Optimisation des entrées-sorties dans les architectures multi-tiers

Brice Ekane Apah

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Optimisation des entrées-sorties dans les architectures multi-tiers

input-output optimization in multi-tiers architectures

Présentée par :

Brice EKANE APAH

Direction de thèse :

Noel DE PALMA PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Directeur de thèse
Daniel HAGIMONT Professeur, Institut National Polytechnique de Toulouse	Co-directeur de thèse

Rapporteurs :

JEAN-MARC MENAUD
Professeur, IMT ATLANTIQUE
DAVID BROMBERG
Professeur des Universités, UNIVERSITE RENNES 1

Thèse soutenue publiquement le **8 décembre 2022**, devant le jury composé de :

NOEL DE PALMA Professeur des Universités, UNIVERSITE GRENOBLE ALPES	Directeur de thèse
JEAN-MARC MENAUD Professeur, IMT ATLANTIQUE	Rapporteur
DAVID BROMBERG Professeur des Universités, UNIVERSITE RENNES 1	Rapporteur
ALAIN TCHANA Professeur des Universités, GRENOBLE INP	Président
DANIEL HAGIMONT Professeur des Universités, TOULOUSE INP	Co-directeur de thèse
MATHIEU BACOU Maître de conférences, TELECOM SUDPARIS	Examinateur

"à ma famille"

Remerciements

Il me sera très difficile de remercier tout le monde car c'est grâce à l'aide de nombreuses personnes que j'ai pu mener cette thèse à son terme. Je voudrais tout d'abord remercier grandement mes directeurs de thèse Noel DE PALMA et Daniel HAGIMONT pour leur soutien tout à long de cette thèse. Je suis ravi d'avoir travaillé en leur compagnie car outre leur appui scientifique, ils ont toujours été là pour me soutenir et me conseiller au cours de l'élaboration de cette thèse. Jean-Marc MENAUD et David BROMBERG m'ont fait l'honneur d'être rapporteurs de ma thèse. Je leur remercie pour le temps consacré à l'évaluation de mon travail. Je tiens à remercier Alain TCHANA pour avoir accepté de participer à mon jury de thèse et pour sa participation scientifique ainsi que le temps qu'il a consacré à ma recherche. Il a toujours été présent pour me conseiller et m'encourager. Je remercie également Mathieu BACOU pour l'honneur qu'il me fait d'être dans mon jury de thèse. Je remercie également toutes les personnes avec qui j'ai partagé mes études et notamment ces années de thèse. Mes derniers remerciements vont à ma famille qui a tout fait pour m'aider, qui m'a soutenu et surtout supporté dans tout ce que j'ai entrepris.

Résumé

Pour des raisons de modularité et de scalabilité, la plupart des services en ligne déployés dans des datacenters reposent sur une architecture multi-tiers dans laquelle plusieurs composants logiciels sont déployés sur des machines différentes (par exemple un équilibreur de charge, un serveur web, une application métier et un serveur de base de données). Ces tiers interagissent à travers des communications réseaux, le plus souvent reposant sur le protocole TCP/IP. Un client externe au datacenter peut se connecter à un tier frontal (le point d'entrée) et soumettre une requête, provoquant un traitement distribué entre les tiers et retournant au client une réponse incluant des données. Les interactions entre les tiers suivent le modèle classique client-serveur.

Nous observons que dans ces services, une quantité significative des données retournées au client sont émises par des tiers dans cette architecture, sans que les tiers par lesquels passe la réponse et ces données (au retour, en passant par le tier frontal) n'exploitent les données. Les tiers qui précédent le tier émettant les données ne font que les propager. Les communications nécessaires pour faire remonter ces données dans l'architecture multi-tiers sont inutiles et sources d'une charge de calcul significative dans le datacenter.

L'objectif de cette thèse est de réduire ces communications inutiles en introduisant un mécanisme de raccourci. Lorsque dans l'architecture multi-tiers, un tier génère des données qui ne sont pas exploitées par les tiers qui le précédent avant leur retour au client, alors un raccourci permet à ce tier de retourner directement les données au client sans passer par les tiers intermédiaires. La mise en place d'un mécanisme de raccourci revient à distribuer la connexion TCP entre le client et le tier frontal, celle-ci devenant accessible à tous les tiers.

Deux problèmes rendent difficile l'implantation de cette notion de raccourci. Premièrement, il faut coordonner les différents tiers émetteurs pour garantir la cohérence interne de TCP (en particulier la gestion des numéros de séquence de TCP et des acquittements). Deuxièmement, les raccourcis bousculent la cohérence interne des applications et il faut modifier les applications dans ce sens. Les solutions actuelles nécessitent soit une modification du protocole TCP, soit des modifications importantes des applications.

Nous montrons comment il est possible de distribuer une session TCP comme évoqué précédemment sans modification du protocole TCP. Nous montrons que les modifications sur les applications sont mineures et qu'il est même possible d'implanter des raccourcis sans modification des applications. Enfin, nous conduisons une évaluation de performances qui montre que ces raccourcis permettent de réduire significativement la charge de calcul dans le datacenter, sans dégrader les performances des communications en termes de latence ou de débit.

Abstract

For modularity and scalability reasons, most online services deployed in datacenters rely on a multi-tier architecture where several software components are deployed on different machines (e.g. a load balancer, a web server, a business logic application, and a database server). These tiers interact through network communications mostly relying on the TCP/IP protocol. A client, external to the data center, can connect to a frontal tier (the entry point) and submit a request, triggering a distributed execution between tiers which returns to the client a response that includes data. Interactions between tiers follow the classical client-server model.

We observe that in such services, a significant amount of data returned to the client are emitted by tiers in this architecture, without any handling by other tiers that lie between the emitting tier and the frontal tier. The tiers which precede the emitting tier only forward such data until the frontal tier returns the data to the client. Therefore, network communications used to transfer such data through the multi-tier architecture until the frontal tier are needless and a significant source of CPU waste in the data center.

The objective of this thesis is to reduce these needless communications by introducing a shortcut mechanism. In a multi-tier architecture, when a tier emits data that is returned to the client without any other handling by intermediate tiers, a shortcut allows the emitting tier to directly return data to the client without traversing intermediate tiers. The introduction of a shortcut mechanism is equivalent to making the TCP connection with the client remotely accessible from any tier in the architecture.

Two problems make it hard to implement such shortcuts. First, emitting tiers have to be coordinated in order to enforce the TCP internal consistency (especially the management of TCP sequence numbers and acknowledgments). Second, shortcuts compromise the internal consistency of applications which have to be modified accordingly. Current solutions require either a modification of the TCP protocol or important modifications in applications.

We show that it is possible to distribute a TCP connection as mentioned above without any modification in the TCP protocol. We show that modifications to applications can be minimal and that it is even possible to implement shortcuts without any modification to applications. Finally, we conducted a performance evaluation that shows that shortcuts allow a significant reduction of the CPU load in the data center, without degrading communication performance in terms of latency and throughput.

Table des matières

1 Introduction

1.1 Contexte

Les outils informatiques sont utilisés dans notre vie de tous les jours, que ce soit à titre privé comme professionnel. Et ces outils sont principalement distribués, c'est à dire communiquants par un réseau. S'il y a encore quelques applications que nous utilisons localement sur un poste de travail, la plupart des applications permettent la communication ou l'interaction entre des individus ou des institutions et nécessitent des communications sur la toile.

Ainsi les deux dernières décénnies ont vu la proliferation de ce qu'on peut désigner sous le terme services en ligne. Il s'agit d'applications qui sont déployées dans des centres d'hébergement (aussi appelés datacenters) suivant la philosophie du cloud computing, et qui fournissent un service partagé entre un ensemble d'usagers qui se connectent à ce service partagé depuis des applications situées sur leur poste de travail (généralement un navigateur web, mais pas que).

Des exemples de tels services sont les sites de commerce électronique (comme Amazon ou Ebay), les banques en lignes (comme Boursorama), les outils de messageries életroniques (que ce soit le mail classique SMTP/IMAP ou Messenger), des outils de partage (comme GoogleDrive ou GoogleAgenda) ou les réseaux sociaux (comme Facebook).

Dans les datacenters, ces services, qui peuvent être utilisés par un très grand nombre d'usagers, nécessitent des architectures logicielles complexes et surtout permettant le passage à l'échelle aussi appelé scalabilité. Pour assurer la scalabilité, ces services sont généralement construits en assemblant des composants logiciels qui s'exécutent sur des machines différentes pour bénéficier d'une plus grande puissance de calcul. De plus ces composants sont souvent répliqués pour gagner encore plus en capacité d'absorption de la charge provenant des usagers.

La construction de ces architectures logicielles suit souvent une organisation qualifiée de multi-tiers (ou multi-couches). Le principe est que chaque tier est responsable d'une tâche précise, reçoit des requêtes provenant soit d'un client externe soit d'un autre tier, et peut nécessiter l'appel à un autre tier (avec une requête). Les interactions entre les tiers reposent sur le modèle client-serveur. Les tiers sont organisés en couches, ce qui implique qu'une requête reçue par un premier tier va potentiellement être propagée de tier en tier jusqu'au dernier tier (la couche la plus profonde), puis les réponses aux requêtes vont remonter de tier en tier (avec un traitement potentiel au niveau de chaque tier) jusqu'à ce qu'une réponse soit retournée au client externe. Cette organisation est illustrée sur la Figure 1.1.

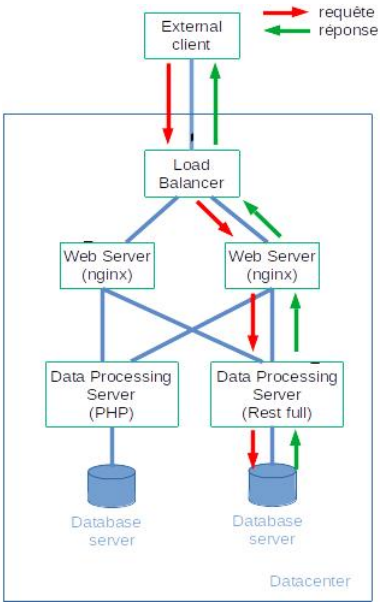


FIGURE 1.1 – *Architecture multi-tier*

1.2 Position de problèmes

Le problème général que nous adressons dans cette thèse est l'économie des ressources dans les datacenters. Alors que les datacenters associés au développement du cloud computing se sont répandus sur la planète, les opérateurs de ces datacenters se sont très vite posés la question de gérer efficacement les ressources, de la même façon qu'un système d'exploitation se doit de gérer efficacement les ressources d'une machine. Une gestion efficace des ressources dans un datacenter permet de réduire le nombre de machines utilisées pour satisfaire les usagers et également de réduire l'empreinte énergétique du datacenter.

De nombreux travaux de recherche se sont intéressés à l'économie des ressources dans les datacenters. On peut citer les travaux concernant la consolidation des serveurs dans des infrastructures virtualisées [AGH+15, FAP+14, CFF14], des travaux concernant l'économie de la mémoire avec des techniques de swap [AMMR92] ou de compression [ES05], ou des travaux visant directement l'économie d'énergie en adaptant les fréquences des processeurs (DVFS) [KGWB08]. On peut observer que ces travaux se sont intéressés à gérer les ressources globalement dans les datacenters, et ils ont principalement adressé la gestion des ressources CPU et mémoire. Cependant, peu de travaux se sont intéressés à la gestion des ressources de communication. En réalité, les stratégies mises en place pour gérer globalement dans le datacenter les ressources CPU et mémoire sont distribuées et utilisent des communications supplémentaires.

Les communications dans un datacenter ont un impact général sur l'utilisation des ressources matérielles, que ce soit les cartes réseaux des machines, mais aussi les mémoires et CPU des machines qui communiquent

Dans cette thèse, nous partons de l'observation que de nombreux communications sont inutiles, provenant de l'architecture des applications déployées dans le datacenter. Le but est alors de réduire ces communications inutiles, ce qui doit permettre de réduire non seulement l'utilisation des équipements réseaux du datacenter, mais surtout de réduire l'utilisation des ressources (CPU et mémoire) sur les machines communiquantes qui exécutent les protocoles de communication.

1.3 Contributions

Comme évoqué précédemment, nous nous situons dans le contexte des applications multi-tiers déployées dans les datacenters. Dans ces applications, comme illustré sur la Figure 1.1, une requête peut être propagée de tier en tier, puis une réponse peut être générée et faire le chemin inverse pour être retournée au client externe. Nous décrivons ici un schéma général, car les interactions entre les tiers peuvent être plus complexes. Notamment, un même tier peut appeler plusieurs tiers, les protocoles entre les tiers peuvent être différents et un tier peut modifier les données reçues par un tier pour générer sa réponse au tier qui le précéde.

Dans ce contexte, nous considérons qu'une donnée incluse dans une réponse est finale par rapport à un tier lorsque ce tier n'exploite pas cette donnée (en lecture ou en écriture) pour ses besoins internes et ne fait que relayer cette donnée dans sa réponse au tier précédent. En d'autres termes, dans une chaîne de tiers A-B-C, si le tier C envoie au tier B une réponse avec une donnée D et B envoie à son tour sa réponse incluant D au tier A sans exploiter D, alors nous disons que D est finale par rapport à B.

L'idée que nous développons est que l'on peut mettre en place un Raccourci permettant d'envoyer la donnée D directement de C à A. Un tel raccourci permet d'éviter le transit de D par la machine B, donc de soulager celle-ci en termes de communications réseau donc de charge de calcul nécessaire pour le traitement de ces communications. Notons que souvent dans de telles applications les données finales de taille significative sont des images (mais pas que) et lorsqu'une donnée est finale, elle l'est pour toute la chaîne de tiers et peut être retournée directement au client avec un raccourci contournant toute la chaîne de tiers. La donnée peut alors être émise par le tier émetteur directement sur la connexion avec le client externe.

Dans nos contributions, nous faisons l'hypothèse que les protocoles de communication entre les tiers reposent sur le protocole TCP. Etant donné qu'une donnée finale peut faire l'objet d'un raccourci et être envoyée sur une connexion appartenant à un autre tier, cela implique d'implanter une notion de session TCP distribuée où une session TCP est accessible à distance. Ou autrement dit, de maîtriser l'art du spoofing.

Mais implanter ce mécanisme de raccourcis pose de nombreux problèmes, les deux principaux étant la cohérence applicative et la cohérence de TCP.

La cohérence applicative concerne le fonctionnement logique des applications. Une application construite sur le modèle client-serveur appelle une autre application en envoyant une requête et attend une réponse. On ne peut pas simplement lui enlever la réponse (en raison d'un raccourci) sans casser son fonctionnement. Dans un premier temps, nous montrons comment mettre en place ces raccourcis en modifiant les applications et nous montrons que ces modifications sont relativement simples (les applications pourraient facilement inclure ces modifications comme une option). Dans un second temps, nous montrons comment il est possible d'implanter le mécanisme de raccourcis sans aucune modification des applications, en exploitant des mécanismes de pagination pour implanter une notion de transfert à la demande (une faute de page récupère les données, sinon il y a raccourci).

La cohérence de TCP concerne principalement les numéros de séquence de TCP. Si on permet à plusieurs tiers d'émettre des messages sur la même connexion TCP, on a forcément un problème de cohérence des numéros de séquence (et aussi de traitement des acquittements), car celui qui reçoit s'attend à recevoir des numéros de séquence linéaires correspondant à un flot de données (stream). En fait, lorsqu'un tier A possède une connexion TCP et qu'un autre tier B est autorisé à émettre dessus, cela ne peut se faire que de façon exclusive (une seul tier à la fois peut émettre) et les tiers A et B doivent se coordonner pour émettre des numéros de séquence cohérents. Dans notre contribution, le tier qui contourne un autre tier se voit momentanément donné le droit (exclusif et avec les bons numéros de séquence) d'émettre sur la connexion TCP qui précède le tier contourné.

Cette notion de raccourci a donné lieu à deux prototypes :

— DISC (Distributed Shared Connection) est un prototype implanté dans un environnement

Linux, prenant la forme d'extensions noyau pour filtrer et adapter des paquets TCP, et d'adaptations des serveurs applicatifs. Il permet d'implanter des raccourcis dans des applications avec de très légères modifications, pour des données finales retournées par des tiers.

— ODB (On Demand Buffer) est un prototype également implanté dans un environnement Linux, qui montre comment on peut s'affranchir de toute modification dans les applications pour implanter cette notion de raccourci. ODB s'appuie sur la notion de buffer à la demande reposant sur les mécanismes de protection mémoire. Une donnée reçue par un tier dans un buffer est reçue virtuellement. Si l'application accède au buffer, une faute de page permet de récupérer effectivement la donnée et il n'y a pas de raccourci. Si l'application n'accède pas au buffer, alors la donnée reste vituelle est ne passe pas au travers de ce tier (il y a donc raccourci).

Ces deux prototypes ont été évalués et montrent (1) des diminutions significatives de charge CPU sur les tiers qui sont contournés par des raccourcis et (2) qu'il est possible d'implanter ces optimisations sans modification des applications.

1.4 Plan du rapport

La suite du document est structurée comme suit. Le chapitre 2 présente les différents travaux de recherche en rapport avec notre thème de recherche. Les travaux présentés sont regroupés en deux sections. La section 2.1 concerne les techniques d'optimisations de copie des données et la section 2.2 concerne les techniques d'optimisation de communications réseaux. Les deux contributions majeures (DISC et ODB) de mes travaux sont respectivement consignées dans les chapitres 3 et 4 dans lesquels nous détaillons les motivations, la conception, les details d'implémentations et les evaluations. Enfin, le dernier chapitre conclut nos travaux et présente les perspectives à court et long terme pour les futurs travaux.

2
Etat de l'art

Ce chapitre présente les techniques d'optimisation de copies de données en environnement centralisé et les techniques d'optimisation de transferts de données en environnement distribué.

Sommaire

De nombreux travaux dans la littérature scientifique et dans le monde de l'opensource se sont penchés sur les mécanismes d'optimisation des communications réseaux. Nous nous intéressons ici aux mécanismes qui visent à implanter une forme de raccourci. Ces raccourcis peuvent être implanté en centralisé (sur une seule machine) et ils visent alors une limitation des copies de données dans les couches systèmes (protocoles de communication). Les raccourcis peuvent être implantés en distribué et ils visent alors un contournement de machines comme nous le faisons dans nos contributions. Dans les deux cas, les travaux que nous avons identifiés donnent des points de comparaison intéressants pour nos contributions.

Nous regroupons donc les travaux existants sous deux catégories : l'approche centralisée et l'approche distribuée.

2.1 Approche centralisée

La problématique de copie de données entre les différentes couches du système a suscité dans la littérature scientifique plusieurs techniques permettant d'optimiser ou de supprimer les copies inutiles. Même si ces techniques ne concernent pas de près la problématique abordée dans ce mémoire, les notions abordées sont transposables dans les environnements distribués. Nous les mentionnons ici à titre de rappel pour mieux cerner le sujet et avoir ainsi une vue d'ensemble. Ces techniques peuvent être regroupées en trois catégories : Zero-copy, Kernel bypass et IO batching.

2.1.1 Zero-copy

Zero-copy [TK95, CGY01, Sta03] désigne l'ensemble des mécanismes permettant de minimiser les copies des données entre les zones mémoires, qui interviennent lors du transfert de données. Si on considère une application qui veut transférer le contenu d'un fichier vers une socket, ce transfert pose deux problèmes à savoir la commutation de contexte entre espace noyau et espace utilisateur et la copie des données à plusieurs niveaux. On peut dénombrer 4 étapes de copie : du disque vers le cache système (1), du cache système vers une mémoire tampon de l'espace utilisateur (2), de la mémoire tampon utilisateur vers la mémoire tampon de l'espace noyau associé à la socket de l'application (3) et enfin de la mémoire tampon de la socket vers la mémoire interne de la carte réseaux (4). Les étapes de copie (2) et (3) peuvent être considérées comme de trop si l'application n'utilise pas les données à transférer. Il existe plusieurs façons de répondre à ce problème. L'application peut déléguer le transfert de la donnée au noyau, on parle dans ce cas de copie intra noyau. C'est le cas des appels systèmes sendfile, splice et mmap qui vont supprimer les étapes (2) et (3). L'application peut directement accéder au matériel, on parle dans ce cas de kernel bypass que nous détaillons plus bas. L'application peut optimiser les transferts de données en réduisant les commutations de contexte grâce aux modèles système scatter/gather et du IO batching. Enfin, on peut citer le partage de mémoire entre l'espace noyau et l'espace utilisateur qu'on peut qualifier de correspondance dynamique par l'utilisation de l'appel système mmap ; cette technique est utilisée par les solutions telles que packet_mmap, pf_ring, netmap.

linux sendfile est un appel système Linux dont le but est de déléguer au noyau le transfert d'un fichier à partir d'un descripteur de ce fichier vers un descripteur de socket. Ceci est possible si le fichier peut être mappé en mémoire. La primitive sendfile est largement utilisée dans les projets opensource comme nginx, haproxy, lighttpd et apache2. Généralement, lorsqu'on veut transférer le contenu d'un fichier vers une socket, une façon de faire consiste à lire le fichier segment par segment et à écrire segment par segment sur la socket : c'est le pattern read/write. Cette façon de faire nécessite plusieurs appels systèmes et commutations de contexte entre l'espace utilisateur et

l'espace noyau. Une amélioration de ce pattern, c'est la lecture vectorisée et l'écriture vectorisée (readv/writev) ; ici, un appel à readv permet de lire d'un coup plusieurs segments du fichier et réciproquement, un appel a writev permet d'écrire plusieurs segments sur la socket. On réduit ainsi le nombre d'appels systèmes et de commutations de contexte. Toutefois, les copies entre l'espace noyau et l'espace utilisateur ne sont pas supprimées. sendfile permet donc de supprimer ces allers et retours et copies. DISC s'inspire de sendfile dans son implémentation que nous détaillons dans la section 3.5.4

splice est similaire à sendfile, mais lève la contrainte sur le descripteur source qui devait être un fichier. Il est implémenté suivant le principe du type abstrait de donnée FIFO (First In First Out). En effet la structure de donnée sur laquelle s'appuie l'appel système est constituée d'un ensemble de pages mémoires physiques de 4 kilos et d'un ensemble de métadonnées permettant de référencer les données. Cet ensemble constitue donc un canal de communication intra noyau avec lequel deux processus de la même machine peuvent communiquer sans qu'il y ait copie de données entre les deux, mais tout simplement par échange de pages mémoires du canal de communication.

