un premier envoi du client pour s’annoncer. Lors de l’implémentation les différentes données ont été sérialisées si nécessaires puis concaténées avant l’envoi. Ces méthodes de négociation utilisent les méthodes fournies par l’interface d’authentification.

#### La classe SVCServer

Cette classe représente le serveur SVC, elle ne fait qu’utiliser les fonctions socket pour se mettre en écoute sur un port et être en attente de connexion d’un client. A la connexion d’un client elle permet la création d’un objet **SVCConnectedClient**. Cette classe contient aussi les paramètres de base pour générer les clés publiques utilisées lors de l’échange de Diffie-Hellman. Ces paramètres sont copiés lors de la création d’un objet **SVCConnectedClient** et peuvent être renouvelés à la demande de l’utilisateur.

#### La classe SVCAuthInterface

C’est une classe abstraite qui représente une interface pour que l’utilisateur implémente ses propres méthodes d’authentification. Cette interface propose quatre méthodes permettant de récupérer l’identité d’une machine, de vérifier l’identité d’une machine, de signer des données et enfin de vérifier une signature.

## Difficultés d’implémentation

L’implémentation de la librairie ne s’est pas faites sans difficultés. Des ajustements ont dû être faits au niveau de la conception, que ce soit des erreurs de conception ou des oublis.

Ainsi, nous avons par exemple dû changer les types *long* en *size\_t*, qui, en plus de leur sens sémantique, sont plus adaptés et plus compatible avec les différents types de machines pour les communications. Nous rencontrons cependant toujours des problèmes si l’on essaye de faire fonctionner un serveur et des clients sur des architectures différentes (32bits / 64 bits).

Lors du développement de la librairie, la grande majorité des tests unitaires étaient fait en local sur nos propres machines. Masquant ainsi les problèmes de mémoire tampon, que nous avons rencontré lors de nos tests pour les communication réseau notamment lors de la réception de données.

En effet, quand la donnée à envoyer était plus grosse que le MTU de l’interface réseau, celle-ci était alors segmentée en plusieurs paquet TCP qui arrivaient avec quelques instants de décalage. Entre temps, le programme avait déjà bien réceptionné le début de l’information envoyée et commençait à traiter les données sans en attendre la fin, menant à l’interruption du programme. Il a donc été nécessaire d’ajouter des boucle *while* pour bien s’assurer que la totalité des informations était arrivée avant de commencer la négociation ou le déchiffrement de la trame reçue.

Enfin, OpenSSL peut être une librairie très complète par ses fonctions cryptographiques ou son implémentation de SSL/TLS mais cela n’empêche pas un manque flagrant de documentation claire et précise, ni un manque de précision quant à certains messages d’erreur produits.

## Fonctions principales d’utilisation du protocole.

L’utilisation de la librairie se résume en 6 fonctions en plus des constructeurs des classes. Ce sont les suivantes :

* Pour le lancement du serveur et l’acceptation de connexion :
* *start (int port, int backlog)*
* *SVCConnectedClient accept ()*
* Pour la connexion d’un client sur un serveur :
* *connect ( string foreignAddress, int foreignPort)*
* Pour la négociation et l’envoi de message :
  + *negociate ()*
  + *send (void \* buffer, long bufferlen)*
  + *long recv (void \* buffer, long bufferlen)*

Ces fonctions offertes à l’utilisateur reprennent quasiment les mêmes signatures que celles utilisées dans les fonctions standard de socket en C. En effet la librairie a été construite dans l’optique de se rapprocher au plus du fonctionnement normal des sockets afin que l’utilisation de la bibliothèque et l’ajout d’une couche supplémentaire se fasse de manière la plus transparente possible. La seule étape supplémentaire est la négociation qui est gérée par la fonction *negociate ()*.

La négociation n’est pas directement réalisée lors de la connexion d’un client au serveur mais seulement après l’appel à la fonction. Il n’était en effet pas envisageable de faire la connexion sans l’intervention de l’utilisateur. D’une part car l’utilisateur doit pouvoir connaître et contrôler l’étape de négociation, d’autre part dans le cas d’un serveur multi-threads, cela impliquerait d’attendre la fin de la négociation après la connexion d’un client avant d’être de nouveau en mesure d’accepter une connexion. En effet, le retour de la fonction d’écoute d’une connexion ne serait fait qu’après la fin de négociation. Il faut être capable de créer un objet **SVCConnectedClient** qui sera transmis au thread s’occupant du client et c’est seulement dans ce thread que sera faite la négociation.

En plus de ces fonctions l’utilisateur doit fournir, lors de la construction des classes serveur et client, une implémentation de l’interface de **SVCAuthInterface.** .

# Comparaison entre les protocoles TLS et SVC

Les tests étaient une partie intégrante de nos objectifs pour la validation de l’implémentation et la démonstration de l’utilité de SVC face à TLS. Néanmoins, eux non plus ne se sont pas trouvés exempts de problèmes, dont certains n’ont pas pu être résolu à temps, nous empêchant de mener à bien tous les tests prévus.

En effet, avoir un contrôle précis sur les connexions TCP pour s’assurer d’avoir un cadre de test homogène et reproductible est difficile quand on influe sur d’autres paramètres que la bande passante de la connexion (déconnexions, timeouts, perte de paquets …). On a même constaté des écarts importants lors de la réalisation des tests sur des machines différentes. Nous voulions aussi tester l’envoi de fichier en perturbant le réseau avec des déconnexions et mesurer le temps de reprise, cependant la gestion des différents timeouts TCP ne nous a pas permis d’obtenir des résultats satisfaisant et nous n’avons pas eu le temps de poursuivre ces démarches.

