Stage de L3

Transfert de session TLS

Adrien VANNSON

Encadré par de Daniel Hagimont avec l'aide de Brice Ekane
Réalisé à Irit (Enseeiht) dans l'équipe Sepia

13 juin 2022 – 22 juillet 2022

Table des matières

1 Introduction

Lorsque quelqu'un navigue sur le web, son navigateur, afin d'afficher le contenu demandé, doit envoyer des requêtes à un serveur web. Ces requêtes sont souvent traitées au sein d'un datacenter dont le fonctionnement interne est complexe : une requête et sa réponse peuvent être traitées successivement par plusieurs machines différentes. Ce fonctionnement peut s'avérer problématique lorsque le contenu de la réponse est lourd : transmettre l'ensemble des données d'une machine à une autre prend du temps et consomme de l'énergie. Souvent, une part importante de la réponse à renvoyer au client provient d'une seule machine et n'est pas modifiée par les autres : il n'est donc pas nécessaire de la faire transiter par toutes les machines du datacenter, et il devrait être envisageable de la renvoyer directement. DiSC – Distributed Shared Connection – a été développé pour permettre cela. Avant mon stage, DiSC n'était pas utilisable lorsque la connexion avec le client est chiffrée avec TLS car le serveur devant renvoyer les données ne possédait pas les informations nécessaires au chiffrement. Mon stage a permis de concevoir une méthode permettant de partager une session TLS entre différentes machines. Dans un premier temps, j'ai recherché une méthode permettant de transférer la session TLS et ai développé un prototype mettant en œuvre cette méthode. Puis, je l'ai intégrée à Nginx afin de rapprocher le fonctionnement du prototype de celui de DiSC. Ensuite, une fois le transfert de session TLS intégré à DiSC, j'en ai évalué les performances afin de vérifier son bon fonctionnement et de quantifier l'amélioration des performances obtenue.

2 DiSC

Un datacenter devant répondre à des requêtes émises par des utilisateurs a généralement un fonctionnement interne complexe suivant une architecture multi-tiers. Plusieurs tiers sont impliqués dans le traitement d'une requête. Par exemple, dans le cas d'un service web, trois tiers sont couramment utilisés : un *load-balancer* servant de point d'entrée et permettant de répartir la charge entre les serveurs, un *serveur intermédiaire* devant traiter la requête et un serveur de *backend* accédant aux données. Lorsqu'un client se connecte au serveur web, il établit une connexion avec le load-balancer, et lui envoie sa requête. La requête est transmise au serveur intermédiaire, qui la lit. Le serveur intermédiaire peut demander des données au backend dont il se sert pour construire la réponse. Puis, il transmet sa réponse au load-balancer qui la renvoie au client. Chaque tier est répliqué : plusieurs machines différentes jouent le rôle de serveur intermédiaire et de backend.

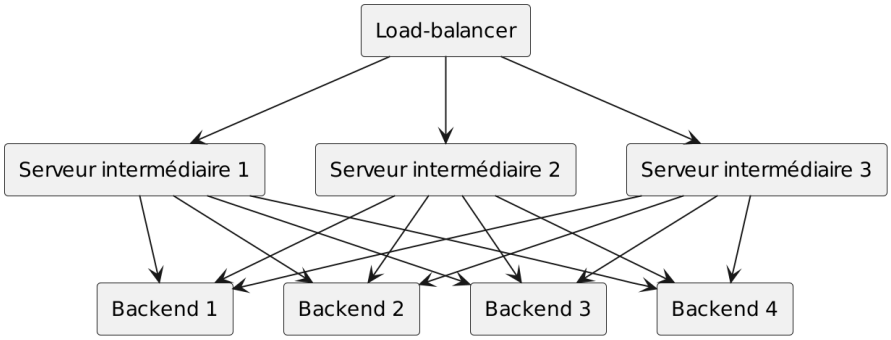

Figure 1 – Architecture 3-tiers

Cette architecture 3-tiers permet une meilleure répartition de la charge et offre des garanties de sécurité en empêchant par exemple le client d'accéder lui-même aux données stockées sur le serveur. Cependant, elle présente l'inconvénient de générer un trafic réseau lourd au sein du datacenter : les données du backend doivent transiter par le serveur intermédiaire et le load-balancer avant d'être renvoyées au client. Si elles ne sont pas modifiées après leur envoi, on dit qu'elles sont *finales*. Le transfert de données finales émises par le backend consomme inutilement les ressources du serveur intermédiaire et du load-balancer. De plus, si le trafic est important, le load-balancer peut être saturé par l'envoi des données.

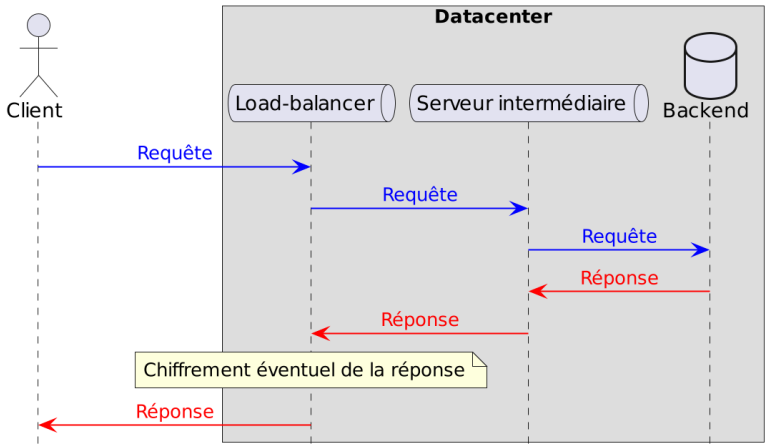

FIGURE 2 – Traitement classique (simplifié) d'une requête

Le protocole DISC – Distributed Shared Connection – a été développé pour éviter ces transferts de données inutiles. Il donne la possibilité à toutes les machines du datacenter d'utiliser la connexion TCP établie initialement entre le client et le load-balancer. Cela permet par exemple au backend de renvoyer directement des données finales au client.