2.1.2 kernel bypass

Nous appelons kernel bypass l'ensemble de techniques utilisées le plus souvent pour les opérations d'entrées-sorties réseaux et dont le but est de court-circuiter certaines couches basses du noyau (le plus souvent la couche TCP) ou limiter le nombre d'appels systèmes afin d'améliorer les performances. On retrouve les mécanismes systèmes memory mapped io (mmio) qui permettent de donner l'accès à la mémoire du matériel en passant par la mémoire centrale et userspace driver (uio) qui rend possible l'implémentation des drivers des cartes PCI dans l'espace utilisateur. Intel dpdk utilise la technique uio pour ses drivers dans l'espace utilisateur. Nous nous intéressons dans cette section aux solutions permettant de réaliser le court-circuit de la pile réseau que ce soit en réception ou en émission de paquets. La liste n'est pas exhaustive, mais ceux sélectionnés permettent de comprendre le mécanisme.

packet_mmap [pac14] est un module noyau qui étend l'API socket avec une nouvelle famille de protocole AF_PACKET ou PF_PACKET et qui permet surtout d'optimiser le processus de capture de trames réseaux. Son implémentation s'appuie sur la primitive mmap et définit un espace mémoire noyau partageable avec l'espace utilisateur dans lequel les paquets réseaux bruts peuvent transiter. Il donne donc à cet effet la possibilité pour une application dans l'espace utilisateur de recevoir ou d'envoyer les paquets réseaux bruts.

socket AF_XDP est une refonte de packet_mmap dans sa version 4. Il étend l'API socket par l'ajout d'une nouvelle famille de protocole AF_XDP et il permet de faire un zero copy depuis les buffers (DMA) de la carte réseaux vers l'espace utilisateur. Son implémentation est mise en œuvre par une combinaison de eBPF [VCP+20] et son extension XDP [VCP+20]. eBPF donne la possibilité à une application d'exécuter du code dans une machine virtuelle embarquée dans le noyau sans nécessiter une recompilation du noyau. XDP quant à lui est un ensemble de points d'attaches sur lesquels on peut injecter du code dynamiquement. AF_XDP permet par exemple de router une partie du trafic réseau depuis la carte réseau ou le driver réseau vers l'application. La mise en œuvre de AF_XDP s'appuie sur le mécanisme de liste circulaire (ring buffer) avec modèle sans verrou 1-producteur/1-consommateur (SPSC). Quatre ring buffers (rx, tx, fq, cq) qui pointent sur une zone mémoire partagée (umem) sont nécessaires pour son fonctionnement. L'application et le sous système AF_XDP jouent les rôles de producteur/consommateur sur ces rings pour passer les données. Dans notre contribution, nous nous appuyons sur les sockets AF_XDP pour réaliser la transmission de données.

netmap [Riz12] combine principalement trois techniques, l'allocation statique de ressources pour la gestion des paquets, la mémoire partagée pour le partage des informations entre l'espace utilisateur et le noyau et enfin le traitement par lot des paquets réseaux. Son implémentation s'appuie sur un module noyau qui déconnecte la carte réseau de la pile réseaux et la relie à l'espace utilisateur via un jeu de ring buffer et une API qui permet d'initier l'envoi ou la réception des paquets.

pf_ring [pfr08] est très similaire à netmap et résoud le même problème à la seule différence que netmap s'appuie sur les appels systèmes standards tandis que pf_ring fournit une API non standard.

dpdk [dpd10] est une suite d'outils proposés par Intel et qui implémente les pilotes matériels dans l'espace utilisateur. Il repose sur la fonctionnalité uio [uio06] fournie par le noyau. uio va permettre d'initialiser le matériel et exposer les registres et la mémoire du matériel dans l'espace utilisateur ce qui donne donc la possibilité de piloter le matériel depuis l'espace utilisateur.

2.1.3 IO batching

IO batching désigne un ensemble de mécanismes permettant d'effectuer le traitement par lot. On retrouve ici les appels systèmes readv, writev, preadv, pwritev, linux io_uring qui implémentent ce principe ; netmap, packet_mmap, dpdk et pf_ring utilisent aussi ce principe pour améliorer les performances.

linux io_uring [iou19] est une nouvelle interface Linux qui permet l'envoi et la réception des données de façon asynchrone. Il s'appuie sur le principe de complétion. En effet, l'application soumet une tâche, cette tâche renfermant les paramètres effectifs que devraient avoir un appel système. L'application est notifiée ultérieurement lorsque la tâche se termine d'où son modèle asynchrone. Son implémentation s'appuie sur les incontournables rings buffers. io_uring apporte une touche asynchrone à la gestion des entrées-sorties, fonctionnalité qui existait déjà avec aio (asynchronous I/O), mais qui souffrait d'une limitation principale due au fait qu'elle n'était applicable que pour les accès non bufferisés et que permettait l'option O_DIRECT.

2.1.4 zIO

Nous détaillons ici zIO, une récente étude qui utilise le mécanisme de pagination afin de rendre transparent le zero copy pour les applications.

zIO, part du constat que les applications le plus souvent ne modifient qu'une partie des données qu'elles reçoivent. Ces applications copient inutilement les données, alors qu'elles ne sont utilisées qu'en partie. Cet aspect cause un gaspillage de ressources (augmente de la charge CPU) comme évoqué précédement.

zIo propose une bibliothèque qui permet d'intercepter les appels systèmes (read, write, memcpy, memmove) d'entrées-sorties afin de capturer les accès mémoires d'une application. L'idée de base consiste à déréférencer la zone mémoire intermédiaire (paramètre dest de la fonction memcpy) et d'ignorer la copie. La zone mémoire déréférencée est protégée et enregistrée dans un gestionnaire de faute de page grâce à la fonctionnalité userfaultfd de Linux. Fait ainsi, toute tentative d'accès de la zone provoque un défaut de page. zIo peut donc prendre la faute de page et résoudre le défaut de page, qui consiste à réaliser la copie ignorée. Ceci permet de ne copier que les données dont l'application a effectivement besoin. zIo permet d'améliorer les performances des applications réseaux qui stockent les données reçues dans une mémoire non volatile (NVM). Notre approche ODB est similaire, car basée sur le même principe, à la différence que la résolution des défauts de page se produit dans une machine distante du réseau et nous voulons limiter les copies pour les **payload** en transit (l'application ne stocke pas nécessairement la donnée). zIo, supprime l'aspect intrusif

du zero copy par l'utilisation de la pagination et les données servant à résoudre les défauts de page sont locales. zIo ne permet pas d'outrepasser un tiers. Il applique un splice transparent entre les couches d'entrés-sorties au sein d'un tiers pendant le traitement de la donnée. Notre approche est plus générale et concerne les architectures multi-tiers distribuées. Dans cette architecture, nous voulons exempter un tiers intermédiaire de toute donnée non utile de façon transparente. L'originalité de notre approche réside dans le fait que nous envoyons la donnée virtuellement et grâce à la pagination, nous ramenons uniquement les données utilisées par le tiers intermédiaire.

2.1.5 synthèse

On constate que plusieurs efforts sont faits pour améliorer les performances dans le cadre du transfert de données en centralisé. L'objectif commun est de supprimer les copies inutiles de données et d'éliminer les aller-retours noyaux en procédant par le traitement par lot. L'élimination des copies passe soit par le mécanisme de partage de buffers noyaux avec l'espace utilisateur, soit par l'accès direct de la mémoire et des registres (mmio) du matériel. Dans notre contribution, nous adressons ce problème en distribué.

2.2 Approche distribuée

Dans un environnement distribué, plusieurs machines ou nœuds de traitement spécialisés peuvent être mis ensembles pour rendre un service ; on parle alors d'architectures multi-tiers. Le traitement d'une requête dans ce type d'architecture peut traverser toute une hiérarchie de serveurs. Ceci augmente les temps de latence et baisse le débit pour les transferts de données, et induit une charge CPU significative dans toute la hiérarchie. Plusieurs travaux se sont penchés sur le sujet afin de réduire ces aspects négatifs. Les techniques utilisées dans ces travaux peuvent être regroupées en 5 catégories : la migration de connexion (tcp hand-off), le déport de traitement (tcp-offload), la translation des segments (tcp-splice), la redirection (redirection HTTP, CDN), le retour direct effectué par le serveur (DSR). Nous étudions dans les sections ci-après ces mécanismes tout en décrivant leur mode de fonctionnement ou implémentation et leur domaine d'application. Dans la section synthèse, nous mettons en lumière les limitations de ces mécanismes tout en indiquant notre apport par rapport à ces mécanismes.

2.2.1 tcp-offload

tcp-offload [SH04], illustré sur la Figure 2.1, est le concept employé pour désigner l'ensemble des techniques qui permettent de déporter dans une carte réseaux toute la pile réseau ou une partie des actions de la pile réseau comme le calcul de somme (CSO), la réception et l'assemblage des paquets en large segment de données (LRO), la segmentation et transfert de larges segments de données (TSO). Alors que ce mécanisme permet de réduire le nombre de cycles du CPU pour le traitement d'un paquet [SH04], la carte quand à elle devient le goulot d'étranglement [KR06]. Considérée au départ comme une idée stupide [Mog03], par le fait que sa mise en œuvre est compliquée et surtout aussi que les ressources de la carte sont limitées [lin05], le tcp-offload refait surface grâce aux avancées technologiques concernant la conception des cartes réseaux (Broadcom NetXtreme, Intel 10Gb) et de nombreuses recherches [MLJP20a, GYLW20, S+21, SSKP22a] qui ont permis d'améliorer la technique d'offloading. Notons ici que cette technique reste centralisée (elle est distribuée entre le CPU et la carte réseau) et ne permet pas de distribuer la charge. Une machine équipée de ce mécanisme ne pourra supporter la charge s'il y a un afflux de données considérable.

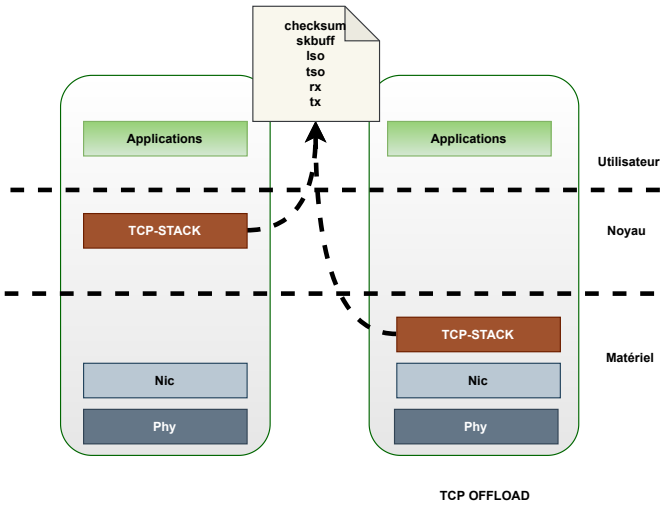


FIGURE 2.1 – *tcp offload*

2.2.2 TCP handoff

tcp_handoff, décrit pour la première fois par [Dru99], est le terme utilisé pour désigner l'ensemble des techniques permettant le transfert d'une connexion TCP/IP d'une entité de traitement A vers une entité de traitement B. Ces entités sont généralement des machines sur le réseau, mais il peut aussi s'agir d'un transfert de connexion vers une carte réseau incluant une pile TCP/IP (comme dans le tcp-offload ci-dessus). Le handoff peut être exploité pour équilibrer la charge entre des machines. Notamment, il est possible de sélectionner un sous ensemble de connexions d'une pile TCP/IP à transférer [KR06] ; on parle alors de handoff partiel. Pour implanter le handoff, sérialiser (sauvegarder et restaurer) la connexion reste la pierre angulaire.

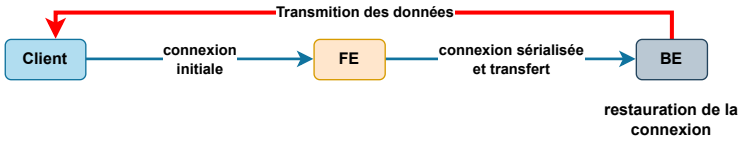


FIGURE 2.2 – *tcp handoff*

Considérant une architecture multi-tiers comme sur la Figure 2.2 où FE et BE font partie du même cluster et ou Client demande une ressource à FE détenue par BE, le TCP handoff consiste à migrer une connexion TCP établie entre le Client et le tier FE vers le tier BE détenant la ressource. L'idée de base consiste à sérialiser la connexion établie entre Client et FE chez FE puis à transférer l'état de la connexion chez BE qui va désérialiser et restaurer la connexion, permettant ainsi de créer un canal de communication direct entre Client et BE. Sa mise œuvre s'appuie entre autre sur TCP_REPAIR, fonctionnalité du noyau introduite à partir de la version 3.5.x et les mécanismes de translation d'adresses IP (NAT). Cette approche permet de baisser le flux de données traversant FE, ce qui a pour conséquence de baisser la charge CPU de FE. Bien que ce mécanisme soit transparent pour le client dans certain cas, son implémentation nécessite une modification des couches basses des serveurs du cluster impliqués dans le traitement de la requête en cours et une modification du switch [HHSE21] ou la gateway. Suivant les implémentations, FE peut totalement être écarté [KIB20] de la communication après que le handoff ait eu lieu ou continuer à relayer les acquittements dans la couche 4 (comme un loadbalancer de couche 4). Lorsque le FE est totalement écarté après que le handoff ait eu lieu, c'est le switch ou la gateaway moyennant des règles de routage qui va servir de relais entre Client et BE. A noter que dans sa définition initiale, TCP handoff ne permet que de migrer une connexion TCP lors de son établissement, bien que rien

n'empêche d'envisager une migration à tout moment.

2.2.3 TCP splice

TCP-splice est le terme utilisé pour désigner l'ensemble de techniques permettant à un serveur applicatif (généralement les proxies web) de relayer une partie du trafic réseau en transit soit à partir de la couche transport [MB99], soit à partir de la couche réseau ou soit directement dans la carte réseau [MLJP20b]. Il est utilisé dans le cas où les données ne sont pas exploitées au niveau applicatif (par exemple dans un proxy pour implanter un cache des données). A noter que certaines données peuvent être relayées dans des couches basses et d'autres remonter au niveau applicatif.

L'objectif de base dans le splicing est d'éviter la copie des données entre les deux extrémités de connexions (socket). Une façon de mettre en œuvre le TCP splice est de modifier les entêtes des paquets réseaux [SHHP00, SSKP22b] et de réemettre. Pour un proxy cache, d'après [KRR02], le splicing améliore les performances pour les données qui normalement remontent au niveau applicatif et ne sont pas cachées (elles bénéficient du splicing), mais cela peut être au détriment des données qui sont normalement servies depuis le cache (en raison du trafic réseau).

2.2.4 Redirection HTTP

La redirection HTTP est le mécanisme qui permet à un serveur web de dire à un client que la ressource demandée se trouve à un autre emplacement. Sa mise en œuvre nécessite un catalogue qui détient la cartographie des ressources situées à différents emplacements. Lorsqu'un client se connecte, le serveur consulte son catalogue et envoie une réponse constituée d'un code retour (généralement 302) et un entête spécial *Location* qui pointe vers l'emplacement effectif de la ressource demandée. À la réception de cette réponse, le client se connecte à l'emplacement spécifié puis récupère la ressource. La redirection HTTP permet de répondre à la problématique de la distribution efficace des données en les répliquant au plus proche des clients [SH02]. Une utilisation possible de la redirection HTTP (qui est largement déployée) est le réseau de distribution de contenu (CDN) que nous détaillons dans 2.2.5. Le CDN est utilisé par de grands acteurs d'internet tels que Google, Netflix, Cloudflare, Amazon CloudFront. Cependant, un des inconvénients majeurs vient du fait que le client effectue deux ouvertures de connexion pour avoir la ressource demandée, ce qui peut être gênant pour des applications qui nécessitent des temps de latence réduits.

2.2.5 Réseau de diffusion de contenu

Un réseau de diffusion de contenu (en abrégé CDN) [Pas12] est un ensemble de serveurs applicatifs interconnectés par un réseau et géographiquement dispersés dont la finalité est de mettre à la disposition des clients du contenu statique tel que les images ou des flux en diffusion continue (vidéo) en fonction de leur proximité géographique. On distingue donc à cet effet deux types de CDN : mise en cache et flux de diffusion. Le CDN est mis en œuvre avec un ensemble de serveurs dont l'un joue le rôle de serveur d'origine et les autres le rôle de serveur de réplication (Figure 2.3). L'algorithme de réplication se charge de répliquer les données vers les serveurs réplicas et le serveur d'origine sert de point d'entré du CDN. Lorsqu'un client sollicite un contenu, il transmet sa requête au serveur d'origine. Le serveur d'origine grâce à un mécanisme de routage sélectionne un réplica à proximité du client et envoie une redirection au client. Ce dernier se connecte à nouveau avec le réplica et récupère le contenu. Il existe une variante hybride [YLZ+09] du CDN où les serveurs réplicas constituent un réseau peer-to-peer permettant ainsi à plusieurs serveurs de participer à la distribution des segments du même fichier.

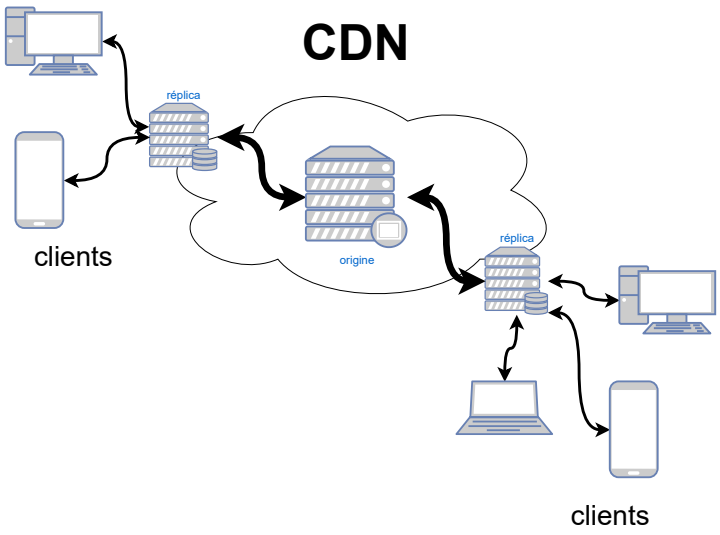


FIGURE 2.3 – *Réseau de diffusion de contenu*

Le CDN [BCT03] répond à une problématique cruciale qui est celle de permettre aux fournisseurs de contenu de distribuer du contenu (image, vidéo..) de façon efficiente. L'enjeu ici est d'éliminer le maximum d'intermédiaire entre le client et l'emplacement de la ressource et donc de minimiser la latence [KMS+09].

Malgré ses avantages (scalabilité, proximité de la ressource), le client effectue deux connexions pour avoir la ressource, une connexion vers le serveur d'origine et une deuxième lors de la redirection pour enfin avoir accès à la ressource. La mise en œuvre est couteuse, car il faut déployer et maintenir plusieurs serveurs.

2.2.6 DSR

Considérant un déploiement comme on peut le voir sur la Figure 2.4 où le frontend route une requête vers un backend, DSR désigne l'ensemble des techniques permettant dans ce type de déploiement de faire en sorte que le backend puisse envoyer la réponse directement au client via une gateway. Il existe plusieurs implémentations dans le monde open source [hap11, ngi] et dans la littérature scientifique [PBY+13, DUB+11].

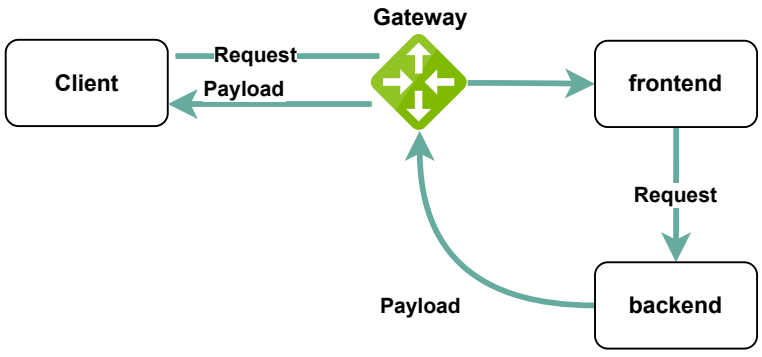


FIGURE 2.4 – *Direct server return*

[hap11] implémente un loadbalancer de couche 4 où le loadbalancer route une requête entrante vers un backend sélectionné en remplaçant l'adresse MAC destination par l'adresse MAC du serveur selectionné. Le frontend est configuré avec une adresse réseau virtuelle (VIP) sur son interface publique, tandis que les backends sont configurés avec la même adresse VIP sur l'inter-

face privée (boucle locale). Chaque backend est configuré de telle sorte qu'il ne puisse pas faire une résolution d'adresse (RARP) pour éviter les conflits. Les réponses émises par les backends sont envoyées avec l'adresse virtuelle VIP comme origine. Cette implémentation est pertinente lorsqu'aucune intelligence n'est requise par exemple au niveau de la couche présentation (L7).

Les implémentations [PBY+13, DUB+11] sont assez similaires et implémentent une version avancée de DSR dans un environnement de Cloud virtualisé. La Figure 2.5 illustre le principe de fonctionnement. Les backends sont des machines virtuelles exécutées dans des hyperviseurs embarquant un agent qui implémente DSR (Nat destination vers les VMs et Nat source vers Router). [PBY+13] se différencie du DSR traditionnel, car la transformation des paquets n'est plus opérée au sein du loadbalancer mais au sein des agents déployés dans les hyperviseurs. Le loadbalancer (MUX, d'adresse VIP) détient la cartographie de tous les DIP (Destination IP) des VMs dont il a la responsabilité de répartir la charge. Lorsque MUX reçoit un paquet venant d'un client (d'adresse CIP), il selectionne une DIP et encapsule le paquet entrant dans une trame (IP-In-IP) comme on peut le voir sur la Figure 2.5. Le paquet envoyé par le client, qui avait une entête avec des adresses origine-destination CIP-VIP, se retrouve encapsulé dans un paquet avec une entête VIP-DIP. L'agent qui intercepte le paquet sur l'hyperviseur de la VM destination, retire l'entête IP-In-IP et fait un Nat destination avec la DIP de la VM spécifiée dans l'entête IP-In-IP. Ainsi, la VM reçoit le paquet comme s'il venait directement du client. Lorque la VM répond, l'agent intercepte la réponse et fait une Nat inverse de la source avec la VIP, pour que le client voit toujours le MUX comme origine de la réponse.