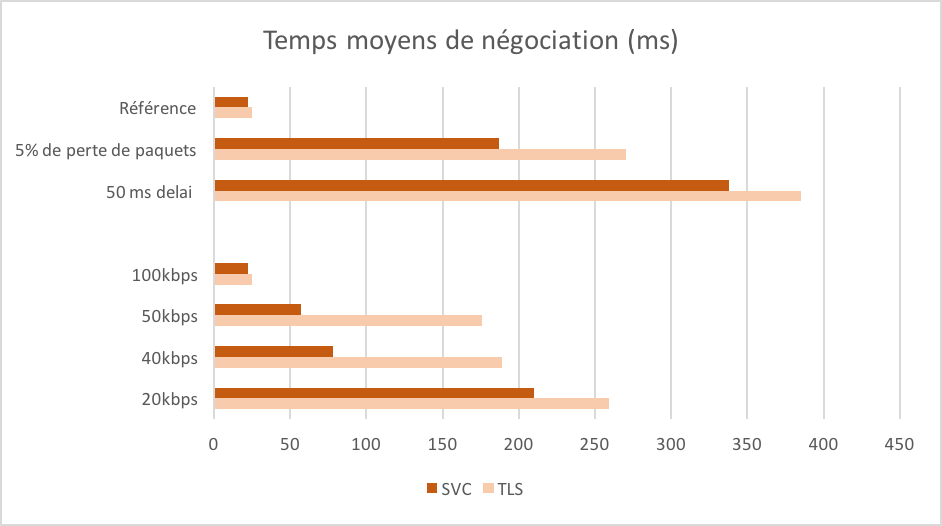
Ainsi, pour les tests effectués, nous accorderons plus d’importance aux moyennes face aux valeurs simples et aux écarts relatifs face aux écarts absolus. Nous avons seulement eu le temps de faire des tests sur la boucle locale que nous avons perturbée, pour simuler des conditions mauvaises de réseau, grâce à l’outil *netem*. Pour être le plus équitable dans notre comparaison nous avons fourni les mêmes PKI (Public Key Infrastructure) au deux protocoles pour réaliser l’authentification du client et du serveur.

Les tests que nous avons obtenus ne sont pas très représentatif d’un cas réel, nous avons seulement voulu faire apparaître une différence entre TLS et SVC et pour cela nous avons réduit fortement le débit du réseau. On arrive ainsi à montrer l’économie de bande passante, le nombre d’échange et de paquets envoyés réalisé par le protocole SVC.

Les tests sont réalisés sur une machine 64bits exécutant le système d’exploitation Debian. Les temps de négociation pour le protocole TLS sont mesurés du coté client car c’est le client qui envoi le premier paquet et reçoit le dernier paquet. Pour le protocole SVC c’est du côté serveur que le temps est mesuré car l’échange commence et se termine sur le serveur.   
Les résultats produit sont les moyennes des temps de négociation obtenus sur 200 mesures avec un temps d’attente entre 2 connexion de 500ms.

Nous avons d’abord pris une valeur de référence, puis tester avec 5% de pertes de paquets et aussi 50ms latence. Ensuite nous avons seulement modifié la bande passante du réseau pour faire apparaître des différences plus importantes. Comme dis précédemment, les tests ne sont pas complets et n’influent que sur un seul paramètre car nous n’avons pas eu le temps d’établir un contexte de test reproductible et constant pour obtenir des résultats valides. De plus les résultats varient selon les machines utilisées.

Les résultats obtenus sont les suivants :



Tout d’abord on remarque que le temps de référence est identique avec une bande passante de 100kbps, on peut donc en conclure que la bande passante n’influe que si elle est inférieure à cette valeur. Pour des valeurs supérieures les résultats sont identiques. Ensuite on remarque qu’une faible perte de paquet ou une latence plus importante influe de la même façon sur le protocole SVC et TLS.   
  
Enfin on remarque facilement que sur des débits très faible SVC est meilleur que TLS et cela est dû aux économies de bandes passantes et sur le nombre d’échange qui sont fait et qui permettent d’être plus rapide dans la négociation.

En conclusion ces tests nous montrent que TLS peut être amélioré dans des contextes particuliers et avec des besoins spécifiques. Pour aller plus loin il faudrait tout d’abord réussir à établir un contexte de tests stable mais aussi implémenter le protocole SVC avec comme support l’UDP. En effet le nombre d’échange est fortement augmenter par le « handshake » TCP et les différents contrôles effectués. Une maitrise plus précise de la connexion entre le serveur et le client pourrait potentiellement réduire les temps de négociation.

# Conclusion

Ce projet nous a permis d’utiliser nos connaissances et d’améliorer notre pratique de la programmation notamment pour les applications réseau. Il a été intéressant d’implémenter un protocole de communication sécurisé et nous avons pu appliquer la théorie des méthodes cryptographiques. Nous avons aussi amélioré notre connaissance de la bibliothèque OpenSSL et de sa très faible documentation.

Malgré une certaine dérive par rapport au planning initial nous avons réussi à produire une librairie fonctionnant et permettant d’arriver à des premiers résultats lors de nos tests. Nous avons essayé de nous mettre à la place de l’utilisateur et proposer des fonctions qui soit le plus compréhensible et facile d’utilisation. De plus nous avons pu mettre en pratique les recherches d’un article scientifique.

Ce projet a aussi été l’occasion de découvrir et d’appréhender le génie logiciel.

Ainsi de bonnes bases ont été faites pour l’implémentation du protocole SVC et des améliorations et ajouts pourraient être faits pour arriver à une version utilisable et complément fonctionnelle.