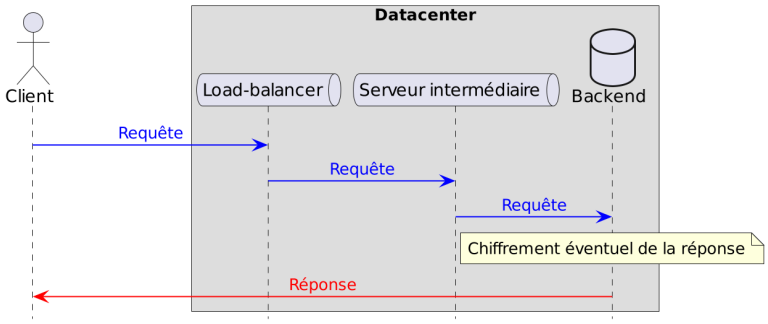

FIGURE 3 – Traitement (simplifié) d'une requête avec DISC

Pour répartir la charge sur le load-balancer ainsi que pour traiter les requêtes sur le serveur intermédiaire et le backend, DISC utilise le logiciel NGINX. Ce logiciel doit être modifié légèrement : c'est un des principaux inconvénients de DISC.

DISC n'était pas utilisable lorsque la connexion entre le client et le load-balancer est chiffrée : le backend ne possède pas les informations nécessaires au chiffrement et ne peut donc pas chiffrer lui-même les données. Mon stage a eu pour objectif de résoudre ce problème en proposant une méthode permettant de partager une session chiffrée entre différentes machines. J'ai tout d'abord réalisé un prototype qui a permis de montrer qu'une session chiffrée peut être transférée d'une machine à une autre. Puis, j'ai intégré ce prototype à NGINX avant qu'il soit intégré à DISC. Finalement, j'ai évalué le gain de performances permis par le transfert de session.

3 Quelques rappels de réseau

Pendant mon stage, j'ai été amené à utiliser plusieurs protocoles réseau. Cette section résume les quelques points de leurs fonctionnements nécessaires à la compréhension de la suite de ce rapport.

3.1 Le protocole IP

Le protocole IP – Internet Protocol – permet d'envoyer un paquet d'une machine à une autre. Les machines sont identifiées par leur adresse IP. Un paquet IP est composé d'un en-tête suivi des données à envoyer. L'en-tête contient notamment l'adresse IP de l'émetteur et du destinataire du paquet.

3.2 Le protocole TCP

Le protocole TCP – Transmission Control Protocol – est un protocole réseau permettant de rendre fiable une connexion entre deux machines. Un échange utilisant TCP est composé de trois phases : l'établissement de la connexion, l'échange des données et la fermeture de la connexion. Sur la figure 4, chaque flèche représente un paquet IP. Les paquets envoyés sont acquittés, et retransmis si nécessaire.

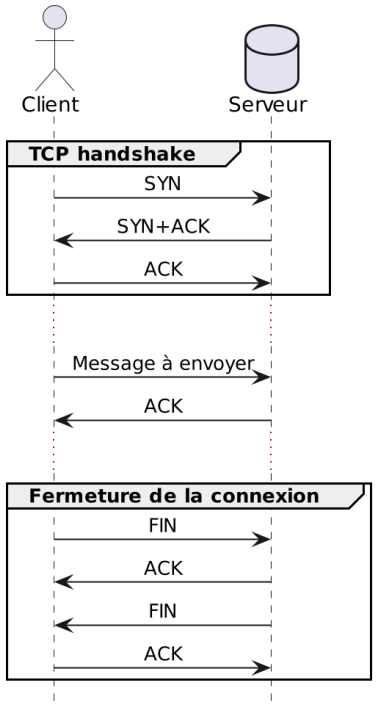

FIGURE 4 – Exemple de connexion avec TCP

3.3 Le protocole TLS

Le protocole TLS – Transport Layer Security – permet d'assurer la sécurité d'une connexion en :

- Donnant la possibilité au client d'identifier le serveur
- Chiffrant les échanges
- Vérifiant l'intégrité des données

Une connexion TLS doit être initialisée. Pendant cette initialisation, le client et le serveur utilisent du chiffrement asymétrique pour échanger une clé secrète. Cette clé servira à chiffrer les échanges suivants à l'aide de chiffrement symétrique, beaucoup plus rapide.

3.4 Le protocole HTTP

Le protocole HTTP – Hypertext Transfer Protocol – permet d'échanger des données entre un client – par exemple un navigateur web – et un serveur web. Il utilise TCP pour réaliser les échanges. Une variante, HTTPS, utilise en plus TLS. Un message envoyé avec le protocole HTTP est composé de deux parties : un en-tête suivi du corps du message.

4 État de l'art

D'autres solutions que DISC permettent déjà d'éviter les transferts de données inutiles au sein d'un datacenter. Elles sont décrites ci-dessous. Cependant, elles présentent des défauts que DISC a pour objectif de corriger.

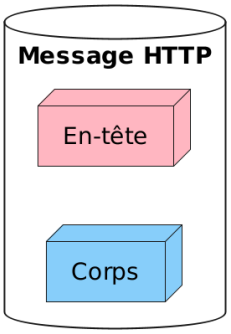

Figure 5 – Structure d'un message Http

4.1 Redirection Http

Le load-balancer, lorsqu'il reçoit une requête, peut rediriger le client vers un serveur de backend. Cette méthode est très peu adaptée :

- Elle n'est pas transparente pour le client.
- L'Url est modifiée, ce que l'utilisateur peut constater.
- Elle est plus lente : le client doit envoyer deux fois sa requête.
- Elle ne fonctionne qu'avec le protocole Http
- Ces redirections sont souvent permanentes : le client contactera toujours le même backend, impossible de répartir dynamiquement la charge

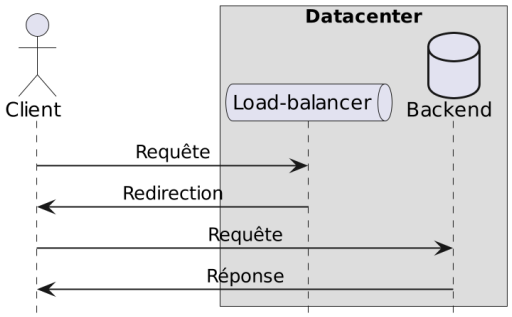

Figure 6 – Redirection Http

Le client doit établir deux connexions différentes, l'utilisation de TLS ne pose donc pas de problème.