Toutes ces implémentations de DSR au vu de l'état de l'art actuel ne permettent pas de faire des traitements sur la couche 7 au niveau du loadbalancer. De plus, le frontend ne peut plus servir de terminaison SSL (SSL offload). Enfin, si on interpose un tiers entre le frontend et le backend, il serait difficile de contourner le tiers intermédiaire.

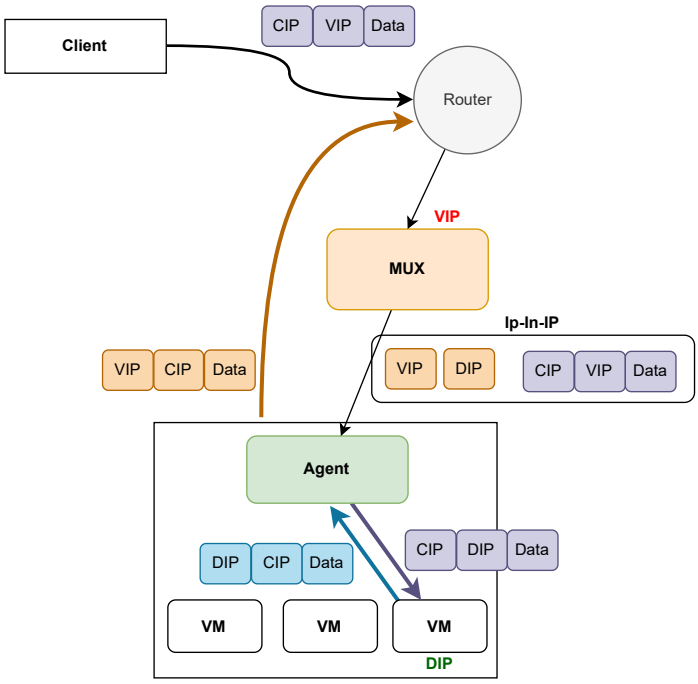


FIGURE 2.5 – *Ananta direct server return*

2.2.7 Prism

Prism [HHSE21] s'appuie sur le principe TCP handoff et permet de migrer une connexion TCP d'un tiers vers un autre tiers du data center. Par rapport au handoff présenté précédemment,

Prism permet de migrer une connexion TCP à tout moment.

La migration de la connexion (entre deux tiers frontend et backend) est mise en œuvre en utilisant le principe sauvegarde et restauration d'une connexion TCP déjà établie grâce la fonctionnalité fournie par le noyau Linux tcp_repair [Cor12] d'une part, et d'autre part, moyennant une reconfiguration dynamique du switch pour rediriger le trafic vers le backend sélectionné par le frontend. Pendant la phase de transfert, le switch est configuré pour bloquer tout le trafic entrant pour la session en cours. Une fois le handoff terminé, le switch est à nouveau reconfiguré pour laisser passer le trafic. À ce stade, un lien direct est établi entre le backend et le client. Avec Prism, le backend collabore avec le frontend pour lui permettre de distribuer la charge. En effet lorsque le backend a fini le traitement de la requête, il transfert l'état de sa connexion au frontend, lui permettant de sélectionner un autre backend s'il reçoit une nouvelle requête, pour mieux répartir la charge. Notons ici que la réutilisation de connexion avec le backend par le frontend pour le traitement des requêtes suivantes n'est possible que si on est en HTTP/1.1. Dans un handoff traditionnel, le backend ne redonne pas le contrôle au frontend. Enfin, Prism fournit une API d'adaptation qui permet au serveur applicatif de contrôler les phases de migration de connexion et donc de s'interfacer avec Prism.

2.2.8 Crab

Crab [KIB20] est une alternative de loadbalancer couche 4 qui implémente un handoff par extension du handshake entre un client et un serveur. La Figure 2.6 illustre le principe de fonctionnement. Contrairement à l'implémentation du handoff de Prism, le frontend dans le cas de Crab ne crée pas la connexion entrante, mais permet de la créer dans le backend.

Le client est modifié pour fournir l'option REDIR_OPTION dans la couche TCP. Ce champ contiendra l'adresse virtuelle du service auquel il tente d'accéder. Le frontend est configuré avec une adresse virtuelle VIP et le backend avec une adresse de destination DIP.

L'ouverture de la connexion se passe de la manière suivante : le client émet un paquet syn vers le frontend, ce paquet contient en plus l'option REDIR_OPTION. Lorsque le frontend reçoit un tel paquet, il met à jour le champs REDIR_OPTION avec sa VIP puis selectionne un backend et remplace l'adresse IP de destination avec celle du backend selectionné. Le paquet syn est ensuite transmis au backend. Le backend acquitte le packet syn (syn-ack) avec son adresse DIP comme s'il provenait directement du client. Lorsque le client reçoit le paquet syn-ack du backend, il recupère l'adresse VIP du champs REDIR_OPTION pour localiser la structure de donnée socket qui avait été allouée lors de l'envoi du paquet syn car sinon il ne s'aurait répondre à syn-ack qui n'a pas de correspondance avec un syn émis au sens IP du terme. La structure trouvée, il peut dès lors acquitter le paquet syn-ack avec un ack, mais cette fois la destination n'est plus VIP mais DIP. A ce stade, le handshake est terminé, et le client et le backend peuvent directement communiquer.

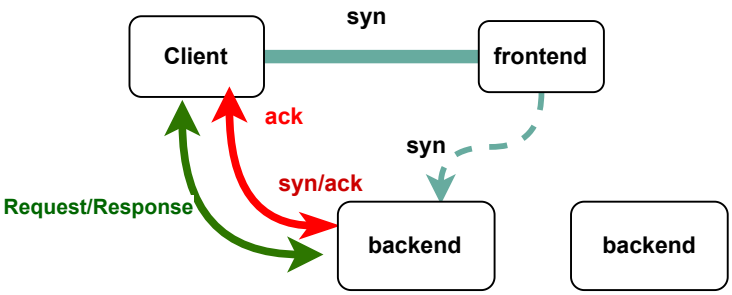


FIGURE 2.6 – *redirection de connexion avec CRAB*

Avec crab, c'est la pile de protocole du client qui indique qu'elle a le support de la redirection et qui active à cet effet l'option REDIR_OPTION, pour que la redirection de la connexion initiale

vers un backend donné puisse se faire. Il se démarque de Prism en ce sens qu'il ne crée pas la connexion sur le frontend, mais la déporte sur le backend. Comme tous les handoffs, il est adapté dans les situations où la redirection ne casse pas une logique applicative et où client et serveur peuvent communiquer directement.

2.2.9 synthèse

La plupart des solutions étudiées s'appliquent aux architectures 3-tiers Client-FE-BE et permettent d'éliminer le bottleneck que constitue le FE. Cependant, elles posent plusieurs problèmes :

— contournement total du FE (Prism, Crab, DSR)

— modification du client ou du protocole TCP (redirection HTTP, Crab)

— Uniformité de protocole

contournement total du FE. Si on considère une architecture multi-tiers Client-FE-IS-BE, un contournement total peut casser la logique applicative si FE ou IS ont besoin d'une partie des données pour un traitement spécifique par exemple exploiter des données contenues dans les entêtes du protocole applicatif. De plus, pour une connexion persistante (keepalive), lorsque le contournement est enclenché, toutes les réponses associées aux requêtes utilisent le même contournement. Il n'est donc pas possible de faire un contournement sélectif (les requêtes arrivent à différents BE ou bien seulement certaines réponses utilisent le contournement) avec les solutions étudiées, sauf en reconfigurant le contournement à chaque requête, ce qui serait très couteux.
Uniformité de protocole. Le contournement n'est possible que si le Client et le BE sont compatibles c'est-à-dire que le BE doit implémenter le même protocole que le Client (par exemple HTTP). Cependant, il y a des architectures où les tiers n'implantent pas tous le même protocole.
Modification du client ou du protocole. Crab nécessitent une adaptation du client. Cet aspect est intrusif et peut donc être un facteur limitant pour une utilisation grand public étant donné qu'on n'a pas la maîtrise sur le client. Notre solution permet d'éviter ces problèmes.

Notre objectif est de mettre en place un mécanisme permettant de contourner tout tier dans une architecture multi-tiers, pour tout ou partie des données transférées. En particulier, lorsque les données transférées comportent une partie entête importante pour les applications et une partie charge utile (payload), il est important de pouvoir utiliser le contournement uniquement pour la partie payload pour ne pas casser la logique applicative. Ce mécanisme doit être indépendant des protocoles applicatifs qui peuvent être différents pour les différents tiers. Enfin, ce mécanisme est interne au datacenter et ne doit pas nécessiter de modifications pour un client externe.

3

Session TCP distribuée

Ce chapitre décrit la conception, les détails d'implémentation et l'évaluation de DISC

Sommaire

3.1 Introduction

Ce chapitre présente la première contribution de la thèse, à savoir DISC. Cette contribution se base sur le constat que de nombreuses applications multi-tiers sont déployées dans les datacenters et que dans ces applications, certaines données générées par un tier sont simplement relayées par les tiers précédents pour être retournées à un client externe au datacenter. Il est alors possible de mettre en place un schéma de raccourci, permettant au tier générant les données de les retourner directement au client externe, ce qui revient à partager la connexion TCP avec le client externe entre tous les tiers de l'application (d'où le nom de DISC pour Distributed Shared TCP). Nous présentons les motivations dans la section 3.2, puis la section 3.3 présente les principaux choix de conception de DISC. La conception de DISC est détaillé dans la section 3.4 puis l'implantation de DISC est décrite dans la section 3.5. Une évaluation de notre prototype est rapportée en section 3.6, puis nous concluons ce chapitre.

3.2 Motivations

3.2.1 Contexte

Nos travaux se situent dans le contexte des plateformes de cloud et des datacenters hébergeant ces plateformes. L'objectif général dans ces plateformes est de fournir des ressources à la demande pour héberger des applications. Le gestionnaire d'une telle plateforme gère les ressources afin d'optimiser leur utilisation et éviter tout gaspillage.

Nous nous intéressons plus particulièrement aux applications fournissant des services en ligne, c'est à dire des services accédés par des clients externes au datacenter. Des exemples de telles applications sont les applications Web ou les applications de gestion de messages électroniques (serveurs IMAP). Ces applications sont accédées par des clients externes au datacenter au travers de protocoles de communication variés (HTTP, IMAP, FTP, etc), s'appuyant généralement sur TCP/IP. Ces protocoles implantent un schéma requête/réponse entre le client externe et l'application hébergée dans le datacenter. Les messages que sont les requêtes et les réponses sont composés d'un entête (header) incluant les paramètres gérés par le protocole et d'un contenu incluant les données gérées par l'application (une page web, une image ou un email dans les deux exemples d'application précédents).

Dans le cadre de nos travaux, nous nous intéressons au contenu (que nous appelons payload) des réponses retournées au client externe, ce contenu pouvant être volumineux. Ces travaux peuvent être toutefois généralisés aux échanges de données dans les deux sens (requête et réponse).

3.2.2 Position du problème

Parmi ces applications hébergées dans des datacenters, de nombreuses adoptent une architecture multi-tiers. Cela signifie que le service implanté par l'application est composé de plusieurs composants applicatifs (appelés tiers), s'exécutant sur différentes machines du datacenter, et communiquant également aux travers de protocoles de communication variés. Ces tiers implantent tous un schéma requête/réponse. La Figure 3.1 montre l'architecture multi-tiers des deux exemples d'application évoqués précédemment. A gauche est représentée l'architecture de l'application Moddle [RW06] (Web) et à droite l'achitecture de l'application Zimbra [zim02] (messagerie). Une requête est reçue du client externe par un tier frontal (généralement un répartiteur de charge ou load-balancer) et cette requête est propagée à un autre tier. Le traitement d'une requête par un tier peut nécessiter d'appeler un autre tier, chaque tier implantant une partie du service

global fourni par l'application.

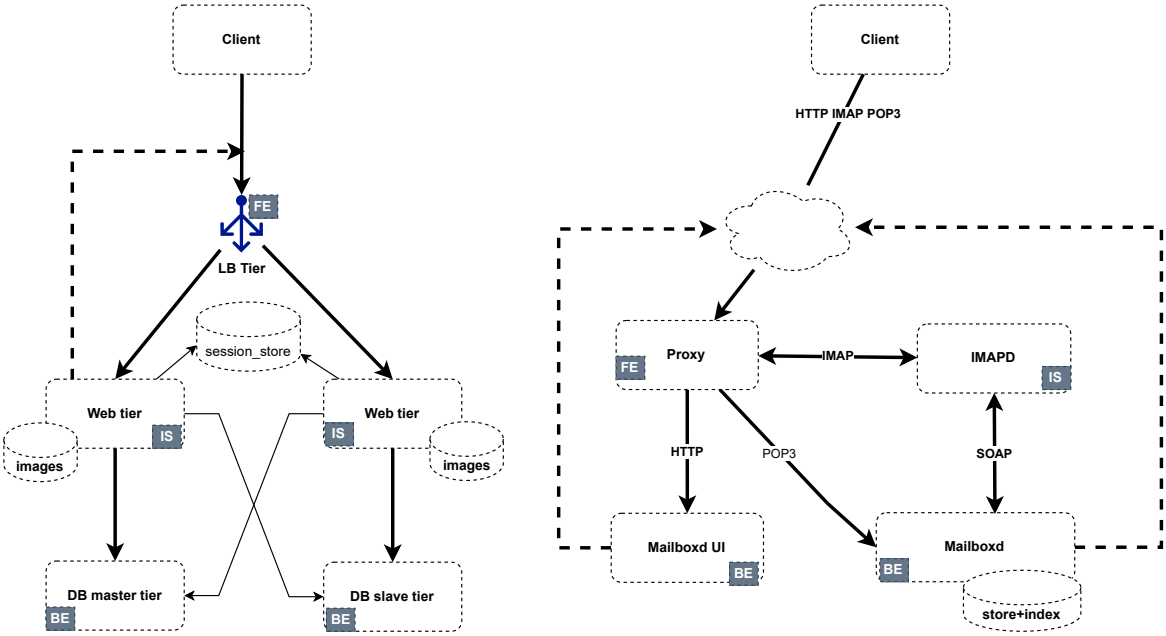


FIGURE 3.1 – *Moddle, Zimbra*

Dans ce contexte, nous observons que très souvent, une requête est propagée du tier frontal (ayant reçu la connexion du client externe, nous l'appelons Front End ou FE) à un dernier tier (qui n'appelle pas d'autre tier, nous l'appelons Back End ou BE) dans l'architecture multi-tiers, en passant par différents tiers intermédiaires (que nous appelons Intermediate Servers ou IS). Ce chemin suivi par la requête est représenté sur la Figure 3.1 (fléches noires). Et également souvent, le BE génère une payload (ou la lit depuis un fichier) qui est ensuite retournée au tier précédent et propagée en retour de tier en tier jusqu'à FE qui retourne cette payload au client externe. Nous disons que cette payload retournée par BE est finale lorsqu'elle n'est pas exploitée (lue ou modifiée) par les IS ou par FE et qu'elle est retournée au client externe telle quelle. Les tiers IS et FE ne font que relayer la payload respectivement vers le tier précédent ou vers le client externe. A titre d'exemple, dans l'application Web mentionnée précédemment, une image retournée par un tier n'est pas exploitée par les tiers précédents dans l'architecture multi-tiers.

Cette observation montre qu'il y a alors des communications inutiles lorqu'une payload finale est seulement relayée par les tiers IS et FE. La Figure 3.1 illustre le trajet suivi par une payload finale dans le cas de l'application Web, lorsque chaque tier est hébergé sur un serveur différent. Les tiers IS et FE utilisent une quantité significative de ressources (CPU, mémoire, I/O) pour seulement recevoir et renvoyer la payload finale vers le tier précédent. Sur la figure, la flèche en pontillé matérialise l'objectif qui est de retourner la payload finale au client externe.

3.2.3 Objectif

L'objectif des travaux présentés dans ce chapitre est de permettre la mise en place de raccourcis (flèches en pontillé sur la Figure 3.1). La mise en place de ces raccourcis ne doit pas nécessiter une refonte des applications (les tiers) déployées dans le datacenter. La contribution présentée dans ce chapitre ne demande que des modifications très légères dans les applications et nous montrons dans le chapitre suivant comment on peut mettre en place ces raccourcis sans modifier les applications.

Les bénéfices attendus de ces raccourcis sont principalement l'économie de ressources dans le datacenter. Les serveurs hébergeant les tiers IS et FE verront leur utilisation de ressources réduites significativement (principalement le CPU) et le trafic réseau dans le datacenter sera également

significativement réduit. L'ambition est également d'implanter ces raccourcis sans dégrader la latence ou le débit des communications avec le client externe.

3.3 Choix de conception

Nous décrivons maintenant les choix de conception de notre contribution DISC.

3.3.1 Définitions

Dans ce chapitre, nous utiliserons les concepts ci-après :

architecture multi-tiers
: Désigne une application déployée selon une architecture composée de plusieurs serveurs, chacun fournissant un service particulier. L'exemple le plus répandu est celui des applications web dynamiques.

FE
: Serveur applicatif frontal dans une architecture multi-tiers. Il s'agit du serveur auquel un client externe se connecte. Ce serveur est souvent un répartiteur de charge.

BE
: Serveur applicatif dans une architecture multi-tiers, qui retourne des données à destination du client.

IS
: Serveur applicatif intermédiaire dans une architecture multi-tiers, s'interposant entre le FE et le BE.

payload
: Données produites par un BE à destination d'un client. En particulier, on s'intéresse aux payloads qui sont retournées par un BE sans avoir besoin d'être exploitées par les tiers précédents (IS et FE).

Raccourci
: Désigne le fait de contourner ou passer outre un IS ou FE pour une payload retournée par un BE à un client.

3.3.2 Motivations

La principale motivation pour la conception de DISC est de limiter les modifications sur les logiciels existants :

— `la pile de protocoles`. Notre objectif est d'implémenter DISC sans nécessiter la compilation ou l'installation d'une nouvelle implémentation de TCP, en s'appuyant ainsi sur un système d'exploitation standard. Nous nous appuyons plutôt sur des adaptateurs (dans le noyau Linux) pour intercepter des événements ou fournir des fonctionnalités supplémentaires.

— `serveurs d'application`. Les serveurs d'application désignent ici tout serveur logiciel impliqué dans un raccourci, c'est-à-dire les serveurs qui sont déployés entre la connexion TCP initiale (initiée par le client externe) et le serveur (inclus) qui veut envoyer directement des données sur la connexion TCP initiale, c'est à dire entre le FE et le BE. Les serveurs d'application doivent être adaptés afin de se conformer au raccourci et notre objectif est d'apporter très peu de modifications à ces serveurs.

3.3.3 Les raccourcis

Le principe de base de DISC dans une architecture multi-tiers est que certains messages (les payloads finales) peuvent passer outre un ensemble de serveurs de la hiérarchie. DISC implémente ce mécanisme que nous appelons **Raccourci** au niveau de TCP mais nécessite peu de modifications dans les serveurs d'applications qui implémentent des protocoles de plus haut niveau (par exemple HTTP). Nous ne considérons pas de tels raccourcis pour les messages entrants (requêtes) car cela briserait l'organisation logique de la hiérarchie des serveurs. Pour les messages de retour (réponses), nous distinguons deux parties dans ces messages, la partie contrôle qui comprend généralement des méta-informations (par exemple, l'entête de réponse dans une réponse HTTP) et la partie donnée qui comprend les données de l'application retournées (par exemple, le contenu dans une réponse HTTP). Cette dernière partie, que nous appelons **payload** de la réponse, est celle qui est potentiellement volumineuse et pour laquelle nous voulons appliquer des raccourcis. Notre objectif est donc d'implémenter des raccourcis pour les payloads volumineuses. Les messages de réponse dans les protocoles de niveau application (au-dessus de TCP) comprennent toujours la partie contrôle et elle suit le chemin normal dans la hiérarchie, mais la payload est retirée de la réponse et envoyée directement sur la connexion TCP initiale.

3.3.4 Architecture générale

Nous considérons un raccourci dans l'architecture illustrée par la Figure 3.2. Bien que notre conception soit illustrée avec HTTP sur TCP, elle peut être appliquée à des scénarios d'application impliquant différents protocoles basés sur TCP tels que FTP ou IMAP.

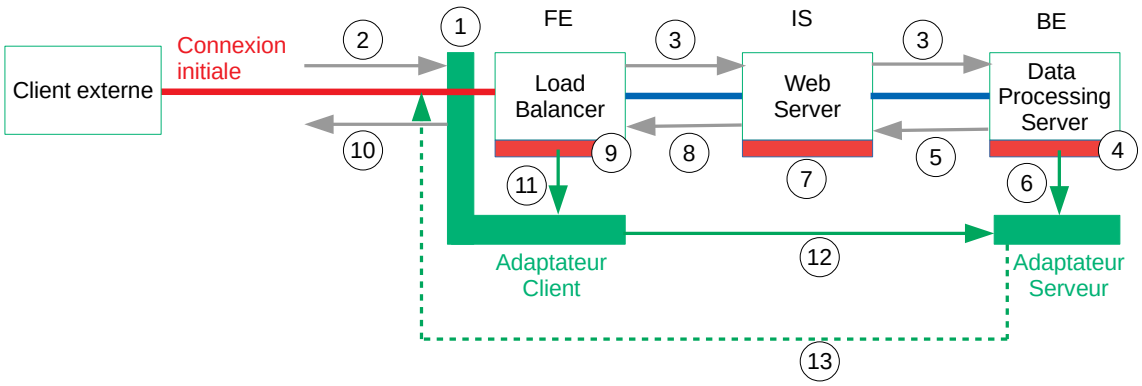


FIGURE 3.2 – *Architecture générale*

La Figure 3.2 illustre l'architecture de DISC. Elle est basée sur des extensions de serveur d'application dessinées en rouge sur la figure et des extensions de noyau dessinées en vert sur la figure. Les extensions du noyau sont uniquement présentes sur la machine qui reçoit la connexion TCP initiale (adaptateur client) et sur la machine qui met en œuvre le raccourci (adaptateur serveur).