4.2 Crab

Crab [3] est un projet de recherche conçu par Marios Kogias, Rishabh Iyer et Edouard Bugnion, chercheurs à l'Epfl. Son objectif est d'éviter à un load-balancer de transmettre toutes les données devant être renvoyées au client. Le load-balancer est seulement utilisé pour établir la connexion TCP. Ensuite, le client et le backend communiquent directement entre eux. Cela est permis par une modification du protocole TCP : une option `Connection Redirect` est ajoutée aux en-têtes.

Cette approche est relativement contraignante : la modification du protocole TCP impose de modifier le client. Le client doit donc être considéré comme faisant partie de l'application, ce qui n'est souvent pas le cas. Par exemple, il ne serait pas envisageable d'utiliser Crab pour un serveur web, car le code s'exécutant sur les machines des utilisateurs ne peut pas être modifié. De plus, il peut parfois être souhaitable que le load-balancer interprète des protocoles de plus haut niveau que TCP, comme par exemple Http (ce que fait Nginx). Cela n'est pas possible avec Crab.

L'article ne mentionne pas l'utilisation de TLS. Cependant, étant donné que Crab effectue la répartition de la charge au niveau de TCP, on s'attend à ce que TLS puisse être utilisé sans que des modifications supplémentaires soient nécessaires (la session TLS n'aurait pas à être transférée d'une machine à une autre).

4.3 Prism

Prism [2] est un projet de recherche développé par Yutaro Hayakawa (Line Corporation), Michio Honda (University of Edinburgh), Douglas Santry (Apple Inc.) et Lars Eggert (NetApp). Il a été présenté lors de la 18ème édition de la conférence Usenix Symposium on Networked Systems Design and Implementation qui s'est tenue en avril 2021. Prism permet de partager une session TCP utilisant éventuellement TLS entre plusieurs machines. L'objectif est similaire à celui de DiSC : améliorer les performances des serveurs stockant des données en permettant à un serveur de backend de renvoyer directement sa réponse à une requête au client sans que les données transitent inutilement par le load-balancer avec lequel la connexion a été établie. Cependant, Prism est limité par sa conception destinée à une architecture matérielle spécifique : les machines doivent être connectées à un même commutateur. Ce commutateur est reconfiguré à chaque requête pour qu'il envoie les paquets à la bonne machine cible, celle-ci n'étant pas toujours la même. Cette contrainte peut être limitante : certains anciens commutateurs ne se reprogramment pas, ou alors lentement. De plus, Prism ne peut pas être utilisé avec des machines n'étant pas connectées directement à un même commutateur.

Pour gérer des connexions TLS, Prism utilise une version modifiée de la librairie TLSe. Elle permet aux différents tiers partageant la connexion de s'échanger l'état de la session TLS, afin que chacun d'entre eux puisse chiffrer lui-même les données à envoyer. Cette bibliothèque semble assez peu maintenue et utilisée, n'est pas complètement compatible avec OpenSSL, n'est presque pas documentée et se comporte parfois de manière inattendue, ce qui pourrait s'avérer problématique pour une utilisation en conditions réelles.

5 Réalisation d'un prototype

Avant d'essayer d'intégrer le transfert de session TLS à DiSC, il a tout d'abord fallu rechercher une méthode permettant de transférer une session TLS entre deux machines, puis réaliser un prototype simple mettant en œuvre cette méthode afin de montrer qu'elle fonctionne correctement.

5.1 Méthode envisagée

La bibliothèque OpenSSL [5] est souvent utilisée pour chiffrer des échanges avec le protocole TLS. Il semblait donc initialement naturel de l'utiliser. Il fallait alors un moyen de sérialiser l'état actuel de la session TLS pour pouvoir l'envoyer vers une autre machine et l'y restaurer. Mes encadrants, qui s'étaient déjà intéressés au problème avant le début de mon stage, m'ont tout d'abord suggéré d'utiliser les fonctions `i2d_SSL_SESSION` et `d2i_SSL_SESSION` de l'API d'OpenSSL. Cependant, celles-ci ne permettaient pas d'exporter toutes les données nécessaires, il n'était donc pas possible de les utiliser. Il s'est ensuite avéré qu'OpenSSL ne permettait pas de transférer une session TLS nativement. Modifier directement le code d'OpenSSL pour ajouter cette fonctionnalité semblait trop complexe, j'ai donc recherché une autre approche. J'ai tout d'abord trouvé la bibliothèque TLSe [6], notamment utilisée dans le projet de recherche Prism [2]. C'est avec elle que j'ai développé le premier prototype qui fonctionnait correctement. Cependant, TLSe était un petit projet qui semblait peu fiable et assez peu maintenu. J'ai donc recherché une alternative, et j'ai finalement choisi d'utiliser WolfSSL [7]. Cette bibliothèque a notamment pour objectif de permettre l'utilisation de TLS sur des machines possédant des ressources très limitées, comme des systèmes embarqués. De tels systèmes n'ayant pas toujours les ressources nécessaires pour réaliser certaines étapes du chiffrement, WolfSSL a intégré la possibilité d'exporter une session TLS pour leur donner la possibilité de déléguer le chiffrement à d'autres machines plus puissantes. J'ai choisi d'utiliser cette fonctionnalité non pas sur des systèmes embarqués mais sur les machines du datacenter, pour permettre au backend de chiffrer lui-même les données à envoyer.

5.2 Architecture du prototype

L'objectif de ce premier prototype n'est pas de développer un exemple réaliste et complet de chiffrement depuis le backend, mais simplement de montrer que transférer une session TLS est possible.

J'ai donc choisi d'utiliser un environnement simple : le datacenter est modélisé par deux processus s'exécutant sur une même machine. Sans transfert de session, le prototype fonctionne de la manière suivante, représentée en figure 7 :

— Le client – un navigateur web – envoie une requête HTTPS au load-balancer
— Le load-balancer, écrit en C, reçoit la requête, la déchiffre et la transmet au backend
— Le backend, écrit en C, lit le contenu de la requête et renvoie une réponse non chiffrée
— Le load-balancer reçoit la réponse du backend, la chiffre et la transmet au client

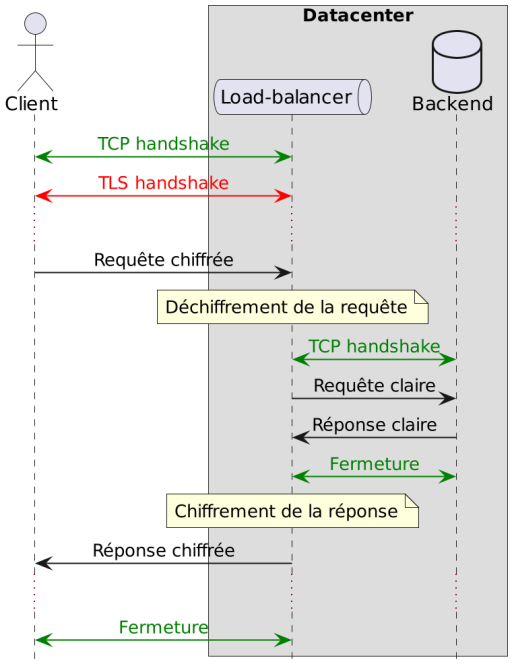

FIGURE 7 – Prototype sans transfert de session TLS

Une fois ce prototype fonctionnel, j'y ai intégré le transfert de session TLS afin de chiffrer les données depuis le backend. Le prototype fonctionne alors de la manière suivante, représentée en figure 8 :

— Le client envoie une requête HTTPS au load-balancer
— Le load-balancer reçoit la requête, la déchiffre et la transmet au backend avec les informations de session
— Le backend :
  — Restaure la session
  — Lit le contenu de la requête
  — Génère une réponse
  — La chiffre
  — L'envoie au load-balancer
  — Renvoie l'état de session TLS après chiffrement au load-balancer
— Le load-balancer restaure la session et envoie la réponse chiffrée au client

J'ai pu réaliser ce prototype et le faire fonctionner correctement, ce qui a permis d'obtenir un moyen de partager une session TLS entre deux processus distincts. La section suivante détaille le protocole de communication entre le load-balancer et le backend que j'ai choisi d'utiliser.

5.3 Protocole de communication

Le load-balancer et le backend doivent communiquer pour s'échanger le contenu de la requête, la réponse, les informations de session TLS, ... Il a fallu concevoir une méthode permettant l'échange de ces informations de manière fiable. Avant l'intégration du transfert de session, les deux processus communiquaient à l'aide d'une socket TCP. Le problème rencontré avec cette méthode est que les processus doivent être capables de faire la différence entre les paquets contenant la requête ou la

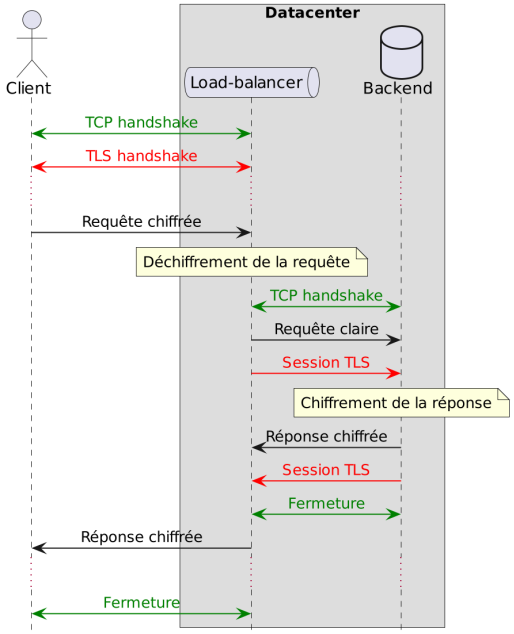

FIGURE 8 – Prototype avec transfert de session TLS

réponse et les paquets contenant une session TLS. Le problème se résout simplement pour les envois de données du load-balancer au backend. En effet, le load-balancer peut simplement envoyer, dans l'ordre :

- La taille de la requête
- Le contenu de la requête
- La taille de la session TLS
- La session TLS

Ainsi, le backend connait la taille des données qu'il doit lire, et peut donc les lire complètement sans difficulté. Cependant, avec TLSE, il n'était pas possible d'utiliser cette méthode pour envoyer les données du backend au load-balancer : les données chiffrées sont envoyées directement sans qu'il soit possible d'accéder à leur taille. J'ai trouvé différentes manières d'envoyer les données.

5.3.1 Lire les "TLS records"

Les données envoyées avec TLS sont encapsulées dans des paquets appelés *"TLS records"*. Pour obtenir la taille des données à lire, j'ai tout d'abord tenté de lire l'en-tête des "TLS records" reçus sur le load-balancer, puis de lire la quantité de données correspondante. Cette méthode fonctionnait bien pour les fichiers suffisamment petits pour être contenus dans un unique "TLS record". Cependant, dès que plusieurs records étaient envoyés, un nouveau problème se posait : combien faut-il en lire ? En raison de la structure spécifique de l'en-tête d'un TLS record, j'ai pu les différencier des données de session TLS, et faire fonctionner le prototype. Cependant, cette méthode restait assez complexe.