Processus de transmission

Le schéma global mis en œuvre par DISC suit les étapes numérotées sur la Figure 3.2.

1. Établissement de la connexion. L'adaptateur client surveille les connexions TCP entrantes sur le FE et enregistre leurs caractéristiques (IP et port du client externe), ce qui permet de les transmettre aux autres serveurs d'application afin qu'ils puissent émettre des messages TCP dans la session.
2. Réception d'un message de requête sur la connexion TCP initiale. Ce message suit le chemin normal et est livré au FE.
3. Le message de requête est transmis par le FE au IS et par le IS au BE.

4. Le BE traite la demande, en générant un contenu (partie donnée ou payload) et un entête (partie contrôle). Le code du BE est adapté pour renvoyer l'entête avec le chemin normal, mais il envoie la payload avec un autre chemin (le raccourci).
5. L'entête est renvoyé sans la payload.
6. La payload est envoyée à l'adaptateur serveur qui met en œuvre une stratégie de mise en mémoire tampon (voir plus loin).
7. Le IS reçoit la réponse du BE, qui ne contient pas la payload. Le code du IS est adapté pour traiter uniquement l'entête (pas de payload) et pour renvoyer à nouveau une réponse sans payload. Notez que dans une hiérarchie plus profonde, tout serveur intermédiaire entre le FE et le BE se comporte de la même manière que ce IS.
8. La réponse est transmise au FE.
9. Le FE reçoit la réponse du IS. Le code du FE est adapté pour traiter l'entête uniquement (pas de payload), pour renvoyer une réponse sans payload et pour invoquer l'adaptateur client pour demander l'émission de la payload.
10. L'entête est retourné au client.
11. L'adaptateur client est invoqué pour demander l'émission de la payload.
12. L'adaptateur client appelle à son tour l'adaptateur serveur pour demander l'émission de la payload.
13. L'adaptateur serveur envoie la payload directement sur la connexion TCP initiale.

Adaptation des applications

Afin de mettre en œuvre les raccourcis, nous devons modifier les serveurs d'application qui se trouvent entre la connexion TCP initiale et le serveur émettant la payload (c'est-à-dire le FE, le IS et le BE dans la Figure 3.2). Notons que dans une architecture plus complexe, il pourrait y avoir plusieurs IS. Tout d'abord, pour les IS, le code doit être adapté pour recevoir et envoyer des entêtes de réponse HTTP sans payload (Figure 3.2 - étape 7). Une propriété (appelée DISC) dans l'entête de la réponse HTTP est ajoutée (par le BE) pour indiquer que la payload fait l'objet d'un raccourci, afin que tout serveur intermédiaire sache que la payload d'une réponse a été omise. D'autres propriétés sont ajoutées et seront détaillées plus tard.

Ensuite, pour le FE, l'adaptation du code (Figure 3.2 - étape 9) est très similaire à ce qui précède, sauf qu'après l'émission de l'entête de réponse (envoyé à la connexion TCP initiale), un appel à l'adaptateur client est ajouté pour déclencher l'émission de la payload par le BE. Nous indiquerons dans les détails d'implémentation, l'endroit où cet appel doit être ajouté.

Troisièmement, pour le BE qui envoie directement la payload, l'adaptation du code (Figure 3.2 - étape 4) se fait en deux temps. Si un raccourci est décidé (cela peut dépendre de la taille du contenu renvoyé), la propriété DISC est ajoutée à l'entête de la réponse pour indiquer le raccourci aux autres serveurs. Un tampon de socket est alloué (c'est une fonction de l'adaptateur serveur). Il fournit un moyen de mettre en mémoire tampon la payload au lieu de l'envoyer par le chemin normal. Notez que ce tampon est synchronisé et peut se bloquer si la payload est trop importante (de manière similaire à la mise en mémoire tampon de TCP). L'identification de ce tampon de socket est ajoutée comme une propriété dans l'entête de la réponse, de sorte qu'elle puisse être passée en paramètre dans l'invocation du BE par le FE pour déclencher l'émission de la payload (Figure 3.2 - étape 12, 13). Une autre propriété (last_server) est également ajoutée pour transmettre l'emplacement de l'adaptateur serveur du BE (IP et port) afin qu'il puisse être invoqué par l'adaptateur client du FE pour déclencher l'émission de la payload (Figure 3.2 - étape 12).

Adaptateurs client et serveur

Les adaptateurs client et serveur sont des extensions du système d'exploitation (au niveau de l'utilisateur ou du noyau) qui fournissent différents services de communication. L'adaptateur client intercepte d'abord la communication entre le client externe et le FE. Il mémorise l'association entre le socket utilisé par le FE et toutes les informations à transmettre à l'adaptateur serveur lorsqu'il est invoqué (Figure 3.2 - étape 12). Ces informations sont principalement l'hôte et le port du client externe (afin que les paquets puissent être envoyés directement au client externe) et les clés TLS (détaillées plus loin). Deuxièmement, l'adaptateur client fournit la fonction qui permet d'invoquer l'adaptateur serveur pour demander l'émission de la payload. Cette fonction prend comme paramètres : l'hôte et le port de l'adaptateur serveur à invoquer, l'identifiant de la payload (payload_id, identifiant le tampon contenant la payload) à émettre (ces paramètres sont reçus comme des propriétés HTTP), l'hôte et le port du client externe vers lequel les paquets doivent être envoyés directement. Enfin, l'adaptateur client gère les numéros de séquence TCP afin qu'ils soient cohérents du point de vue du client externe (détaillé dans la section suivante). L'adaptateur serveur implémente la gestion des tampons de socket qui reçoivent les payloads et fournit le service permettant de recevoir l'invocation (par l'adaptateur client) pour l'émission de la payload et le service d'émission d'une payload à partir d'un tampon de socket.

Adaptation d'estampille

Comme nous ne voulons pas modifier la pile TCP intégrée dans les systèmes d'exploitation, l'application éxécutée dans le FE émettra des paquets vers le client externe avec un certain numéro de séquence TCP, sans tenir compte du fait que l'application dans le BE émettra des paquets sur la même connexion TCP avec le client externe. Par conséquent, les numéros de séquence (SN) émis par le FE et le BE doivent être coordonnés. Ceci est illustré dans la Figure 3.3.

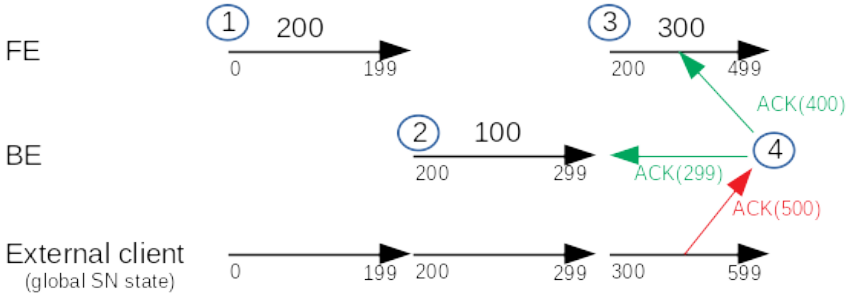


FIGURE 3.3 – *Dynamique d'estampille*

Dans cette figure, FE peut envoyer 200 segments de données (1) avec des SN pris dans l'intervalle 0 à 199, puis BE peut envoyer à son tour 100 segments de données (2) ; les SN de ce dernier doivent être compris entre 200 et 299. BE doit donc être informé des SN qu'il doit utiliser.

Puis FE peut envoyer 300 segments de données (3). Mais comme FE n'est pas au courant des transferts de données entre le BE et le client, il va envoyer ses paquets avec des SN pris dans l'intervalle de 200 à 499, ce qui est incohérent du point de vue du client externe. Le client externe devrait voir les données émise par le FE avec des SN pris dans l'intervalle 300 à 599. Il faut donc ajuster les SN émis par le FE.

La gestion des SN TCP dans DISC est implémentée dans l'adaptateur client comme suit. L'adaptateur client maintient un état SN global dans une table de hachage, en tenant compte de tous les segments de données émis d'une part par FE et d'autre part par BE. L'invocation du BE par le FE pour déclencher l'émission de la payload prend un paramètre supplémentaire qui est l'état SN global, afin que le BE puisse envoyer des segments de données avec un SN cohérent ;

l'état SN global peut être mis à jour, d'une part en connaissant la quantité de données à émettre par le BE (à partir de l'entête de réponse) et d'autre part en fonction des acquittements du client.

Dans la Fig. 3.3 à l'émission (2), l'état SN global est 200 et l'état SN global est mis à jour à 299 après qu'un transfert de données ait eu lieu entre le BE et le client. Dès lors, tout segment de donnée émis par FE est modifié par l'adaptateur client pour adapter ses SN en fonction de l'état SN global. Ceci implique que dans la Fig. 3.3 à l'émission (3) du FE, les SN de 200 à 499 vont se voir appliquer un décalage de sorte qu'ils appartiennent à l'intervalle allant de 300 à 599.

L'adaptateur client maintient dans une table de hachage les SN émis et les serveurs (FE ou BE) qui les ont émis. Tout accusé de réception de paquet (ack ou sack) reçu par FE est intercepté par l'adaptateur client et son SN est analysé pour soit envoyer un paquet ack ou sack à la pile TCP de FE (et le SN est retraduit) soit envoyer un paquet ack ou sack à BE qui l'a émis (ce qui lui permet de renvoyer un paquet, de libérer ses tampons internes et d'adapter sa fenêtre d'émission).

Dans la Fig. 3.3, un ack avec SN 500 est reçu sur la connexion TCP initiale. L'adaptateur client, en fonction de son état interne, modifie le SN pour qu'il devienne 400 (puisque 100 paquets ont été envoyés par le BE) afin que le ack soit cohérent pour FE. De plus, il génère un paquet ack pour l'émission précédente de BE avec SN 299.

Enfin, les SN des paquets reçus du client externe sont tous traités par le FE et génèrent un ack ou sack vers le client externe. Cependant, tous les paquets émis par le BE doivent inclure un ack avec le dernier SN reçu du client. Ce SN (que nous appelons le *client SN*) est transmis au BE lorsqu'il est invoqué pour l'émission de la payload, afin qu'un ack approprié puisse être ajouté dans les paquets émis.

3.3.5 Contraintes

Après avoir vu les principaux choix de conception, nous allons voir plus en détails la conception de DISC. Pour être pratique, DISC doit relever 6 défis principaux.

1. Il doit fournir un support fin pour la mise en place des raccourcis : un raccourci peut être activé sélectivement pour chaque serveur et pour chaque paire requête-réponse dans une connexion TCP ; cela permet de supporter différents types de requêtes, charges de travail et politiques d'optimisation. En outre, il doit avoir un surcoût de performance négligeable lorsqu'aucun raccourci n'est activé.

2. Il ne doit pas nécessiter d'hypothèses spécifiques concernant les fonctionnalités TCP disponibles (ou leur absence).

3. Il doit être complètement transparent pour les nœuds clients existants (externes au datacenter), et donc ne pas nécessiter une quelconque modification de leur pile logicielle.

4. Il ne doit nécessiter que des modifications très limitées de la pile logicielle sur tous les nœuds serveurs concernés dans le datacenter ; en particulier, il ne doit pas nécessiter de modification des bibliothèques standards du système d'exploitation ni du noyau du système d'exploitation.

5. Il doit être générique, c'est-à-dire qu'il doit pouvoir s'adapter à divers protocoles au niveau de l'application (par exemple, HTTP, IMAP, etc.) grâce au développement d'une simple bibliothèque d'adaptateurs pour chaque protocole.

6. Il doit prendre en charge les architectures multi-tiers avec des topologies arbitraires (par exemple, le nombre de niveaux et de saut), des types arbitraires de niveaux (front-end, logique d'application, backend de stockage, équilibreur de charge) et éventuellement des protocoles d'application hétérogènes entre les niveaux (par exemple, HTTP, JDBC).

3.4 Conception

DISC implante des raccourcis qui contournent les serveurs (sur le chemin de la réponse) pour les payloads, mais pas pour contourner complètement les serveurs intermédiaires en ce qui concerne les métadonnées incluses dans les entêtes et les pieds de page du protocole. L'objectif principal est de réduire le trafic dans les serveurs intermédiaires et globalement dans le datacenter, améliorant ainsi l'utilisation des ressources.

3.4.1 Principe

Le but de DISC est de contourner les serveurs intermédiaires lorsqu'une payload est renvoyée en réponse à une requête donnée. DISC s'appuie sur un nouveau protocole (DISC-PROT) qui ajoute des informations d'entête dans les messages de type requête ou réponse. Il permet à un nœud d'indiquer dans une requête qu'il peut être contourné pour la payload de la réponse associée à sa requête. Ensuite, le serveur qui émet la payload peut contourner la chaîne de tous les serveurs en amont qui ont permis le contournement. À l'instar de SSL, DISC-PROT se situe entre la couche TCP et le protocole de niveau application (par exemple HTTP ou IMAP). Nous fournissons DISC-STACK, une implémentation modulaire de DISC-PROT. DISC-STACK est organisé en deux parties, l'une qui dépend de TCP et est donc générique et l'autre qui dépend du protocole de niveau application et doit donc être implémentée au niveau de l'application, ce qui nécessite une légère modification de l'application.

3.4.2 Vue d'ensemble

Dans la Figure 3.4, les composants utilisés sur chaque serveur sont colorés en jaune.

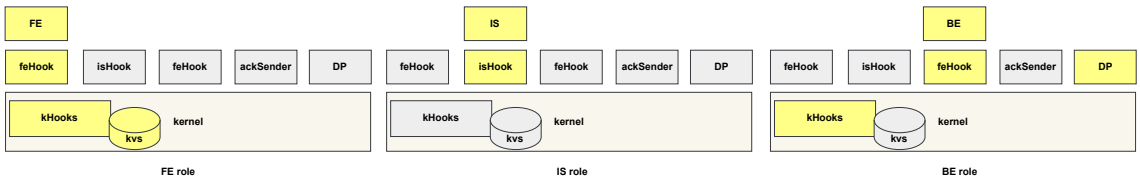


FIGURE 3.4 – *Vue d'ensemble de* DISC *sur chaque machine ou tiers*

Nous identifions quatre rôles qu'une machine ou tiers peut jouer dans l'architecture multi-tiers :

— *Backend (BE)* : un serveur qui émet une réponse qui peut inclure une payload finale.

— *Client* : une machine qui émet la requête et qui est la destination cible de la payload. Le contournement doit envoyer la payload sur la connexion (appelée *connexion initiale*) avec cette machine. Dans un premier temps, nous supposons que cette machine est externe au datacenter, mais nous relâcherons cette contrainte (implantant ainsi des raccourcis internes au datacenter) lorsque nous généraliserons le principe dans la section 3.4.3.

— *Serveur intermédiaire (IS)* : la réponse émise par le BE est finale et il peut être contourné.

— *Frontend (FE)* : il peut être contourné (comme un IS), mais le FE est directement connecté au Client avec la connexion initiale.

Par soucis de simplicité, nous supposons pour l'instant que chaque serveur joue un rôle unique. Cependant, nous montrerons dans la section 3.4.3 que les rôles des serveurs sont définis pour une paire requête-réponse donnée dans l'architecture.

La Figure 3.4 décrit les composants logiciels qui permettent l'intégration de DISC dans une telle architecture. Ces composants sont soit des points de personnalisation (hook), soit des fonctions de rappels (callback). Ils permettent donc la personnalisation d'une portion du code applicatif

ou l'ajout de nouveaux composants. Les composants de DISC sont disséminés dans tous les tiers puisqu'ils peuvent jouer n'importe quel rôle (FE, IS et BE). Nous étendons le noyau avec un nouveau composant appelé *kvs* qui est un dictionnaire clé-valeur pour enregistrer les informations sur les requêtes en cours, et un composant appelé *kHook* pour traiter les paquets TCP entrants/sortants (établissement de la connexion, accusés de réception, ...). Dans l'espace utilisateur, il y a des hook(s) pour adapter le comportement de l'application (*feHook*, *isHook* et *beHook*), et deux nouveaux composants pour gérer les accusés de réception (*ackSender*) et permettre l'envoi d'une payload directement sur la connexion initiale (*DP*).

La Figure 3.5 illustre DISC dans un diagramme de séquence de paquets pour un scénario où le BE envoie directement la payload émise sur la connexion initiale entre le FE et le client.

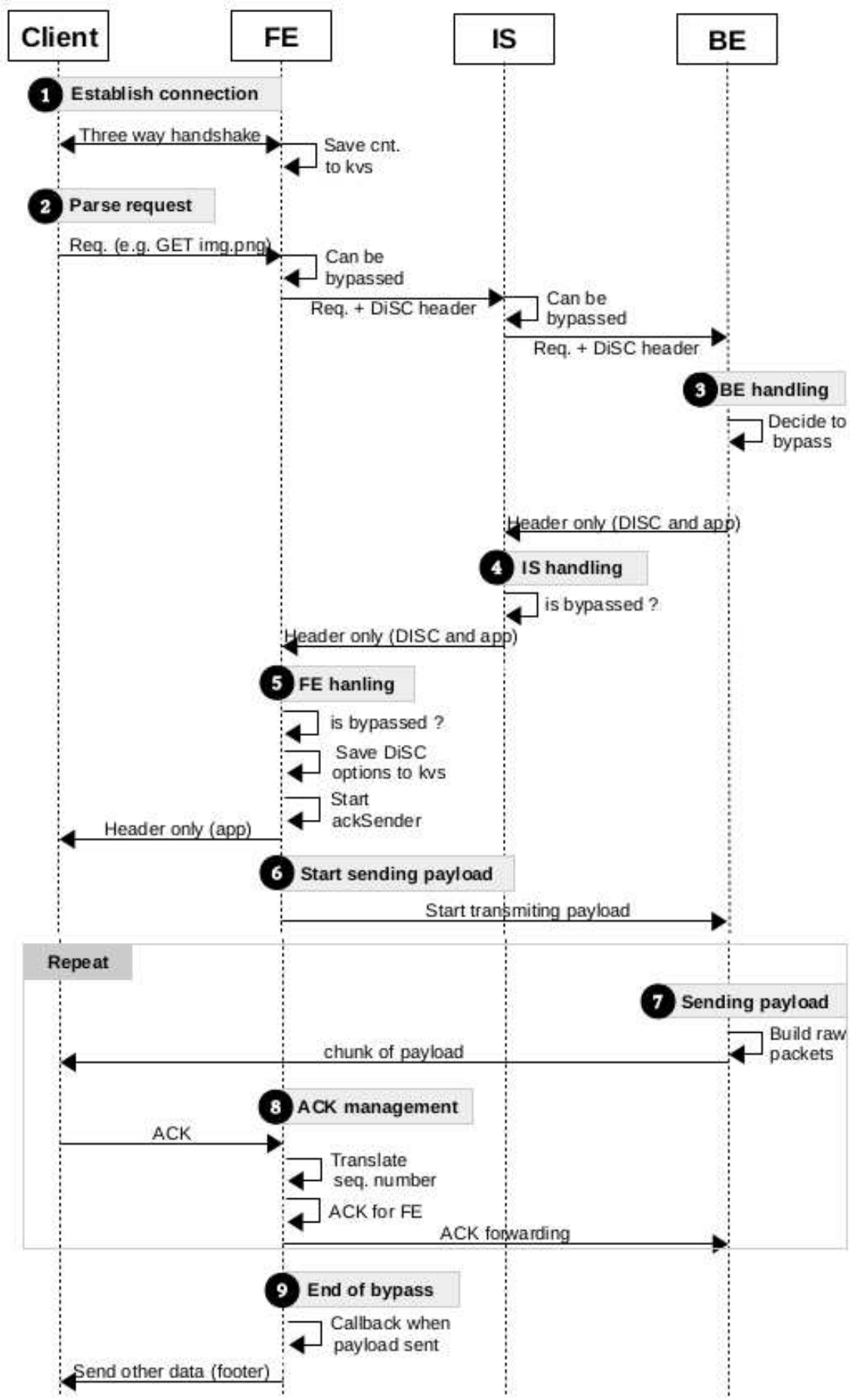


FIGURE 3.5 – *Interactions avec* DISC

`(1) Établir la connexion.` Lorsque le client se connecte avec le FE, kHook intercepte trois messages ; ces trois messages, communément appelés "handshake", concernent la séquence d'initialisation du protocole TCP. Et lorsque la connexion est établie, il crée une clé dans kvs qui identifie la session TCP. Cette clé est associée à un enregistrement `<source IP, source port, ... >` qui comprend toutes les informations nécessaires à l'envoi de paquets au client et au traitement des acquittements (acks). Notez que tous les paquets reçus sur cette connexion sont interceptés par kHook, en particulier les acks (détaillés dans la section 3.3.4).

`(2) Analyser la requête.` Lorsque le client envoie une requête, le FE peut indiquer (via feHook qui ajoute une option dans le protocole DISC-PROT) qu'il peut être contourné pour les payloads retournées (l'utilisation de cette option est détaillée dans la section 3.4.3). Ici, feHook est une adaptation au sein de l'application dans le FE, où n'importe quelle politique peut être appliquée pour accepter ou non le contournement, en fonction de la nécessité d'avoir ou non accès aux payloads retournées. Notez ici que si le FE est contourné en ce qui concerne la payload, il recevra quand même l'entête de réponse de l'application (par exemple, l'entête IMAP) et le pied de page (par exemple, le pied de page IMAP) qui peuvent être utilisés par l'application sur le FE (souvent les entêtes sont utilisés pour gérer des statistiques). Ensuite, le FE transmet la requête au IS qui se comporte de manière similaire (avec son isHook) et transmet à son tour la requête au BE.

`(3) Traitement du BE.` Sur le BE, beHook est une adaptation de l'application qui décide si la payload peut contourner les serveurs précédents. Si le raccourci est activé, la payload est envoyée à DP, au lieu de la connexion TCP avec le IS précédent. À ce stade, DP génère un identifiant (payload_id) qui désigne la payload dans DP. Ensuite, beHook construit un entête DISC-PROT qui comprend les options suivantes : une option (DISC) indiquant que la payload est sujette au raccourci, l'identifiant DP précédent (payload_id), la taille de la payload et l'adresse IP du BE (permettant au FE de demander ultérieurement au DP l'émission de la payload, contournant ainsi les autres serveurs (FE et IS)). Ensuite, le protocole de niveau application du BE construit son entête, mais avec une taille de payload égale à zéro. Les deux entêtes (DISC-PROT et protocole de niveau application) sont inclus dans le message de réponse.

`(4) Traitement de IS.` Sur IS, le message de réponse est reçu et isHook vérifie l'option de raccourci (DISC) afin d'omettre la lecture de la payload si elle est absente de la réponse. IS traite ensuite la réponse normalement et envoie la réponse (y compris l'entête DISC-PROT) à son FE précédent, sauf qu'il omet la payload si elle est absente.

`(5) Traitement de FE.` De la même manière que pour IS, feHook vérifie l'option de raccourci pour lire ou non la payload. Si le FE est contourné, alors feHook extrait toutes les informations sur le raccourci de l'entête de réponse du protocole DISC-PROT et il les stocke dans kvs dans l'entrée associée à la connexion initiale associée. Ces informations comprennent l'IP du BE, l'identifiant de la payload et sa taille, qui seront utilisés ultérieurement pour gérer les acks. Alors, le BE peut renvoyer l'entête de réponse de niveau application au client.

`(6) Envoi de la payload sur le FE.` Sur le FE, feHook fait alors un appel distant au DP sur le BE (il a son adresse IP) demandant l'émission de la payload identifiée par le payload_id. Cet appel fournit au DP toutes les informations nécessaires pour se faire passer pour le FE (y compris les numéros de séquence TCP à utiliser) et envoyer ainsi la payload sur la connexion initiale.

`(7) Envoi de la payload dans DP.` DP forge des segments de données TCP et les envoie sur la connexion initiale vers le client. A partir de ce moment, le FE recevra des acks pour ces paquets. Afin de permettre au DP de continuer à émettre la payload (d'ajuster sa fenêtre d'émission), les acks reçus sur le FE (relatifs à la payload) sont transmis au DP.

`(8) Gestion des ACK.` Tous les acks reçus sur le FE sont interceptés par kHook (le module du noyau), analysés et traités. kHook est chargé d'assurer la cohérence des numéros de séquence TCP. En effet, la pile TCP sur le FE n'est pas au courant des paquets envoyés par DP, donc kHook doit appliquer des traductions sur les numéros de séquence. De plus, les acks reçus du Client peuvent accuser réception de paquets émis par la pile TCP sur le FE, ou par DP. Les acks concer-

nant la pile du FE doivent être traduits et les acks concernant le DP sont transmis au DP. Chaque ack concernant DP est stocké dans kvs et le processus ackSender (démarré sur le FE) interroge kvs pour extraire les acks et les transmettre à DP. La gestion des numéros de séquence TCP est détaillée dans la section 3.3.4.

`(9) Fin de la dérivation.` Un protocole de niveau applicatif peut comporter un entête et un pied de page. C'est par exemple le cas du protocole IMAP. Par conséquent, le FE, qui a envoyé sur la connexion initiale son entête de réponse et a délégué l'émission de la payload au BE, peut avoir besoin d'envoyer un pied de page après la payload. Afin de fournir un moyen pour cela au feHook, nous fournissons une API pour enregistrer une fonction de rappel. Cette fonction de rappel est appelée lorsque kHook observe à partir des acks reçus que le client a reçu toute la payload. feHook peut alors préparer le pied de page et enregistrer une fonction de rappel qui enverra le pied de page sur la connexion initiale lorsque la payload a été entièrement envoyée. Notez ici que nous n'avons pas voulu répondre à ce besoin de pied de page par une invocation synchrone du DP (pour qu'il envoie le pied de page lorsque la payload a été envoyée), car cela aurait affecté le modèle de threading du FE. Avec ce mécanisme de rappel, la gestion des pieds de page reste asynchrone (sans bloquer un thread). Avec DiSC, nous observons que la capacité d'émettre sur la connexion initiale est temporairement déléguée au BE et revient au FE pour l'envoi du pied de page ou l'envoi de l'entête de réponse de la requête suivante.

3.4.3 Généralisation

La section précédente a donné le principe général de DiSC avec un scénario où une payload envoyée par le BE contourne le IS et le FE et est envoyée directement au Client. Cependant, les rôles (FE, IS et BE) définis par DiSC peuvent être joués par n'importe quel serveur. Comme mentionné dans la section précédente, chaque niveau peut décider s'il accepte d'être contourné, déterminant ainsi ses rôles. Pour généraliser, nous considérons un exemple avec une architecture à 4 niveaux (First-Inter1-Inter2-Last) illustré sur la Figure 3.6 où First et Inter2 peuvent être contournés, mais pas Inter1.

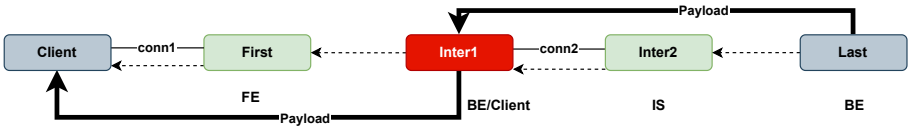


FIGURE 3.6 – *Généralisation des raccourcis*

Le protocole DiSC-PROT inclut dans son entête deux options pour gérer le contournement des serveurs :

— NB_TIER est un compteur initialisé à 0 sur le serveur First, incrémenté chaque fois qu'une requête est transmise au serveur suivant et décrémenté chaque fois qu'une réponse est renvoyée au serveur précédent. Il permet à un serveur de connaitre sa place dans le chemin (dans BYPASSED_TIER).

— BYPASSED_TIERS est une bitmap, chaque bit indiquant si un niveau permet ou non le contournement.

Plus précisément, les significations des rôles qui peuvent être joués par les serveurs sont :

— Client : une machine qui reçoit une payload (peut maintenant être interne au datacenter).

— BE : un serveur qui envoie la payload (avec ou sans raccourci). En cas de raccourci, la payload est envoyée au DP local.

— IS : un serveur qui reçoit une réponse sans payload et envoie une réponse sans payload.

— FE : un serveur qui reçoit une réponse sans payload et invoque le DP pour demander l'émission de la payload sur la connexion qui précède (dans le chemin) le serveur.

Ensuite, voici le comportement sur chaque type de serveur (Last, Inter et First).

Last. Le serveur Last joue le rôle de BE. Le raccourci est possible si ce serveur (Last) le souhaite pour la payload actuelle (selon sa politique) et si le serveur précédent accepte d'être contourné. Si le raccourci est possible, il envoie une réponse sans payload et la payload est envoyée au DP local. Sinon, la payload envoyée dans la réponse.

Inter. En ce qui concerne un serveur Inter, s'il reçoit une réponse qui comprend une payload, il joue les rôles de Client (pour son successeur dans le chemin) et BE (pour son prédécesseur dans le chemin). S'il reçoit une réponse sans payload, cela signifie qu'il accepte d'être contourné, car il a été contourné par le serveur qui a envoyé cette réponse. Alors, si le serveur précédent (dans le chemin) accepte d'être contourné, alors le serveur actuel joue le rôle IS (simple relais). Si le serveur précédent n'accepte pas d'être contourné, alors le serveur courant joue le rôle FE (et il demande au DP qui détient la payload de l'envoyer sur la connexion précédente). Notez que dans ce dernier cas, le serveur précédent recevra une réponse avec la payload (il joue donc le rôle Client) et il jouera les rôles BE s'il est Inter ou enverra simplement la payload au Client externe s'il est First.

First. En ce qui concerne le serveur First, s'il reçoit une réponse avec une payload (il joue un rôle de Client), il l'envoie simplement sur la connexion initiale. S'il reçoit une réponse sans la payload, il joue le rôle de FE.

Dans le scénario de la Figure 3.6, Last joue le rôle BE. Il décide d'activer le contournement et comme Inter2 accepte d'être contourné (d'après la bitmap), Last envoie une réponse sans la payload et la payload est envoyée à DP. Inter2 joue le rôle de FE puisque Inter1 n'accepte pas d'être contourné. Par conséquent, Inter2 envoie une réponse sans payload et demande au DP de Last d'envoyer la payload sur la connexion Inter2-Inter1. Inter1 reçoit une réponse avec une payload, il joue donc les rôles de Client et BE et puisque First accepte d'être contourné, il envoie une réponse sans la payload et la payload est envoyée à son DP. First reçoit une réponse sans payload et comme son prédécesseur (Client) ne peut pas être contourné, il joue le rôle de FE en envoyant une réponse sans la payload, mais en demandant au DP de Inter1 d'envoyer la payload sur la connexion First-Client (connexion initiale).

3.5 Implémentation

DISC est implémenté avec un module noyau (kHook, kvs), deux modules de niveau utilisateur (ackSender et DP) et des adaptations d'applications (feHook, isHook, beHook). kHook s'appuie sur eBPF (Extended Berkeley Packet Filter) pour intercepter et adapter les paquets et kvs est implémenté avec les tableaux associatifs eBPF (un système de stockage générique pour le partage de données entre le noyau et l'espace utilisateur). En ce qui concerne les adaptateurs d'application, ils peuvent être mis en œuvre par les développeurs avec des correctifs ou des extensions d'application lorsqu'un tel support est fourni (par exemple, Nginx permet la définition de filtres, handler et callback). Cette section détaille l'implémentation de chaque composant de DISC, illustrée dans la Figure 3.7. À la fin de la section est présenté le principe d'intégration dans serveurs applicatifs, que nous avons utilisé pour les deux applications (web et email) présentées dans la section 3.2.

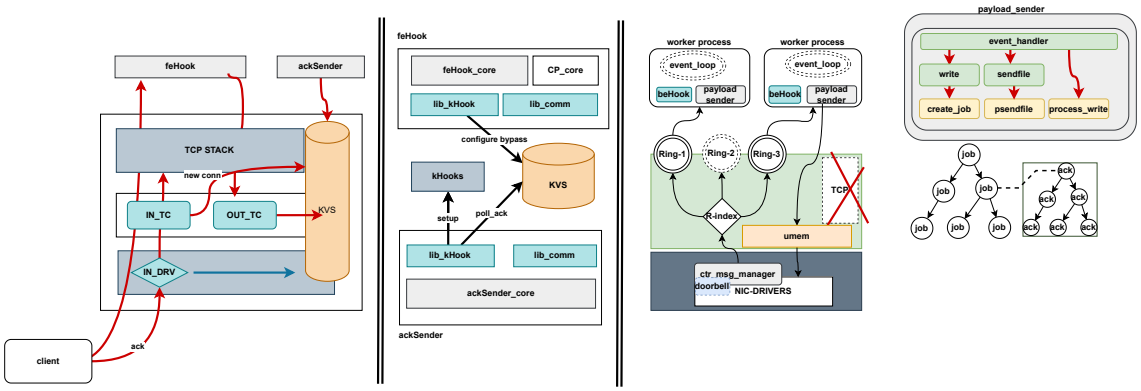


FIGURE 3.7 – *Composants de* DISC

3.5.1 librairie d'intégration avec DISC

Libs. Afin de fournir un support pour le développement des adaptateurs au niveau des applications, nous fournissons deux bibliothèques en espace utilisateur : lib_kHook et lib_comm.

librairies	fonctions	descriptions
lib_kHook	setup_khook()	instancie le kHook dans le noyau
	poll_ack()	extrait les ACKs du KVS à transmettre au DP
	register_range()	enregistre une plage de données envoyée par le DP dans la table de hachage KVS
lib_comm	foward_ack()	transmet au DP un ACK extrait du KVS
	start_transmit()	envoie un message au DP pour démarrer la transmission

Lib_kHook. Implémente une API pour l'instanciation de kHook (setup_khook()) dans le noyau et pour l'accès aux kvs : interrogation des acks des kvs (poll_ack()) à envoyer à DP par ackSender, et enregistrement d'une plage de paquets envoyée par BE (register_range()). register_range() est appelé par feHook après la réception de l'entête de réponse qui inclut dans son entête DISC-PROT l'identifiant de la payload, sa taille et l'IP du BE (localisation du DP). Ces paramètres sont passés à register_range() qui enregistre dans kvs la plage de paquets associée à la payload et les informations pour traiter les acks de ces paquets (IP du BE et identifiant de la payload). register_range() reçoit également comme paramètre le SN associé à la plage de paquets enregistrée (déduit de l'état global SN et de la taille de la payload) ainsi que l'adresse IP et le port du client. Notez que register_range() reçoit également une fonction de rappel appelée (par kHook) lorsque la payload complète a été reçue par le client (permettant ainsi de gérer les pieds de page du protocole comme dans IMAP).

Lib_comm. Implémente une API pour la communication entre les composants DISC situés sur des machines différentes. Elle inclut des fonctions pour l'invocation à distance d'un DP (start_transmit()) et pour la transmission des acks au DP (foward_ack()). start_transmit() reçoit

comme paramètres l'IP du BE et l'identifiant de la payload. Elle reçoit également l'adresse du client (IP et port) et l'état global SN à utiliser pour l'envoi des paquets de la payload.

3.5.2 Adaptation du serveur frontal (FE)

kHook. Est activé sur chaque serveur jouant le rôle de FE. kHook est composé de trois intercepteurs appelés in_tc, out_tc et in_drv. Les deux in_tc et out_tc sont implémentés dans la couche TC (Traffic Control) tandis que in_drv qui s'appuie sur XDP (eXpress Data Path) et est intégré dans le pilote de la carte réseau. kHook aurait pu être entièrement implémenté au niveau TC, mais nous avons introduit in_drv pour des raisons de performance pour éviter les remontées inutiles de paquets dans la pile TCP du FE.

in_tc. Fait principalement deux choses. Premièrement, il intercepte les connexions TCP entrantes et initialise kvs avec les caractéristiques de la connexion établie. Plus précisément lorsque in_tc reçoit un paquet SYN, il extrait de ce paquet la fenêtre initiale du client, le facteur de grandissement de la fenêtre (WS), la taille de segment TCP supportée par le client (MTU), l'indication si le client supporte l'acquittement sélectif (SACK-PERM). Il associe à ces informations l'identifiant de la connexion entrante (ip source, port source, ip destination, port destination) puis crée une entrée dans kvs. Deuxièmement, il intercepte le dernier ack de fermeture d'une connexion, car c'est à ce moment où il peut supprimer la connexion dans le kvs. Tous les autres paquets entrants sont traités par in_drv.

in_drv. Intercepte tous les paquets reçus du client après la connexion. Un tel paquet peut contenir des données et doit inclure un ack. Si le paquet comprend des données, alors les données sont transmises à la pile du FE. Mais de toute façon (que le paquet contienne des données ou non), les acks doivent être traités, de sorte qu'ils soient envoyés de manière cohérente (en ce qui concerne les numéros de séquence (SN)) à la pile du FE ou à DP (qui a émis la payload). S'il n'y a pas encore eu de raccourci, les paquets sont transmis tels quels à la pile du FE. Dès qu'un raccourci se produit, les SN inclus dans les acks doivent être traduits comme décrit dans la section 3.3.4. Une table des SN qui ont été émis (retournés au client) par la pile FE et le DP est enregistrée dans le kvs. Cette table inclut les plages de SN émis et les adresses IP des serveurs émetteurs, et sont conservés dans le kvs tant qu'ils ne sont pas acquittés par le client. Un SN dans un ack reçu qui acquitte des données d'une plage génère un ack qui est envoyé au serveur émetteur de la plage (FE ou DP) avec le SN acquitté le plus élevé de la plage (notez que plusieurs plages peuvent être acquittées par un ack). Si un ack généré est transmis à FE, le SN est retraduit en SN tel qu'envoyé par FE. Si un ack généré doit être envoyé au DP, il est stocké dans kvs et consommé par le processus ackSender.

out_tc. En ce qui concerne out_tc, il traduit le SN envoyé par FE au client, en tenant compte de l'état global du SN (voir Section 3.3.4) qui est retourné lorsque DP est invoqué pour l'envoi d'une payload. En cas de fermeture de connexion active par le serveur, il intercepte le dernier ACK sortant pour supprimer la connexion enregistrée par in_tc dans le kvs.

AckSender. C'est un processus lancé sur chaque FE. Il s'appuie sur poll_kvs() de lib_kHook et forward_ack() de lib_comm afin d'extraire les acks de kvs et de les envoyer au DP. Ces acks comprennent les acks ACK, SACK, DUP ACK et ZERO WINDOW. DP peut alors gérer la mise en mémoire tampon de la payload (libération des tampons), la réémission des paquets perdus et la fenêtre d'émission de DP.

FeHook. Il appelle register_range() après la réception de l'entête de réponse pour stocker dans kvs les informations concernant la payload émise. Il appelle ensuite start_transmit() pour demander à DP l'émission de la payload.

3.5.3 Adaptation du serveur intermédiaire (IS)

Lorsqu'un serveur joue le rôle de IS, son implémentation s'appuie sur le module **isHook**. Ce module permet au serveur intermédiaire de prendre en compte le protocole DISC-PROT et DISC-

STACK. D'une manière générale, il intercepte toutes les réponses pour savoir s'il faut exploiter la payload comme expliquée dans la section 3.4.3.

3.5.4 Adaptation du serveur de traitement (BE)

L'adaptation d'un serveur jouant le rôle de BE se fait au moyen de deux modules : beHook et DP (Figure 3.8).

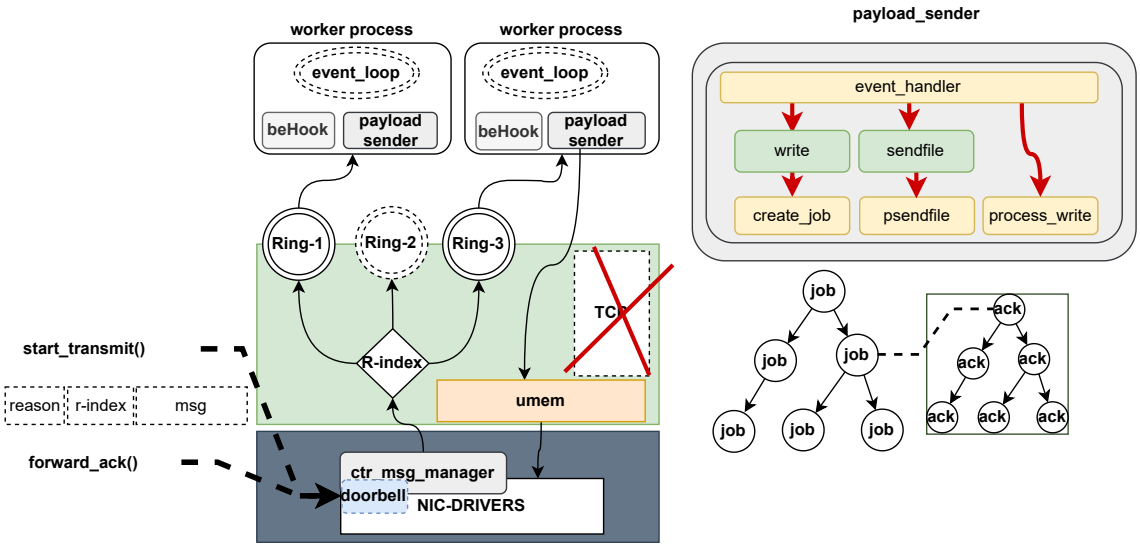


FIGURE 3.8 – *Composants de* DISC *sur le BE (DP et beHook)*

beHook. beHook est le module applicatif appelé chaque fois qu'au niveau applicatif, l'application parse une requête. beHook détermine s'il faut appliquer un raccourci. En cas de raccourci, beHook ajoute les informations de raccourci dans l'entête de la réponse associé à la requête. Ces informations sont : DISC, last_server, payload_id et payload_size. DISC est un flag indiquant qu'il y a un raccourci, last_server contient l'adresse IP du serveur jouant le rôle du BE et un identifiant de file de message (r-index) associée au processus sur le serveur, payload_id et payload_size sont respectivement l'identifiant et la taille de la payload.

DP. L'implémentation de DP repose sur deux modules appelés ctr_msg_manager et payload_sender.

ctr_msg_manager. Aiguille les requêtes émises par FE lorsqu'un ack est transmis (foward_ack()) ou l'émission d'une payload est demandée (start_transmit()). Chaque message reçu par ctr_msg_manager est déposé dans une file de message (Ring-n) en fonction de r-index. Notons ici que ctr_msg_manager court-circuite complètement la pile TCP pour les requêtes DISC venant du FE.

En effet, **ctr_msg_manager** est directement intégré dans le driver de la carte réseau grâce à XDP qui est extension eBPF permettant d'injecter du code dans le noyau sans recompilation. Il intercepte tous les messages en provenance du FE sur un port particulier destiné à cet effet.

Le mécanisme employé ici est le kernel bypass comme vu dans l'état de l'art. Ring-n est un espace mémoire noyau partagé avec l'espace utilisateur du BE. Fait ainsi, tout message du feHook ou du ackSender, lorsqu'il est reçu par la carte réseau du BE est directement disponible pour le DP. Sur cet espace mémoire, nous associons un descripteur de fichier et de cette façon, tout processus désirant consommer les données ou être notifié, peut le faire via un mécanisme de polling (epoll, select, ...).

Nous rappelons qu'il est d'usage que la plupart des serveurs applicatifs implémentent des gestionnaires de boucle d'évènements basés sur epoll, poll, kqueue, select, libuv et libevent. Dans ce modèle, un descripteur de fichier ou de socket permet de générer les évènements et chaque évènement est traité par une fonction de rappel (callback/handler). Il suffit donc de fournir un

descripteur de fichier et un callback au serveur applicatif pour qu'il soit apte à ingérer et traiter les données d'une source.

C'est grâce à ce mécanisme que nous injectons les messages du FE au BE, pour lui permettre de lire les messages de la liste circulaire, les décoder et réaliser le traitement adéquat. Cette approche nous évite donc l'ajout d'un processus supplémentaire dans le BE.

payload_sender est le handler ou callback d'évènement qui s'occupe de l'émission des payloads (c'est-à-dire des paquets sur la connexion qui précède le FE) lorsque beHook (l'adaptation applicative de BE) a décidé que l'envoi de la payload se fait avec un raccourci. Il peut être déclenché par la boucle d'évènement (event_loop) dans les cas suivants :

— lorsque le BE soumet une payload au DP (sendfile/write/writev),

— lorsque event_loop reçoit un message (provenant du FE) dans Ring-n.