J'ai aussi testé une autre méthode en comptant le nombre de TLS records envoyés depuis le backend, mais je me suis aperçu que je n'avais pas de garantie que la valeur que j'obtenais soit la valeur exacte. Je n'ai donc pas approfondi cette méthode.

5.3.2 Avec WOLFSSL

Le protocole de communication a pu être simplifié avec l'utilisation de WOLFSSL. En effet, contrairement à TLSE, WOLFSSL permet de connaître la taille des données chiffrées avant de les envoyer. J'ai donc pu utiliser la même méthode que pour l'envoi des données du load-balancer au backend, en envoyant depuis le backend :

- La taille de la réponse chiffrée
- Le contenu de la réponse

— La taille de la session TLS
— La session TLS

5.4 Conclusion

Après avoir testé différentes méthodes permettant de communiquer des informations entre le load-balancer et le backend, je suis parvenu à développer un prototype fonctionnel permettant de réaliser le chiffrement des données sur le backend, ce qui était l'objectif fixé. Avant que cette méthode soit intégrée à DiSC, j'ai réalisé un nouveau prototype utilisant cette fois Nginx comme load-balancer. L'objectif était de montrer que l'utilisation du transfert de session avec Nginx est bien possible.

6 Intégration du prototype dans Nginx

Le load-balancer utilisé dans DiSC utilise Nginx [4] – un logiciel permettant de jouer le rôle d'un serveur Http ou d'un load-balancer. Pour intégrer le transfert de session TLS à DiSC, il a donc fallu commencer par l'intégrer à Nginx en réalisant un nouveau prototype. La difficulté de cette intégration venait principalement de la taille du code source de Nginx : le prendre en main s'est parfois avéré difficile.

6.1 Utilisation de Nginx avec WolfSSL

Nginx prend en charge les connexions TLS, mais il utilise pour cela la bibliothèque OpenSSL qui ne permet pas le transfert de session. Heureusement, WolfSSL peut s'utiliser de la même façon qu'OpenSSL : les fonctions d'OpenSSL sont implémentées avec le même nom et le même comportement. En modifiant très légèrement le code de Nginx, j'ai pu le compiler avec WolfSSL au lieu d'OpenSSL.

6.2 Ajout d'un module à Nginx

Afin d'éviter d'avoir à modifier en profondeur le code source de Nginx et d'obtenir un code plus facile à maintenir, j'ai tout d'abord tenté d'implémenter le transfert de session TLS dans un module Nginx – Nginx permet l'ajout de *filtres* qui sont appliqués sur la réponse envoyée par le backend pour la modifier avant de la transmettre au client. Cependant, cette approche n'a pas abouti. En effet, le load-balancer doit renvoyer un en-tête Http indiquant notamment la taille des données de la réponse... sans envoyer lui-même la réponse car le backend s'en charge. En faisant cela, Nginx se bloquait en attendant les données à envoyer.

6.3 Modification du code source de Nginx

N'ayant pas trouvé comment intégrer le transfert de session TLS à Nginx en utilisant un module, j'ai finalement choisi de modifier le code source de Nginx. Tout d'abord, pour éviter de rencontrer des problèmes lors de la réception des données, le backend doit renvoyer un en-tête Http dont le champ `Content-Length` est mis à 0. La taille réelle des données est stockée dans un champ `Backend-Content-Length`. Deux champs supplémentaires sont également créés : `Backend-IP` et `Backend-port`. Ils donnent au load-balancer un moyen de recontacter le backend. Pour envoyer des données au client lorsqu'une connexion TLS est utilisée, Nginx utilise une fonction nommée `ngx_ssl_write`. J'ai modifié le code de cette fonction pour que :

— Elle modifie l'en-tête Http pour envoyer la bonne valeur de `Content-Length`.
— Elle supprime les champs commençant par `Backend-` de l'en-tête.
— Elle envoie l'en-tête au client.
— Elle établisse une nouvelle connexion avec le backend, lui envoie la session TLS, attende qu'il envoie sa réponse puis reçoive le nouvel état de la session.

Cette méthode a fonctionné correctement, montrant ainsi qu'il est possible d'intégrer le transfert de session TLS à un load-balancer utilisant Nginx.

7 Évaluation des performances

La méthode de transfert de session TLS que j'ai proposée a finalement été intégrée à DiSC par Brice Ekane, terminant ainsi le développement d'un prototype complet dans lequel le backend chiffre et envoie lui-même les données. J'en ai mesuré les performances afin de constater :

— Que les données sont envoyées directement depuis le backend
— La baisse de la charge des serveurs bypassés
— L'augmentation de la charge du backend liée au chiffrement

La figure ci-dessous décrit l'architecture choisie pour mesurer les performances de DiSC. Quatre machines sont utilisées : un client, un load-balancer, un serveur intermédiaire et un backend. Toutes sont hébergées sur CloudLab [1].

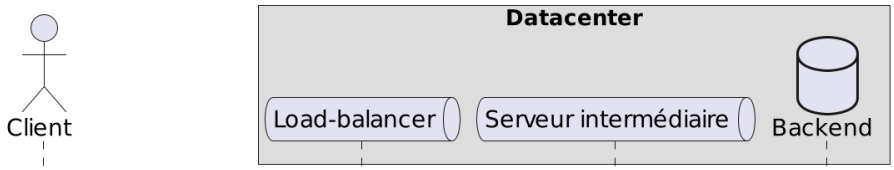

Figure 9 – Architecture de test

Afin de répondre convenablement aux objectifs fixés, j'ai choisi de mesurer les données suivantes :

— Sur le load-balancer :
  — Temps de calcul par requête
— Sur le serveur intermédiaire :
  — Temps de calcul par requête
  — Données reçues sur le réseau par requête
— Sur le backend :
  — Temps de calcul par requête

7.1 Protocole de mesure

Pour des raisons de reproductibilité, les mesures sont effectuées par un script s'exécutant sur une machine tierce que l'on appellera par la suite *évaluateur*. Celui-ci établit une connexion SSH avec le client, le load-balancer, le serveur intermédiaire et le backend. Ces connexions lui permettront de paramétrer les machines, de lancer des requêtes depuis le client et d'effectuer les mesures.