Lorsque le BE soumet une payload, payload_sender fait appel à la primitive create_job. Cette primitive, crée un job et l'ajoute dans une structure de donnée arborescente (arbre rouge noir), l'arbre rouge noir ayant pour avantage que les opérations d'insertion, de recherche et de suppression ont une complexité logarithmique ; le job créé comprend un identifiant unique (payload_id) de la payload. La primitive retourne cet identifiant et le descripteur de fichier associé à Ring-n. payload_sender ajoute le descripteur dans la boucle d'évènement event_loop et lui associe la fonction de rappel process_write. Enfin, payload_id est rajouté dans l'entête de réponse.

Dans notre prototype, le contenu de la payload dans un job peut être de deux types :

— il peut s'agir d'un flux de données mis en mémoire tampon (tx descriptor) comme on peut le voir sur la Figure 3.9 et envoyé par l'application (BE) sur le descripteur de fichier. BE commence à remplir le tampon et bloque jusqu'à ce que les tampons soient libérés par les acquittements reçus.

— il peut s'agir d'un fichier, ce qui signifie que les paquets émis sont directement lus depuis un fichier. Ceci est équivalent à la primitive Linux sendfile() qui copie directement les données d'un descripteur de fichier vers un descripteur de socket. Notons que ceci nous permet d'éviter une double copie du fichier. En effet, pour lire le fichier, il suffit de passer un tx descriptor aux primitives systèmes readv et preadv et ainsi, le contenu du fichier lu est directement logé à l'emplacement correspondant (payload) du tampon mémoire associé à la socket (raw_socket_fd).

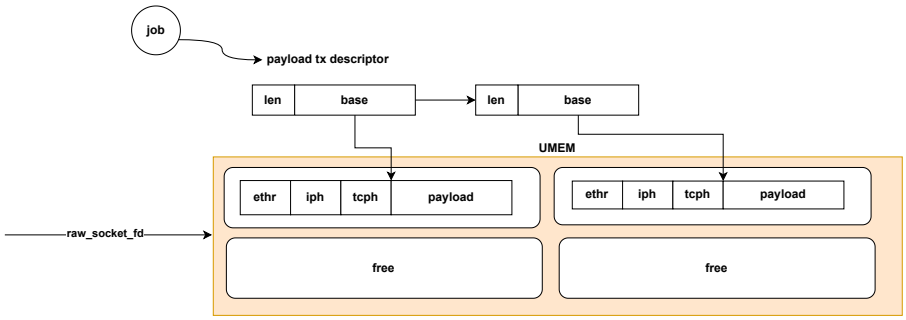


FIGURE 3.9 – *Géométrie des trames dans umem ring*

Dans la Figure 3.9, raw_socket_fd est le tampon mémoire associé à notre socket. Nous nous appuyons sur les sockets de la famille af_xdp, une solution native Linux du kernel bypass permettant de transmettre et recevoir les paquets réseaux bruts par lot. Ce tampon mémoire est unique pour chaque processus et est initialisé à chaque démarrage de processus.

À la réception d'une notification, lorsque feHook (sur FE) appelle start_transmit() pour déclencher l'émission de la payload, ctr_msg_manager reçoit le message et récupère l'index de Ring-n associé, puis pose le message. Par la suite event_loop est notifié et invoque process_write via

event_handler qui lui passe l'évènement reçu. Cette action marque le début de la transmission. Le message reçu contient les informations nécessaires pour construire les trames TCP en encapsulant la payload avec les entêtes (ethr, iph, tcph). Toutes les données qui avaient été mises en mémoire tampon peuvent être transmises directement par le driver de la carte réseau.

Transmission. Le processus de transmission se déroule de la façon suivante : payload_sender dans un premier temps récupère l'id du job dans le message puis fait une recherche dans l'arbre pour retrouver la structure de donnée job. Ensuite il récupère dans le job le descripteur du fichier à transmettre ou le descripteur (tx descriptor) de la payload.

Les informations de session TCP (ip source, port source, ip destination port destination, seq, ack, window) sont contenues dans le message reçu. Comme c'est le début de la transmission, payload_sender va stoquer ces informations dans job pour une utilisation ultérieure, par exemple dans le cas d'une retransmission ou de la transmission du segment suivant.

Une fois les informations collectées, payload_sender va sélectionner un ensemble de descripteurs (tx descriptor) à envoyer. Ces descripteurs ont été alimentés par le BE ou sont alimentés depuis un fichier dans le cas d'un sendfile. La taille du segment de données à transmettre est déterminée à partir la fenêtre d'émission initiale contenue dans le message. Comme expliqué précédemment, le contenu est directement logé dans les blocs du tampon mémoire de la Figure 3.9. Ensuite payload_sender va construire in situ dans le tampon mémoire les entêtes (ethr, iph, tcph) à l'aide des informations de session TCP et calculer les contrôles de somme (checksum) des entêtes iph et tcph. Enfin, en une transaction, le noyau est notifié qu'il y a des paquets du tampon mémoire à transmettre. Cette notification se fait au moyen de sendto et le noyau sait qu'il doit transmettre les données. Tous les emplacements du tampon restent occupés tant qu'il y a pas eu acquittements. Ceci nous évite de reconstruire les trames dans le cas d'une retransmission. Les emplacements seront libérés au prochain message d'acquittement reçu.

Acquittement. Ici, payload_sender avance la fenêtre d'émission, libère les emplacements occupés conformément aux acks reçus, et détermine les nouveaux tampons à envoyer. Notons ici qu'il peut s'agir d'un acquittement sélectif (SACK) ou d'un acquittement signalant la perte de données (DUP ACK). Dans tous les cas, le principe s'apparente au GO-BACK-N ARQ [Tow79]

Retransmission. La retransmission est implémentée avec des timers. DISC fournit un timer par descripteur (raw_socket_fd) avec une granularité de 200ms et une fonction de rappel retransmission_callback. DISC expose ce timer au BE grâce à timer_fd qui est un mécanisme avancé de Linux pour générer les notifications d'horlorge. A chaque tic d'horloge, retransmission_callback est appelé pour balayer la structure job.

Pour chaque job, retransmission_callback détermine s'il y a lieu de retransmettre un segment. En effet chaque fois qu'un segment est transmis, on enregistre un compteur. Ce compteur est décrémenté à chaque tic (200ms se sont écoulés).

Si le compteur tombe à zéro et que le segment n'est pas acquitté, payload sender considère qu'une partie du segment est perdue et démarre la retransmission à partir du dernier ack reçu pour le segment. Fait ainsi, toutes les données comprises entre ack et taille segment sont retransmises.

Fin de la transmission. Ici, payload_sender fait le ménage, il libère la structure job et l'enlève de l'arbre.

3.5.5 Modèle d'intégration dans un serveur applicatif

Nous décrivons ici l'intégration de DISC dans les serveurs applicatifs en prenant l'exemple d'un serveur HTTP. Les adaptations décrites ici correspondent aux feHook, isHook et beHook décrit précédemment.

Le (Programme 3.1) est un prototype du code du IS (Web Server) et son isHook (ligne 11 à 17). Nous supposons que le serveur a deux descripteurs de fichiers socket fdc (côté client) et fds (côté serveur). Ce code est exécuté lorsqu'une requête est reçue sur fdc et doit être transmise à fds.

```
request_header = receive_HTTP_request_header(fdc, ...);
// request header may be processed and adapted
send_HTTP_request_header(fds, request_header, ...);
content = receive_HTTP_content(fdc, ...);
// content may be processed and adapted
send_HTTP_content(fds, content, ...);
...
response_header = receive_HTTP_response_header(fds, ...);
// response header may be processed and adapted
send_HTTP_response_header(fdc, response_header, ...);
if (! response_header->DISC) {
  content = receive_HTTP_content(fds, ...);
  // content may be processed and adapted
  send_HTTP_content(fdc, content, ...);
} else {
  // XXX
}
```

Listing 3.1 – Programme 1

On voit que la payload (content) est omise si la propriété DISC est vraie.

Ensuite, pour le FE, le code de son feHook (Figure 3.2 - étape 9) est très similaire à ce qui précède, sauf qu'après l'émission de l'entête de réponse (envoyé à la connexion TCP initiale), un appel à l'adaptateur client est ajouté pour déclencher l'émission de la payload par le BE. L'endroit où cet appel doit être ajouté dans le modèle de code ci-dessus est identifié par le marqueur XXX.

Troisièmement, pour le BE qui envoie directement la payload sur la connexion initiale, le code de son beHook (Figure 3.2 - étape 4) est illustré ci-dessous (Programme 3.2). On voit que s'il y a un raccourci, l'entête de la réponse est envoyé sur le descripteur normal (fd_header) mais la payload est envoyée sur le descripteur du DP.

```
request_header = receive_HTTP_request_header(fdc, ...);
content = receive_HTTP_content(fdc, ...);
// process the request and compute response content
response_header = prepare_HTTP_response_header(..);
// decide whether the payload is shortcut
DISC = ...
fd_header = fdc;
fd_content = fdc;
if (DISC) {
  addProperty(response_header, "DISC", "true");
  fd_content = allocate_socket_buffer(...);
  addProperty(response_header, "payload_id", fd_content);
  addProperty(response_header, "payload_size", ...);
  addProperty(response_header, "last_server", ADAPTATOR_LOCATION);
}
send_HTTP_response_header(fd_header, response_header, ...);
send_HTTP_content(fd_content, content, ...);
```

Listing 3.2 – Programme 2

3.5.6 Intégration avec TLS

L'intégration de TLS dans notre prototype s'appuie sur wolfssl [wE18]. La conception du support TLS s'inspire du travail décrit dans [HHSE21].

Elle s'appuie sur deux fonctions qui se comportent comme des primitives de sérialisation :

— wolfSSL_tls_export qui transforme un objet SSL_SESSION en une représentation ASN1

— wolfSSL_tls_import qui transforme une représentation ASN1 d'une session SSL/TLS en un objet SSL_SESSION

Et d'un cache au niveau du FE qui nous sert d'espace de stockage pour les sessions SSL/TLS sérialisées. Le cache est une structure de données clé-valeur qui s'appuie sur les MAPs eBPF. La clé ici est l'identifiant de la session TCP en cours et la valeur est constituée d'un numéro de version et la session SSL. Dans notre prototype DISC, FE et DP s'exécutent sur des machines différentes. FE et DP partagent le même certificat et les algorithmes de chiffrement sont identiques. Cette précaution nous évite d'envoyer systématiquement les certificats. Fait ainsi, la pile TLS est initialisée uniquement au démarrage du BE. Lorsque feHook invoque start_transmit() pour demander l'émission d'une payload, il invoque wolfSSL_tls_export et passe en paramètre la représentation ASN1 de la session SSL. DP reçoit cette représentation et invoque wolfSSL_tls_import pour recréer la session SSL et l'associer au descripteur de fichier utilisé par payload_sender pour émettre des morceaux de payload. En fin de transmission de la payload, DP renvoie la représentation ANS1 de la session SSL, kHook intercepte ce message et met à jour le cache SSL. feHook peut consulter le cache pour restaurer la nouvelle session SSL. Remarquez ici que :

— les morceaux chiffrés de la payload sont mis en mémoire tampon dans le DP afin qu'ils puissent être envoyés à nouveau si un SACK ou un DUP ACK est reçu.

— le FE ne peut pas émettre de données vers le client tant que la session SSL se trouve sur le DP (jusqu'à ce que la session SSL soit ramenée sur le FE)

— des alertes peuvent être émises par le client pendant l'émission de la payload par le BE.

Le principal impact de ce support de TLS est que nous déplaçons la charge de travail de TLS du FE vers le BE. Du point de vue du datacenter, il ne s'agit pas d'une surcharge, mais d'un déplacement, et cela peut d'ailleurs être considéré comme un avantage, puisque le FE (qui est souvent un équilibreur de charge) peut être un goulot d'étranglement alors que le BE est souvent répliqué pour des raisons de scalabilité.

Le prototype développé ne prend pas en compte la gestion des alertes [tls06]. Dans le protocole TLS, une alerte peut être de type *warning* ou *fatal*. Une alerte peut être générée par le client pour signaler une anomalie. Si une telle alerte survient, l'emission de la payload doit être mise en veille pour permettre au FE de résoudre l'anomalie. Ensuite, le FE doit disposer d'une version de la session TLS la plus à jour dans son cache. Si ce n'est pas le cas, le FE doit récupérer auprès du BE la dernière version de la session TLS. Deux cas de figure peuvent se présenter :

— Une alerte de type fatal mettra fin à la transmission.

— Une alerte de type warning, une fois résolue, provoquera un nouvel export de la session TLS vers le BE et débloquera la transmission de la payload au sein du BE.

3.6 Évaluation

Cette section présente les résultats de l'évaluation de DISC. Notre évaluation porte sur les questions de recherche suivantes :

— **RQ1 - Avantages**. Quels sont les avantages de DISC par rapport à une architecture classique ?
— **RQ2 - Surcharge**. Quelle est la surcharge potentielle de DISC lorsqu'il n'y a pas de raccourci ?

Pour répondre à ces questions, nous utilisons les métriques suivantes : consommation du processeur, débit et latence des requêtes.

3.6.1 Environnement de tests

Tous les tests sont réalisés sur la plateforme Cloudlab [DRM+19] qui est similaire à Grid5000 [CCD+05]. L'environnement de test est constitué des machines ci-après (Tableau 3.1). Toutes les machines ont les mêmes configurations et sont connectées entre elles suivant une topologie Clos [Clo53] comme le montre la Figure 3.10

C220G5	Intel Skylake, 20 core, 2 disks
CPU	Two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz
RAM	192GB ECC DDR4-2666 Memory
DISK	One 1 TB 7200 RPM 6G SAS HDs
DISK	One Intel DC S3500 480 GB 6G SATA SSD
NIC	Dual-port Intel X520-DA2 10Gb NIC (PCIe v3.0, 8 lanes)
NIC	Onboard Intel i350 1Gb

TABLEAU 3.1 – *Configurations machines*

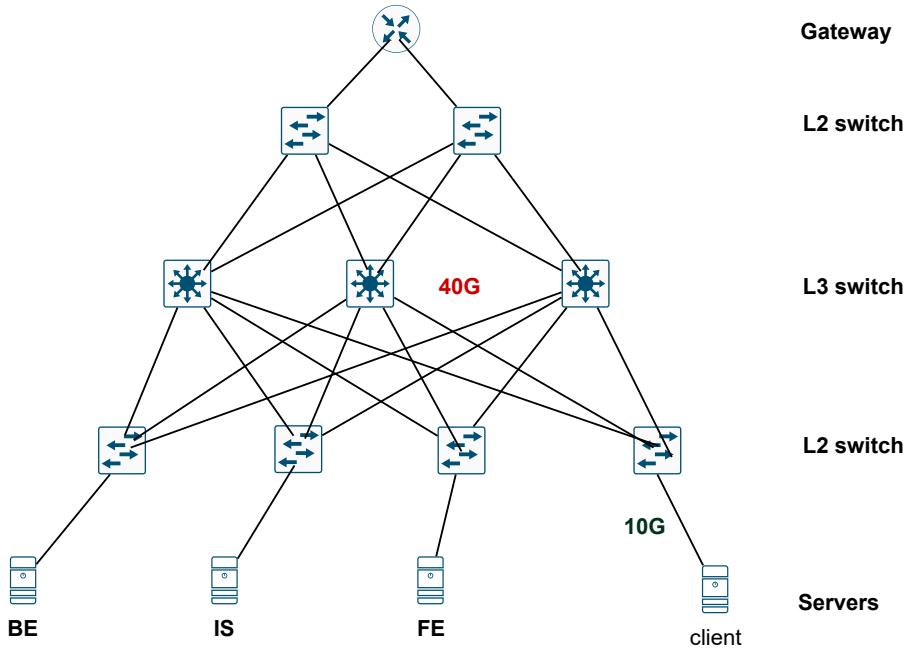


FIGURE 3.10 – *Architecture du banc de test*

Machines et Systèmes Toutes les machines ont les mêmes caractéristiques (voir Tableau 3.1) et ont la même version d'une distribution Linux, pour notre cas Ubuntu 20.04 et un noyau 5.12. La machine client sert d'injecteur de charge à la fois pour le microbenchmark et le macrobenchmark. La machine FE sert de frontal ; elle joue le rôle de reverse proxy quand on a une architecture client-FE-IS-BE et le rôle de loadbalancer lorsqu'on a une architecture client-FE-BE. La machine IS joue le rôle de loadbalancer lorsqu'on a en frontal un reverse proxy c'est-à-dire suivant l'architecture client-FE-IS-BE. La machine BE joue le rôle d'émetteur de payload dans tous les cas. Nous utilisons first pour désigner une instance de FE, middle1 et middle2 comme instances de IS et last1 pour une instance de BE.

Conditions sur les machines Sur la machine client, bien qu'aucune configuration n'est appliquée pour être compatible avec DISC, nous avons tout de même augmenté ses capacités (il s'agit du nombre threads, du nombre de descripteurs de fichier ouverts pour un processus) dans le but de simuler un grand nombre de clients. Sur les autres machines, aucune compilation du noyau n'est faite, seuls sont installés : les composants DISC, les outils pour récolter les métriques et les différents serveurs applicatifs en fonction du rôle joué par la machine. Les tests se déroulent chaque fois dans deux configurations, vanilla et DISC. Dans la configuration vanilla, les serveurs applicatifs sont déployés en l'état et DISC n'est pas installé. Dans la configuration DISC, les serveurs applicatifs sont déployés avec tous les composants de DISC. Dans les deux configurations, nous désactivons l'offload du checksum de la carte pour les transmissions, car DISC n'a pas le support de l'offload. Pour ces évaluations, nous désactivons la primitive sendfile pour le serveur applicatif du BE afin que le transfert de fichier ne puisse pas être délégué au noyau.

Benchmarks Nous utilisons à la fois un microbenchmark et un macrobenchmark.

Le microbenchmark est constitué d'un injecteur de charge synthétique wrk [wrk] qui permet d'émuler plusieurs sessions concurrentes de clients HTTP. L'application à tester ici est un serveur web avec en frontal un reverse proxy et un loadbalancer L7. Nous utilisons nginx, un serveur applicatif opensource assez versatile qui peut faire office de reverse proxy, loadbalancer et serveur web. Notre serveur web sert du contenu statique. Pour notre scénario, nous avons des images avec des tailles variables de 16k à 64k, ces images sont stockées sur le disque SSD du serveur web last1 avec un système de fichier ext4. Nous prenons ces tailles car elles sont assez représentatives. D'après le site httparchive.org sur les 5 dernières années, la somme totale de toutes les images requises par une page à une valeur médiane de 1039k pour un total de 25 requêtes, soit approximativement 40k par image. Ces statistiques sont obtenues grâce aux données récoltées sur les sites de e-commerce faits avec drupal, magento et wordpress.

Pour le macrobenchmark, nous utilisons specweb [Spe09].

3.6.2 Overhead

L'objectif de cette expérimentation est de mesurer l'impact des hooks. Nous utilisons une architecture client-FE-BE. Les hooks de DISC interceptent toutes les requêtes envoyées par le client au FE ou par le FE au BE, même lorsqu'il n'y a pas de raccourci.

Pour cela, nous comparons deux configurations. La première configuration est une architecture sans DISC. La deuxième configuration est une architecture avec DISC, mais où aucun raccourci n'est activé. Nous utilisons le microbenchmark wrk [wrk] qui est un injecteur de charge avec lequel nous définissons deux profils de charges. Dans le premier profil (scénario A) de charge, le client effectue des requêtes de page de 1k avec un débit constant pendant 15s (un débit de 1000 requêtes par seconde). Dans le deuxième profil (scénario B) de charge, le client fait une requête d'une page de 16k puis effectue les mêmes accès que dans le scénario A. Dans les deux scénarios, le BE est configuré pour faire un raccourci pour les pages dont la taille est supérieure à 16k. Ainsi, dans les deux scénarios, il n'y a pas de raccourci (sauf pour le premier accès du scénario B). Le scénario A correspond à une exécution avec DISC sans translation de SN et le scénario B avec translation de SN.

Nous collectons les métriques suivantes : le temps passé dans chaque hook, la consommation CPU, la latence et le débit.

Impact sur le CPU - scénario A Dans la Figure 3.11 ci-après, on observe globalement que le pourcentage d'utilisation du CPU sur le FE et le BE est identique avec et sans DISC. Ceci traduit un impact négligeable sur la consommation CPU des modules DISC lorsqu'on n'effectue pas de raccourci. En réalité, les modules in_drv, in_tc et out_tc effectuent des opérations basiques à savoir consulter les informations de la session encours (kvs) pour vérifier si un raccourci est en cours.

Pour chaque paquet entrant entre le client et le FE, on a 189 ns d'ajout dans in_drv et 231ns dans in_tc. Pour chaque paquet sortant entre le FE et le client, on a 329 ns dans out_tc.

Au niveau de l'application (FE), le feHook vérifie les entêtes des réponses pour déterminer s'il y a raccourci ou pas ; cette opération n'est pas couteuse et est de l'ordre de 350ns. Sur le le BE, il vérifie s'il peut faire un raccourci et cette vérification prend 150ns. La Table 3.2 ci-après regroupe le temps passé dans chaque hook.

Pour avoir un ordre de grandeur, ces temps d'exécution dans les composants DISC peuvent être rapportés au temps d'exécution de la remontée d'un paquet entre le driver et le niveau utilisateur, qui est de l'ordre de 29 microsecondes.