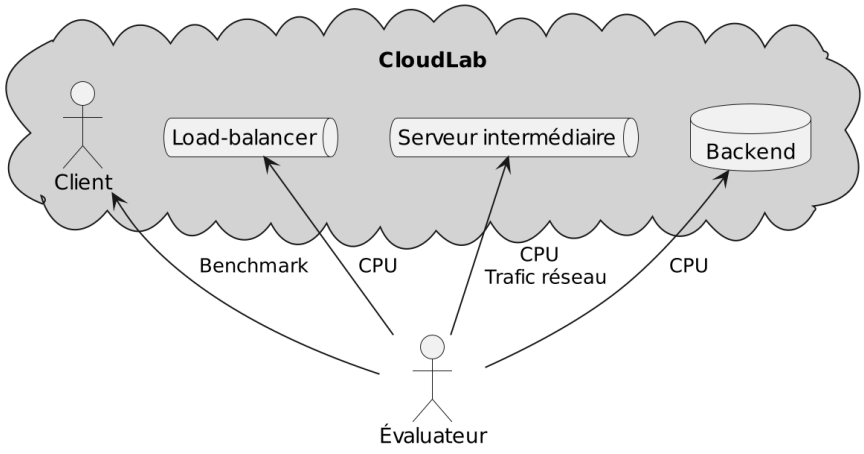

Figure 10 – Machines utilisées pour évaluer DiSC

L'évaluation des performances est réalisée en plusieurs étapes :

1. Fixation de la fréquence des CPUs
2. Mesure des consommations de CPU initiales des *workers* Nginx

3. Mesure de l'utilisation initiale du réseau du serveur intermédiaire
4. Envoi des requêtes par le client
5. Mesure des consommations de CPU finale des *workers* NGINX
6. Mesure de l'utilisation finale du réseau du serveur intermédiaire
7. Relâchement de la contrainte sur la fréquence des CPUs

7.1.1 Envoi des requêtes

Les requêtes sont envoyées par le client avec `curl`. Chaque requête est envoyée de nombreuses fois, afin d'obtenir des mesures suffisamment fiables. Toutes les requêtes sont indépendantes : une nouvelle connexion TCP ainsi qu'une nouvelle connexion TLS doivent être initialisées pour chacune d'entre elles.

7.1.2 Mesure du temps CPU

La fréquence d'un CPU n'est pas constante : la technologie DVFS – Dynamic voltage and frequency scaling – permet de l'ajuster en temps réel en fonction de la demande de l'utilisateur. Cela permet une exécution rapide des programmes en cas de forte demande et une consommation d'énergie limitée en cas de faible demande. Pour ne pas que le temps CPU mesuré soit influencé par ces variations de fréquence, l'évaluateur commence par fixer la fréquence CPU des machines à 800 MHz avec la commande `echo 10 | sudo tee /sys/devices/system/cpu/intel_pstate/max_perf_pct`. On peut vérifier la fréquence des CPUs avec `cpufreq-info`.

Le temps total pendant lequel un processus de pid donné s'est exécuté – le *temps CPU* – est lu dans le fichier `/proc/[PID]/stat`.

7.1.3 Mesure du trafic réseau

La quantité de données reçues par une machine sur une interface réseau est obtenue avec la commande `ifconfig [interface]`, où `[interface]` est le nom de l'interface. Elle est affichée sur la ligne commençant par `RX packets`.

7.2 Réalisation des mesures

Le protocole de mesure décrit ci-dessus a été utilisé dans quatre scénarios :
— Sans DISC (*"Vanilla"*), sans TLS
— Sans DISC (*"Vanilla"*), avec TLS
— Avec DISC, sans TLS
— Avec DISC, avec TLS

À chaque fois, les mesures sont réalisées avec des tailles de fichiers à envoyer variant de 1 ko à 4 Mo. Initialement, certains résultats n'étaient pas cohérents. La détection de ce problème a permis de découvrir des erreurs lors de l'intégration à DISC. Elles ont pu être corrigées avant la réalisation des mesures finales.

7.3 Résultats

Les courbes obtenues lors des mesures finales sont représentées ci-dessous. Il y en a quatre :

1. Le trafic réseau moyen par requête sur le serveur intermédiaire
2. Le temps CPU moyen par requête sur le load-balancer
3. Le temps CPU moyen par requête sur le serveur intermédiaire
4. Le temps CPU moyen par requête sur le backend

Pour des raisons de clareté, l'échelle est logarithmique en abscisse et linéaire en ordonnée, ce qui explique l'allure exponentielle de certaines courbes.

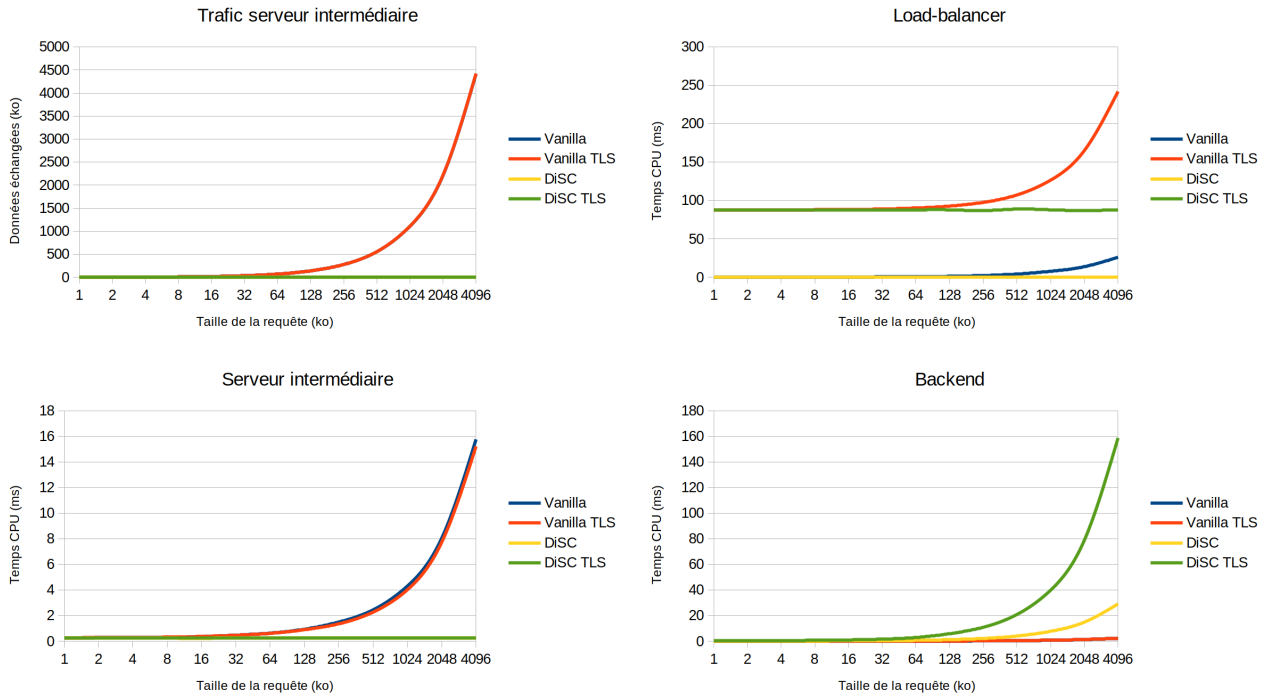

FIGURE 11 – Résultats des évaluations de performance

Les mesures du trafic réseau sur le serveur intermédiaire permettent tout d'abord de constater l'efficacité de DISC : lorsque DISC n'est pas utilisé (cas vanilla), la quantité de données transmises par le serveur intermédiaire est proportionnelle à la taille des fichiers envoyés. Dès que DISC est utilisé, les données sont envoyées directement par le backend. Le trafic réseau du serveur intermédiaire est alors très faible et surtout indépendant de la taille du fichier envoyé. De même, le temps de calcul nécessaire au serveur intermédiaire pour traiter une requête est proportionnel à la taille du fichier envoyé lorsque DISC n'est pas utilisé, et est constant sinon. L'utilisation ou non de TLS n'influence pas le serveur intermédiaire car celui-ci ne doit ni initialiser la connexion TLS, ni chiffrer les données.

Le même phénomène peut être constaté sur l'utilisation du processeur du frontend : lorsque DISC n'est pas utilisé, celle-ci est constante, alors qu'elle augmente avec la taille des données sinon. Cependant, sur le frontend, l'utilisation ou non de TLS a une incidence. Dans les deux cas où TLS est utilisé, un temps indépendant de la taille du fichier – il vaut environ 80ms – est nécessaire : il est utilisé pour établir la connexion TLS. En effet, pendant cette initialisation, une clé de chiffrement symétrique est échangée entre le load-balancer et le client à l'aide de chiffrement asymétrique, ce qui est un processus coûteux. On peut également constater que, dans les cas où DISC n'est pas utilisé, le temps CPU par requête augmente plus rapidement lorsque TLS est utilisé que lorsqu'il ne l'est pas. Cela est dû au chiffrement des données effectué par le load-balancer. On peut d'ailleurs constater que pour des fichiers de taille inférieure à 1 Mo, le temps de chiffrement est négligeable devant le temps d'établissement de la connexion TLS.

Sur le backend, lorsque DISC est utilisé, on constate aussi une différence de temps de calcul selon l'utilisation ou non de TLS. Il est plus élevé lorsque TLS est utilisé car le backend doit alors chiffrer lui-même les données. On peut aussi noter que l'utilisation de DISC augmente légèrement le temps de calcul par requête.

La figure 12 présente les mêmes données d'une autre manière, en mettant en avant la différence de performances entre une architecture utilisant DISC et une architecture classique dans les mêmes conditions d'utilisation. On constate que sur des fichiers de taille assez élevée, DISC permet de réduire la charge CPU totale, que TLS soit utilisé ou non. Lorsque TLS est utilisé, on remarque aussi un transfert de charge du load-balancer au backend. Il est causé par le report du chiffrement des données sur le backend. Ce transfert de charge n'est visible que pour des fichiers volumineux : pour les petits fichiers, l'essentiel du temps est passé à établir la connexion TLS.