Composants	temps (ns)
FE_IN_TC	231.078
FE_OUT_TC	329.505
FE_IN_DRV	189.687
FE_HOOK	350
BE_IN_DRV	284.462
BE_HOOK	150

TABLEAU 3.2 – *Temps d'exécution dans les composants* DISC

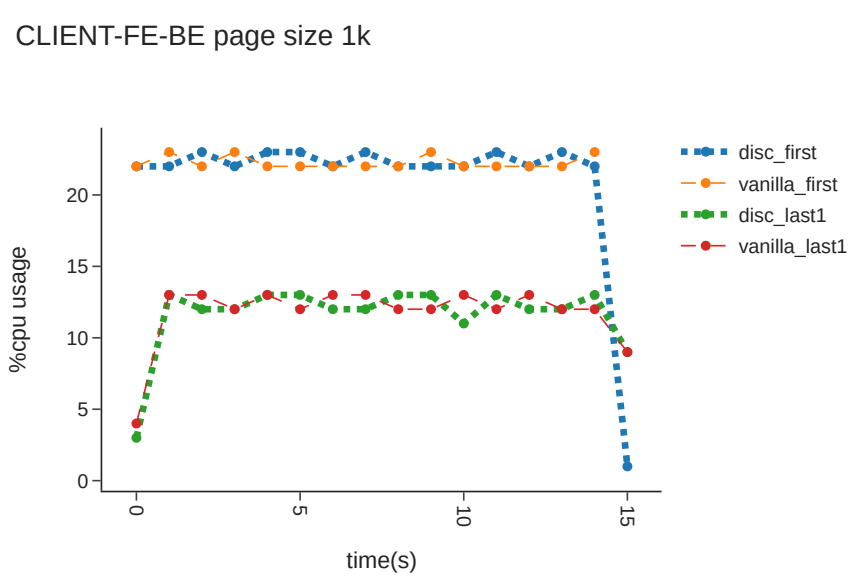


FIGURE 3.11 – *Impact de* DISC *sur le CPU, sans raccourci ni translation de SN*

Impact sur le débit - scénario A Dans la figure 3.12, on voit que le débit n'est presque pas impacté dans le scénario A (lorsqu'il n'y a ni raccourci ni translation de SN).

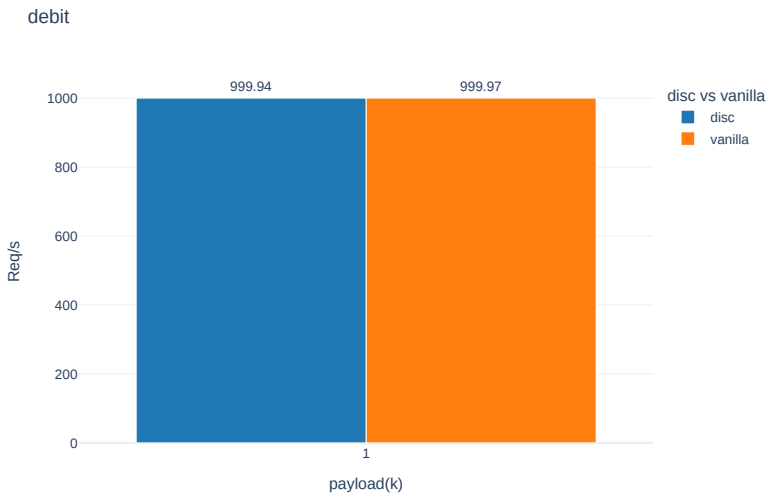


FIGURE 3.12 – *Impact de* DISC *sur le débit, sans raccourci ni translation de SN*

Impact sur la latence - scénario A Dans la figure 3.13, on a globalement une latence faiblement impactée qui s'observe par une superposition des courbes.

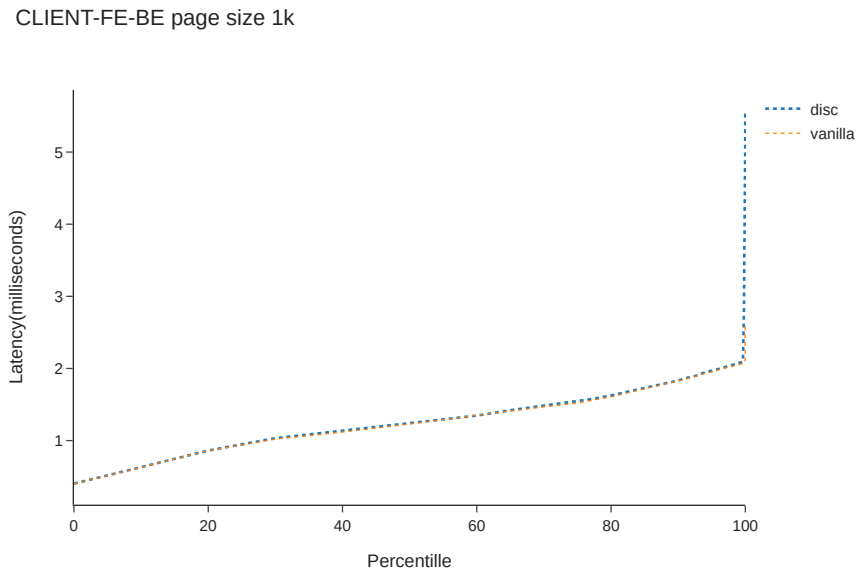


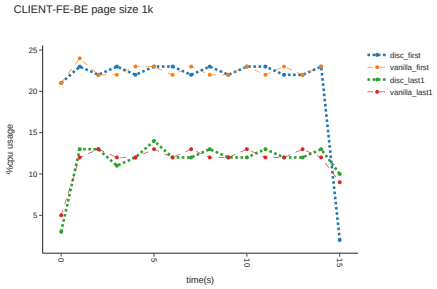
FIGURE 3.13 – *Impact de* DISC *sur la latence, sans raccourci ni translation de SN*

La Table 3.3 détaille ces latences et montre des écarts très faibles jusqu'à 99ème percentile. Notons que ces mesures sont effectuées avec un client local au réseau, avec un temps d'aller-retour d'un paquet entre le client et le FE qui est de 45us (un temps qui serait bien plus long sur Internet).

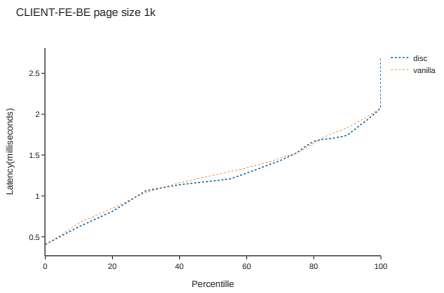
Percentille	latence DISC	latence vanilla
50.000%	1.24ms	1.23ms
75.000%	1.54ms	1.52ms
90.000%	1.83ms	1.83ms
99.000%	2.07ms	2.04ms
99.900%	3.96ms	2.08ms
99.990%	5.57ms	2.58ms
99.999%	5.57ms	2.58ms
100.000%	5.57ms	2.58ms

TABLEAU 3.3 – *Distribution de la latence vanilla vs* DISC*, sans raccourci ni translation de SN*

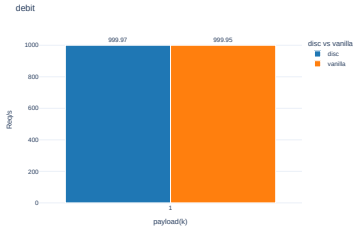
On considère ensuite le scénario B où on effectue un raccourci uniquement pour la première requête, le but ici étant d'évaluer le coût de la translation des SN. Nous rappelons que le client accède à une page de 16k puis accède à un ensemble de pages de 1k, la fréquence des requêtes étant maintenue comme dans le précédent test. Fait ainsi, le FE va translater tous les acks après que le raccourci de la première page ait eu lieu. Les métriques seront toujours les mêmes à savoir latence, débit et CPU.



(a) Impact sur le CPU



(b) Impact sur la latence



(c) Impact sur le débit

FIGURE 3.14 – *Evaluations sans raccourci avec translation de SN*

Percentille	latence DISC	latence vanilla
50.000%	1.18ms	1.25ms
75.000%	1.53ms	1.52ms
90.000%	1.74ms	1.84ms
99.000%	2.04ms	2.05ms
99.900%	2.08ms	2.08ms
99.990%	2.68ms	2.09ms
99.999%	2.68ms	2.09ms
100.000%	2.68ms	2.09ms

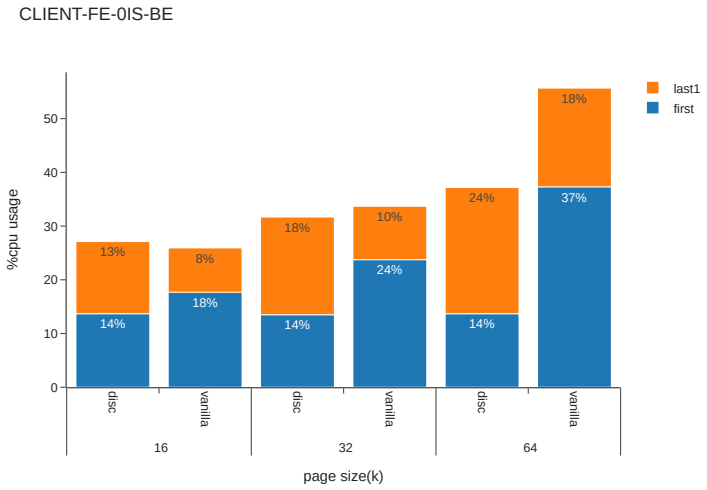
TABLEAU 3.4 – *Evaluations sans raccourci avec translation de SN*

Au regard des résultats de la Figure 3.14 et de la Table 3.4, on constate que les composants de DISC qui interviennent dans DISC et dans la translation d'estampilles rajoutent un overhead négligeable. Notons que de façon surprenante, les mesures de la Table 3.4 paraissent meilleures que celles de la Table 3.3, mais ceci est vrai pour des mesures après le 99$^{\text{ème}}$ percentile et cela rentre dans les fluctuations d'une évaluation en conditions réelles.

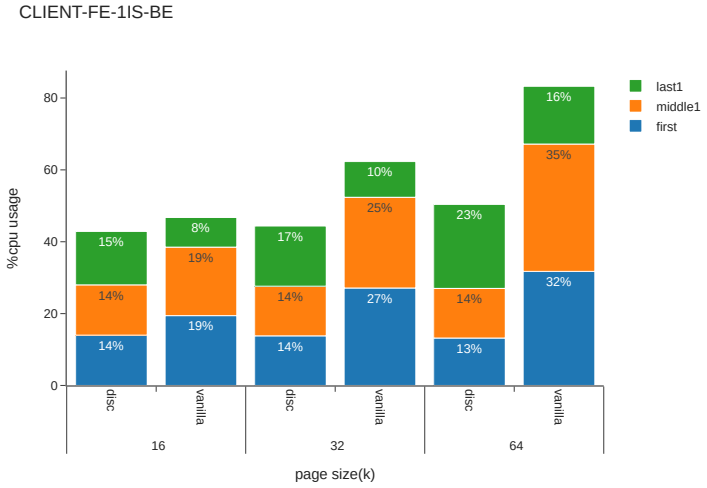
3.6.3 Consommation de ressources

La configuration utilisée est la suivante : FE-nIS-BE (on fait varier le nombre de niveaux IS entre 0 et 2).

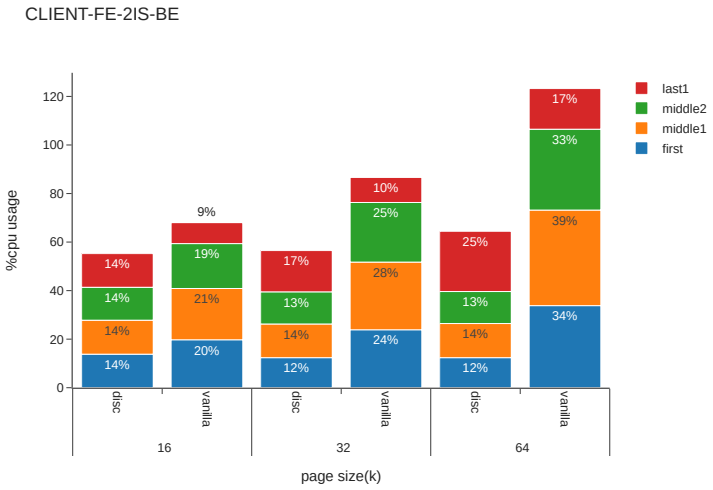
Chaque serveur utilise seulement 1 coeur. La taille de la payload de réponse varie de 16k à 64k. L'injecteur de charge est configuré pour assurer un débit de requêtes constant de 600 requêtes par seconde.



(a) Pas de serveur IS



(b) Un serveur IS

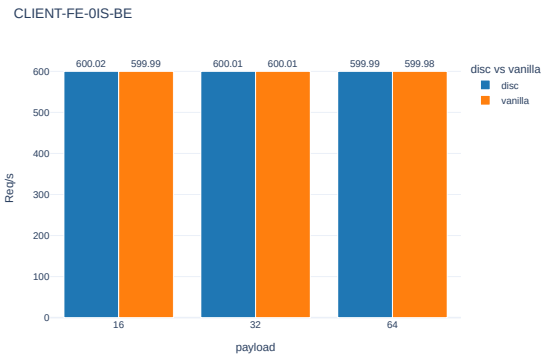


(c) Deux serveurs IS

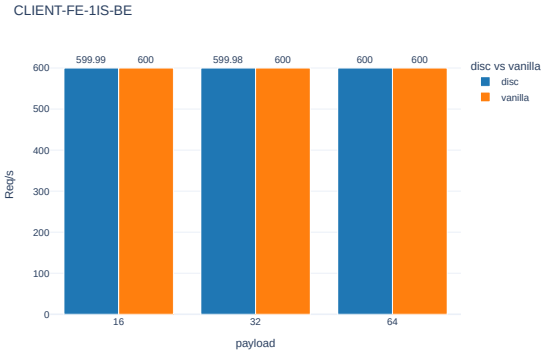
FIGURE 3.15 – *Utilisation du CPU avec et sans raccourci*

Nous rappelons que first est une instance de FE, middle1 et middle2 sont des instances de IS et last1 une instance de BE.

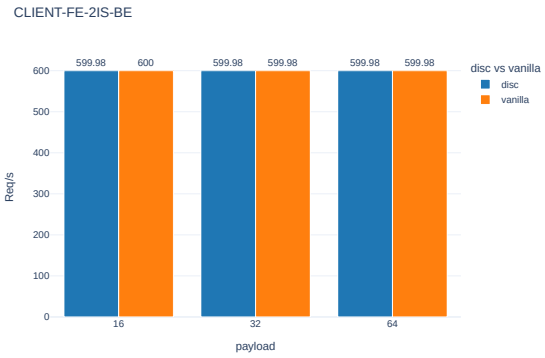
On observe sur les graphiques (Figure 3.15) une baisse globale d'utilisation du CPU lorsqu'on effectue un raccourci d'une part, et surtout lorsqu'il y a des serveurs intermédiaires (IS) et d'autant plus que les payloads sont importantes. D'autre part, on observe une augmentation de la charge sur le serveur last1 due à l'implémentation de DISC et surtout à la gestion des acquittements (qui sont traités sur first dans vanilla). La baisse de charge au niveau des serveurs first, middle1 et middle2 s'expliquent par le fait qu'ils ne servent plus de relais pour le transfert de la payload et cette baisse est de plus en plus significative lorsque la taille de la payload augmente. C'est que l'on constate dans les configurations 3.15b et 3.15c où la charge globale baisse pratiquement de moitié à partir de 64k. On gagne 18% dans la configuration 3.15a à partir de 64k.



(a) Pas de serveur IS



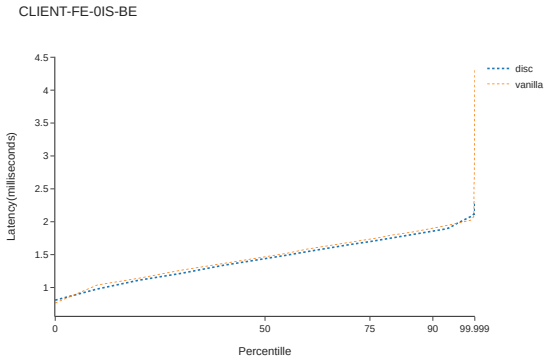
(b) Un serveur IS



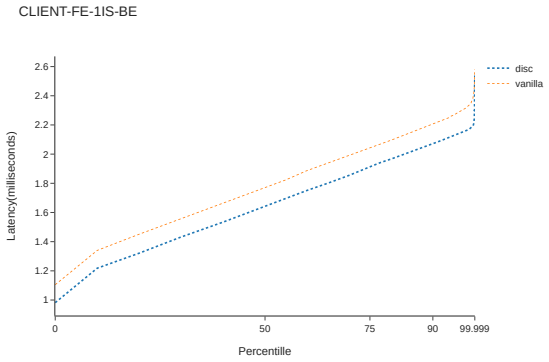
(c) Deux serveurs IS

FIGURE 3.16 – *Débit avec et sans raccourci*

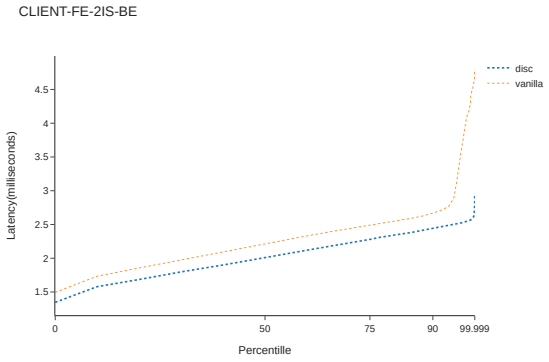
Les courbes de la Figure 3.16 nous montre que DISC permet de maintenir le débit.



(a) Pas de serveur IS



(b) Un serveur IS



(c) Deux serveurs IS

FIGURE 3.17 – *Distribution de la latence avec et sans raccourci pour une payload de 64k*

Pour la latence (Figure 3.16), on peut faire le même constat pour la configuration 3.17a et dire que DISC permet de maintenir la latence. Cependant, elle permet même d'améliorer la latence dans les configurations 3.17b et 3.17c.

3.6.4 Déplacement de bottleneck et amélioration de la scalabilité

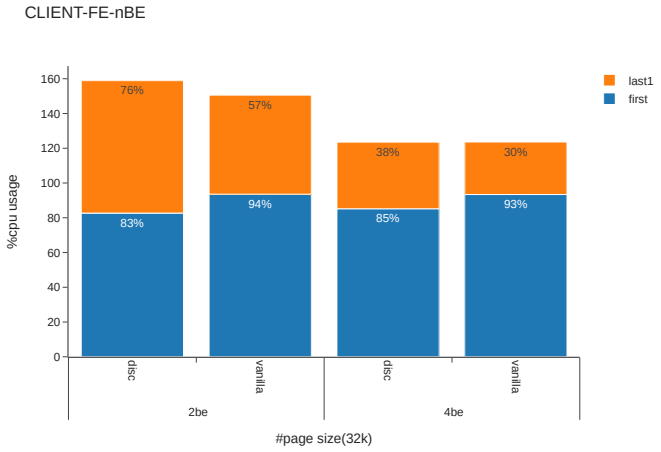
L'objectif de cette expérimentation est de voir/montrer comment notre approche permet de déplacer le bottleneck du FE vers le BE et d'améliorer ainsi la scalabilité générale de l'application. Pour cette expérimentation, nous considérons l'architecture Client-FE-BE ou le FE s'exécute avec

un seul coeur, ackSender est limité à 2% de CPU et s'exécute sur le même coeur que le FE. Pour le BE, nous faisons varier son nombre de slaves de 2 à 4 et chaque slave s'exécute séparément sur un coeur. Pour le client, nous calibrons la charge de l'injecteur de telle sorte que le first soit saturé ; ici le first est saturé avec la configuration de l'injecteur calibré à 14000 requêtes par secondes. La taille de la payload est 32k (et provoque donc des raccourcis).

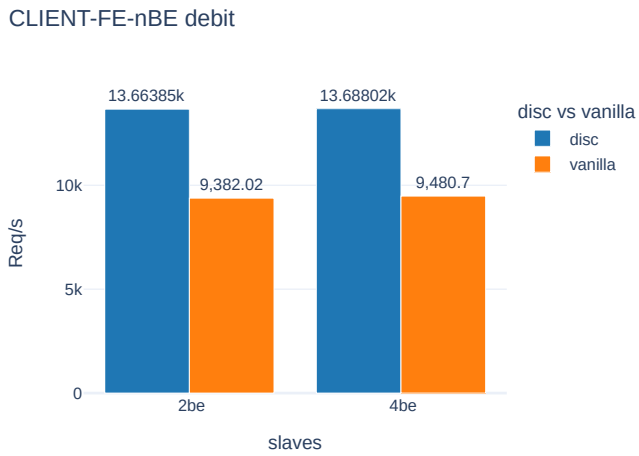
Nous traçons deux graphes :

— Débit (y) en fonction du nombre de BE (x), pour DISC et la version vanilla.

— Latence(y) en fonction du nombre de BE (x), pour DISC et la version vanilla.