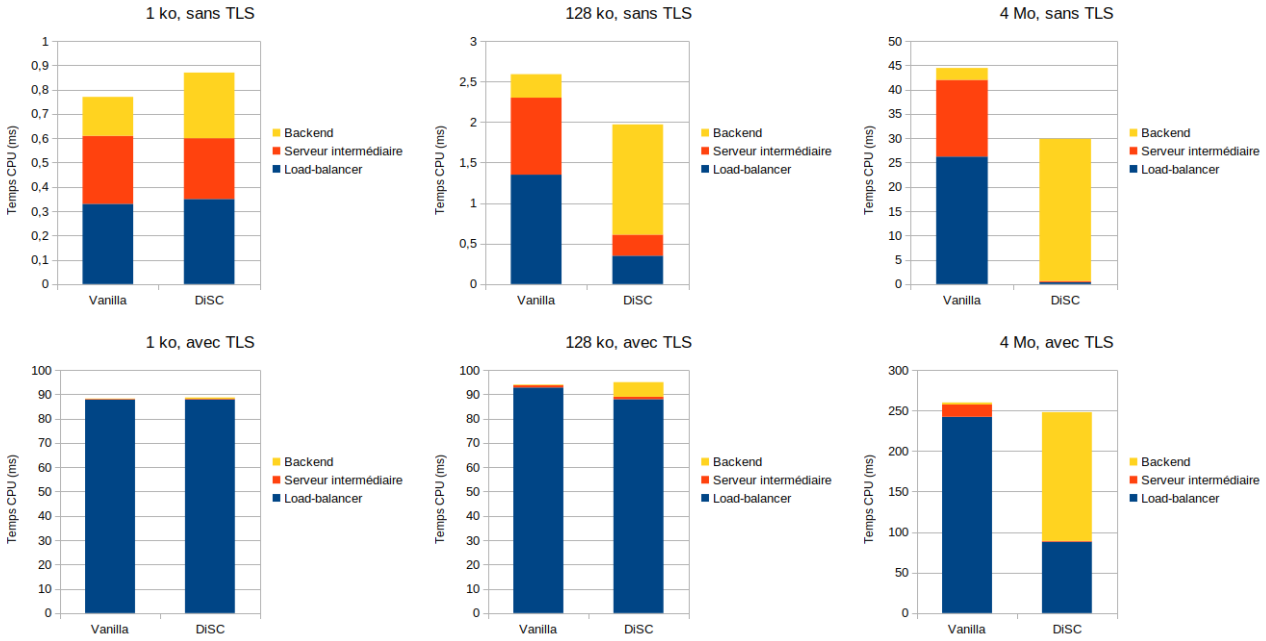

FIGURE 12 – Comparaison entre utilisation avec DISC et sans DISC

8 Conclusion

Mon stage a permis de développer une méthode de transfert de session TLS, d'intégrer cette méthode à NGINX et d'évaluer les performances de son intégration à DISC. La recherche de la méthode de transfert ainsi que la réalisation du premier prototype ont pu se dérouler sans problèmes majeurs. J'ai ensuite passé beaucoup de temps à travailler sur l'intégration à NGINX : j'ai rencontré des problèmes liés à la complexité de ce logiciel, et d'autres liés à WOLFSSL. Cela ne m'a pas laissé assez de temps pour réaliser moi-même l'intégration à DISC. J'en ai simplement évalué les performances à la fin de mon stage. Cette évaluation de performances a permis de détecter des erreurs dans l'intégration de TLS à DISC. Une fois ces erreurs corrigées, elle a permis de constater le bon fonctionnement de l'intégration ainsi que de mesurer le gain de performances obtenu, ce qui était l'objectif initial.

Références

[1] *CloudLab*. URL : https://www.cloudlab.us/.

[2] Yutaro HAYAKAWA et al. "Prism : Proxies without the pain". In : *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 2021, p. 535-549.

[3] Marios KOGIAS, Rishabh IYER et Edouard BUGNION. "Bypassing the load balancer without regrets". In : *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, p. 193-207.

[4] *Nginx*. URL : https://www.nginx.com/.

[5] *OpenSSL*. URL : https://www.openssl.org/.

[6] *TLSe*. URL : https://github.com/eduardsui/tlse.

[7] *WolfSSL*. URL : https://www.wolfssl.com/.

A L'Irit, l'équipe Sepia

L'Irit – Institut de Recherche en Informatique de Toulouse – est une unité mixte de recherche (UMR 5505). Elle réunit environ 600 membres répartis sur sept sites majoritairement proches de Toulouse. Ses recherches se déclinent selon cinq axes majeurs :

- Conception et construction de systèmes (fiables, sûrs, adaptatifs, distribués, communicants, dynamiques...)
- Modélisation numérique du monde réel
- Concepts pour la cognition et l'interaction
- Étude des systèmes autonomes adaptatifs à leur environnement
- Passage de la donnée brute à l'information intelligible

Les résultats obtenus sont ensuite appliqués dans les *domaines d'action stratégique* suivants :

- Santé, Autonomie, Bien-être
- Ville Intelligente
- Aéronautique, Espace, Transports
- Média sociaux numériques et diffusion de l'information
- e-Education
- Cybersécurité, Sécurité des biens et des personnes

Les vingt-quatre équipes du laboratoire sont réparties dans sept départements de recherche :

- Département ASR : Architecture, Systèmes, Réseaux (5 équipes)
- Département CISO : Calcul Intensif, Simulation, Optimisation (2 équipes)
- Département FSL : Fiabilité des Systèmes et des Logiciels (4 équipes)
- Département GD : Gestion de Données (3 équipes)
- Département IA : Intelligence Artificielle (3 équipes)
- Département ICI : Intelligence Collective, Interaction (3 équipes)
- Département SI : Signaux et Images (4 équipes)

L'équipe Sepia, appartenant au département ASR, s'intéresse surtout à la gestion de ressources dans les datacenters. Ses recherches se font selon trois axes principaux :

- Optimiser l'énergie (consommation et effets thermiques à l'intérieur des datacenters)
- Consolider dans les datacenters virtualisés (c'est-à-dire regrouper les machines virtuelles sur les mêmes machines physiques pour pouvoir éteindre des machines)
- Améliorer le support du système d'exploitation (permettre à l'hyperviseur de communiquer avec les systèmes d'exploitation afin d'optimiser la gestion des ressources)

B Codes écrits

Les codes écrits pendant mon stage sont disponibles en ligne :

- Le premier prototype :
  http://perso.ens-lyon.fr/adrien.vannson/l3-internship/proof-of-concept.tar.xz
- L'intégration à Nginx :
  http://perso.ens-lyon.fr/adrien.vannson/l3-internship/nginx-load-balancer.tar.xz
- L'évaluation de performances :
  http://perso.ens-lyon.fr/adrien.vannson/l3-internship/evaluation.tar.xz