(a) Utilisation du CPU, DISC *vs vanilla*



(b) Débit, DISC *vs vanilla*

FIGURE 3.18 – *Débit et utilisation du CPU,* DISC *vs vanilla, en variant le nombre de BE*

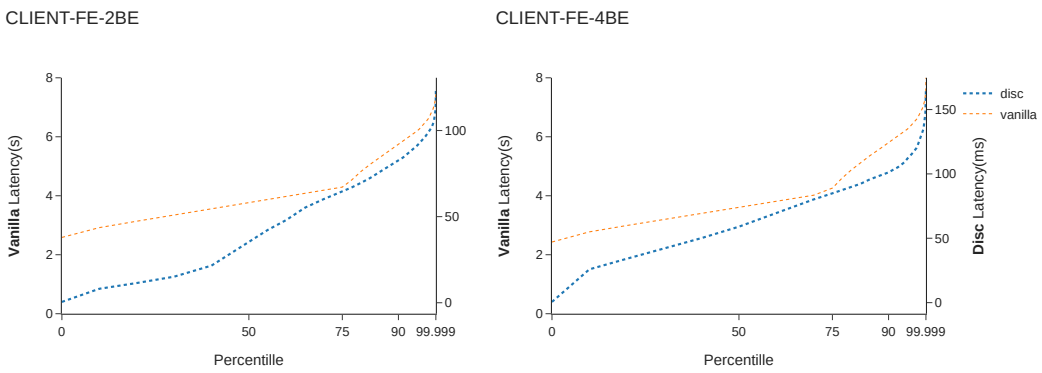


FIGURE 3.19 – *Latence,* DISC *vs vanilla, en variant le nombre de BE*

	slaves=2		slaves=4	
	Disc	Vanilla	Disc	Vanilla
50.000%	35.20ms	3.76s	59.01ms	3.61s
75.000%	64.51ms	4.29s	84.93ms	4.26s
90.000%	82.69ms	5.74s	101.18ms	5.79s
99.000%	102.33ms	6.86s	131.20ms	6.94s
99.900%	111.61ms	7.28s	154.62ms	7.40s
99.990%	116.74ms	7.39s	163.97ms	7.83s
99.999%	120.51ms	7.44s	165.50ms	8.00s
100.000%	123.78ms	7.44s	165.50ms	8.00s

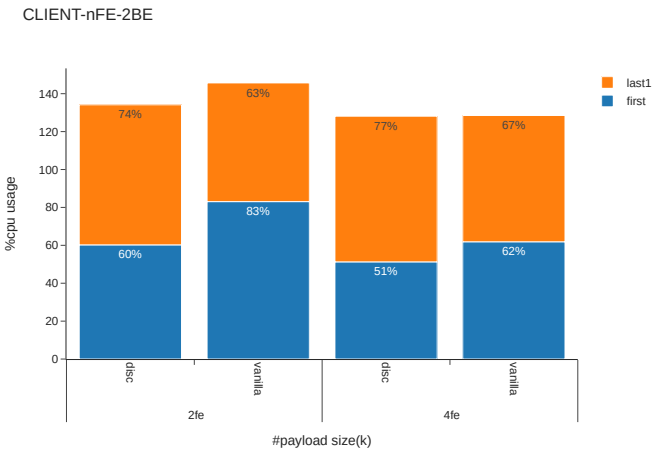
TABLEAU 3.5 – *Distribution de la latence pour 8 slaves alloués au be*

On observe sur la courbe 3.18a un taux d'utilisation du CPU pour first plus élevé dans la configuration vanilla (et même proche de la saturation) ; dans le même temps, sur la Figure 3.18b le débit pour vanilla plafonne à 9k requêtes par seconde. Ainsi, vanilla n'arrive pas à assurer le débit de 14k requêtes par seconde et l'augmentation du nombre de BE ne change rien. Ceci est du à la saturation du first. Par contre dans la configuration DISC, le CPU de first est moins élevé et ne sature pas (bien qu'il traite plus de requêtes), et il arrive à assurer le débit de 14k requêtes par seconde. A noter qu'avec DISC, si le CPU est moins élevé pour first, cela se traduit par une augmentation sur le tier BE par rapport à vanilla (principalement en raison de la gestion des acks). Dans les deux configurations, la charge sur le BE se répartit sur les slaves (et la charge du tier BE est rapportée au nombre de slaves, ce qui explique pourquoi elle est divisée par 2).

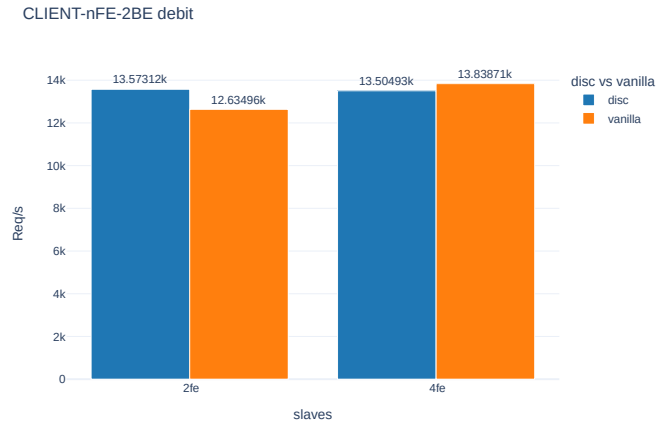
On peut donc déduire ici que DISC permet de réduire la charge sur le FE, ce qui permet de répondre à la charge soumise sans saturation du FE. Cette réduction sur le FE se traduit aussi par une augmentation de la charge sur le BE, mais cette charge peut être amortie sur plusieurs BE.

Pour la latence (Figure 3.19 et la Table 3.5), on peut voir qu'avec DISC, le temps de réponse reste meilleur par rapport à vanilla qui subit une forte dégradation en raison de la saturation du FE.

Nous considérons maintenant une variante de l'expérimentation précédente où nous fixons le nombre de slaves du BE à 2 et où nous faisons varier le nombre de slaves du FE de 2 à 4. Nous maintenons le même profil de charge et nous traçons les mêmes courbes comme dans l'expérimentation précédente.



(a) Utilisation CPU DISC *vs vanilla*



(b) Débit DISC *vs vanilla*

FIGURE 3.20 – *CPU et débit avec et sans* DISC*, avec 2 BE et de 2 à 4 slaves FE*

On constate sur la Figure 3.20a qu'en répartissant la charge du FE sur plusieurs slaves, vanilla arrive réduire la charge CPU sur le tier FE et à absorber la charge soumise de 14k requêtes par seconde, ce qui confirme que le FE était bien le bottleneck. Avec 2 slaves sur le FE, la charge soumise est presque absorbée, mais pas complétement et on constate sur la Figure 3.21 et la Table 3.6 que la latence reste perturbée. Pour 4 slaves sur le FE, les latences redeviennent normales.

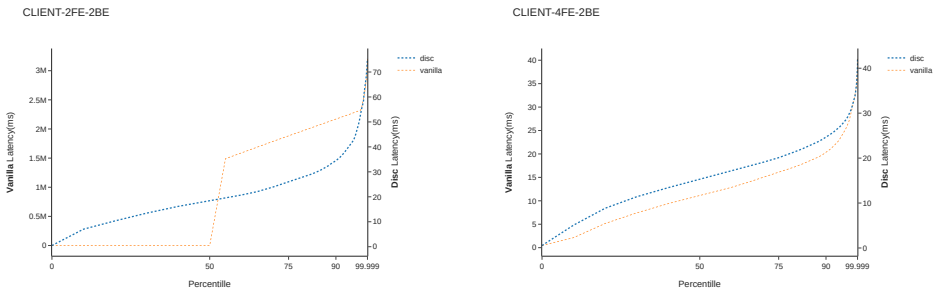


FIGURE 3.21 – *Latence avec et sans* DISC*, avec 2 BE et de 2 à 4 slaves FE*

	slaves=2		slaves=4	
	Disc	Vanilla	Disc	Vanilla
50.000%	18.35ms	20.24ms	15.30ms	11.10ms
75.000%	26.01ms	1.88s	20.09ms	16.06ms
90.000%	34.46ms	2.17s	24.64ms	20.30ms
99.000%	62.78ms	2.59s	33.22ms	31.84ms
99.900%	71.87ms	3.02s	38.49ms	37.31ms
99.990%	74.56ms	3.16s	40.13ms	39.71ms
99.999%	75.01ms	3.19s	41.92ms	40.29ms
100.000%	75.26ms	3.20s	42.08ms	40.29ms

TABLEAU 3.6 – *Latence avec et sans* DISC*, avec 2 BE et de 2 à 4 slaves FE*

3.6.5 Breakdown Time

L'objectif de cette expérimentation est de mesurer la consommation CPU des différents composants rajoutés par DiSC.

L'architecture est la suivante : Client-FE-BE. Nous utilisons ici un microbenchmark avec wrk2 [wrk]. Nous injectons une charge avec 10 clients, chaque client émet des requêtes avec une fréquence de 20 requêtes par secondes et dont la réponse est de taille 32k. Chaque serveur a un seul slave et le slave n'utilise qu'un seul coeur de sa machine.

Sur le FE, on s'intéresse aux composants : feHook, acksender, in_tc, out_tc et in_drv comme on peut le voir sur la Figure 3.22. On s'appuie sur l'outil bpftrace [bpf07] pour déterminer les temps d'exécutions des hooks dans l'espace utilisateur. Pour les modules eBPF en espace noyau, nous utilisons bpftool [bpf06] qui permet de récolter les temps d'exécutions des modules eBPF lorsque le profiling est activé.

Hook noyau du FE. On observe dans les modules ebPF un temps d'exécution global de 47875570 ns pour 46319 invocations de in_drv, 9119468 ns pour 40034 invocations de out_tc, 6840230 ns pour 27999 invocations de in_tc. Le Tableau 3.7 permet de résumer le temps CPU moyen par paquets traités dans les modules eBPF.

Composants	Temps
in_tc	244.3 ns
out_tc	227.8 ns
in_drv	1.033 us

TABLEAU 3.7 – *Temps moyens dans les modules eBPF sur le FE*

in_drv fait plus de travail que in_tc et out_tc. En effet, il prend la décision si un paquet remonte dans la pile réseau du FE (si il comprend des données), il calcule les translations à effectuer sur les acks entrants à destination du FE et il dépose dans le kvs les acks qui doivent être transmis au BE et qui seront envoyés par le ackSender. Il faut donc 1.033us pour que ackSender puisse voir un message dans la map kvs pour le BE.

in_tc et out_tc se chargent de mettre à jour la couche TCP des paquets en modifiant les champs ack_seq et seq en respectant le format réseau. Il faut en effet passer au format machine, ajouter les translations calculées par in_drv et enfin remettre au format réseaux afin de mettre à jour les champs.

Hook espace utilisateur du FE. Pour les modules en userspace, le feHook s'exécute 120011 us pour un total de 4000 invocations qui correspondent au nombre de réponses dans le scénario joué soit une moyenne de 30 us par réponse. En réalité, le feHook fait deux appels systèmes, un pour consulter la map kvs et l'autre pour mettre à jour la map avec les informations de raccourci. Plus tard, dans une fonction de rappel, le feHook vérifie la map pour savoir s'il peut envoyer le signal de transfert au BE. Cette opération pour notre scénario a été appelée 4000 fois avec un temps d'exécution global de 56187 us soit en moyenne 14 us. Le ackSender pour sa part, grâce au mécanisme epoll du système, est notifié chaque fois que in_drv pose un message (ack) dans la map. Pour ce scénario, on a eu 13212 appels avec un temps global de 187286 us soit en moyenne 14 us. Notons ici que le nombre d'appels du acksender est corrélé avec le nombre de acks émit par le client. Si le client n'agrège pas suffisamment les acks, ceci peut entrainer une charge inutile à la fois dans le FE et le BE.

Hook noyau du BE. Au niveau du BE, nous avons le ctr_message_manager qui se charge d'aiguiller les messages venant du FE vers le bon processus du BE. Il intervient au niveau du pilote et intercepte les messages venant du FE. Pour ce scénario, nous avons enregistré un total de 37632 paquets pour un temps d'exécution global de 72532206 ns soit en moyenne 1.9 us par paquet traité. Notons ici que ce temps est influencé par le traitement des raccourcis qui consiste à décoder le message afin de sélectionner le bon ringbuffer du processus en attente de signal de transfert. Il faut donc en moyenne 1.9 us pour qu'un message venant du FE une fois dans le pilote de la carte réseau puisse arriver au processus en attente. Ce temps est relativement petit comparé à si le message devait traverser toute la stack.

Hook espace utilisateur du BE. Au niveau du BE, dans l'espace utilisateur, Ring-1 est directement monitoré par le processus applicatif par un mécanisme epoll (event_loop). Lorsqu'un message est disponible (demandant une transmission), il le récupère et fait appel à payload_sender qui va enclencher la transmission. payload_sender a été appelé 8805 fois, pour un temps global de 228930 us soit en moyenne 26 us. Ce chiffre peut paraitre grand, mais il n'en est rien, car lorsque payload_sender est appelé, il récupère les informations de transfert dans l'arbre, construit les trames TCP puis les charge dans le ringbuffer de transfert (umem) et enfin fait un appel système sendto qui va provoquer le transfert de la payload contenue dans umem. Ce temps est en réalité de l'ordre de grandeur du temps qu'aurait pris le processus applicatif s'il n'y avait pas de raccourci. Pour le beHook, il ajoute les informations de raccourci dans les headers des réponses. Cette opération est réalisé 4000 fois pour un temps CPU global de 37334 us soit en moyenne 9 us. La Figure 3.22 montre un aperçu de la répartition du temps pris par les composants de DISC.

Pour avoir un ordre de grandeur, ces temps d'exécution dans les composants DISC peuvent être rapportés pour la partie noyau au temps d'exécution de la remontée d'un paquet entre le driver et le niveau utilisateur qui est de l'ordre de 29 microsecondes. Et pour la partie utilisateur, on peut rapporter ces temps d'exécution à la latence d'une requête sur l'architecture Client-FE-BE en configuration vanilla qui est de l'ordre de 1 ms (Table 3.3).

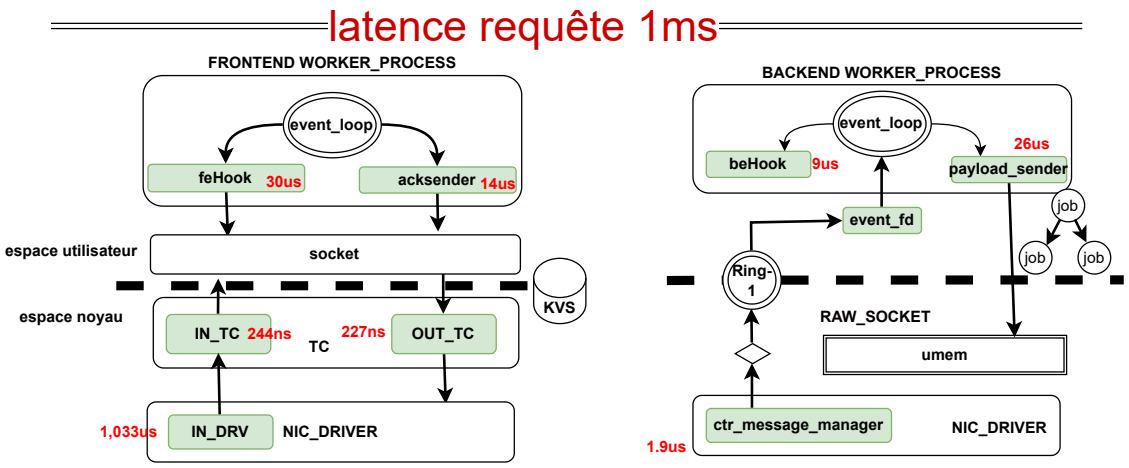


FIGURE 3.22 – *Temps CPU des modules DisC*

3.6.6 Evaluation de TLS

Ici, nous voulons observer l'impact du support de TLS dans DISC. Pour cela, nous utilisons notre microbenchmark Wrk. Nous utilisons comme payload des images de taille 16k. L'injecteur de charge est un client à débit constant de 300 requêtes par secondes.

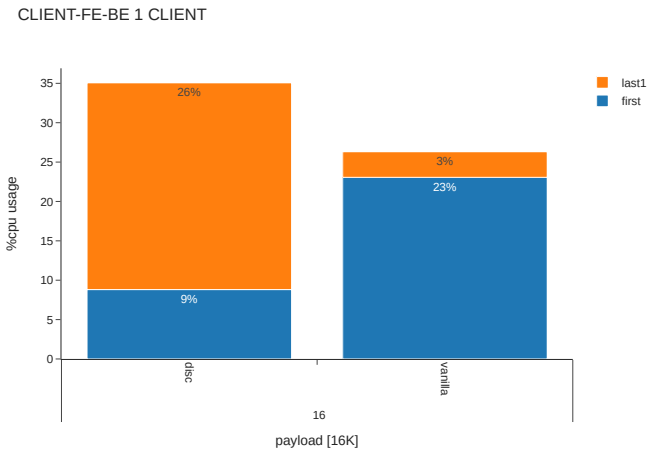


FIGURE 3.23 – *Evaluation de la charge CPU avec TLS,* DISC *vs vanilla*

Nous observons que la charge CPU a diminué significativement sur le FE, comme pour les expérimentations précédentes du fait que les payloads ne passent plus par ce serveur. De même, ces payloads ne sont plus chiffrées sur le FE, mais sur le BE. Cette charge de chiffrement est donc reportée sur le BE et nous observons une forte augmentation de sa charge CPU. Cette augmentation est anormalement grande et nous suspectons une mauvaise méthode de chiffrement dans notre implémentation où nous chiffrons paquets par paquets alors qu'il est possible de chiffrer des records TLS plus grands.

3.6.7 Résultats avec Specweb 2009

Specweb est notre macrobenchmark et on s'intéresse à l'application Ecommerce, car elle fait intervenir plus de contenu statique non exploité par les serveurs intermédiaires (payloads finales). Dans sa version native, SpecWeb décrit uniquement le serveur Web, un serveur de stockage (optionnel) et un serveur métier. Le paramétrage et les choix de déploiements sont laissés à la charge de l'utilisateur. Pour nos tests, nous construisons un déploiement qui se rapproche d'un déploiement dans le Cloud Amazon. Typiquement un déploiement dans le Cloud Amazon est constitué d'une API gateway, d'un ELB (elastic loadbalancer) et d'un ou plusieurs EC2 (elastic compute node). Par similitude sur la Figure 3.24, le Reverse proxy joue le rôle de l'API gateway, le LoadBalancer joue le rôle ELB et les autres serveurs le rôle des EC2.

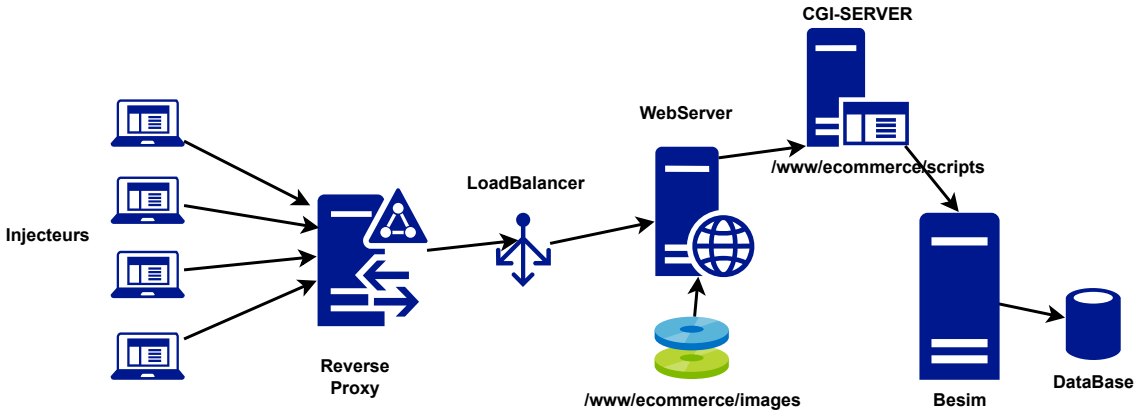


FIGURE 3.24 – *Déploiement Specweb avec Reverse proxy et LoadBalancer*

Le scénario ici est un ensemble d'utilisateurs effectuant des achats. Pour ce scénario, nous générons les images avec les tailles pouvant aller jusqu'à 64k. D'après le site HTTP archive, la taille moyenne des images pour un site web moderne est de 40k. Chaque composant, Reverse proxy, LoadBalancer et WebServer exécute une instance de nginx câblée sur un coeur. Les images sont stockées sur le serveur web et ce dernier est relié à Besim via le serveur CGI. Nous appliquons une charge de 500 utilisateurs simultanés au niveau de l'injecteur de charge. Comme métrique, nous collectons la consommation CPU sur les serveurs FE (Reverse Proxy), IS (LoadBalancer) et BE (WebServer). Cette évaluation est conduite sans support de TLS.

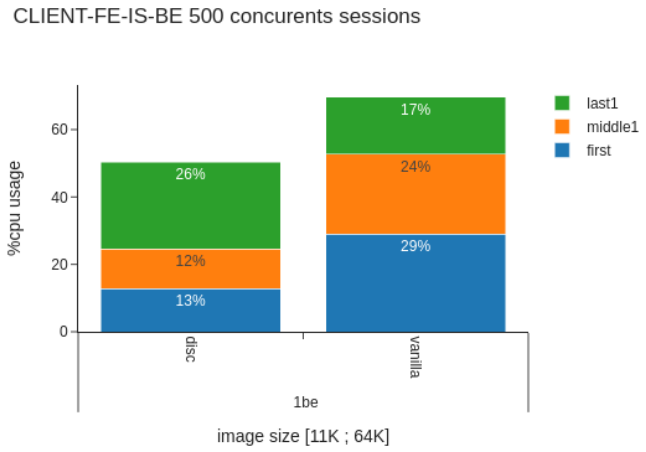


FIGURE 3.25 – *Charge CPU pour Specweb,* DISC *vs vanilla*

On observe sur la courbe de la Figure 3.25 une baisse de CPU à la fois sur le FE et le IS et une

augmentation de CPU au niveau du BE. Cette augmentation est principalement due à la gestion des acks. Le bilan global est un gain de CPU.

3.7 Conclusion

Dans ce chapitre, nous avons proposé DISC comme approche permettant d'optimiser les communications dans un datacenter. DISC propose une telle optimisation à travers la notion de raccourci, en permettant à un serveur dans une architecture multi-tiers de court-circuiter les tiers qui le précédent dans cette architecture. Un raccourci est mis en place pour les réponses retournant une payload qui n'est pas exploitée (lue ou modifiée) par les tiers précédents. L'implantation de DISC nécessite une intégration fine dans TCP pour garantir une gestion cohérente des numéros de séquence et des acquittements. Les évaluations conduites montrent les bénéfices qu'apportent ces raccourcis, en terme de ressources de calcul sur les serveurs contournés.

La mise en place des raccourcis nécessite la modification des logiciels applicatifs déployés sur les tiers de l'architecture (feHook, isHook, beHook). Nous proposons dans le chapitre suivant une approche qui permet de s'affranchir de toute modification de ces applications.

4

Tampon mémoire à la demande

Ce chapitre décrit la conception, les détails d'implémentation et l'évaluation de ODB

Sommaire

4.1 Motivations

Nous avons présenté dans le chapitre précédent notre première contribution DISC, un protocole accompagné d'un système pour réaliser du zero-copy entre les noeuds d'un cluster d'application multi-tiers. DISC fait l'hypothèse que les applications multi-tiers interagissent suivant un schéma requête/réponse dans lequel les requêtes et réponses sont composés d'un header et d'une payload. Les headers peuvent être exploités par les tiers applicatifs pour diverses raisons (statistiques par exemple). Les payloads quant-à elles peuvent être soit exploitées (par les tiers intermédiaires) soit finales (non exploitées). DISC évite la copie des données finales entre les machines en permettant que le serveur possédant la donnée l'envoie directement au client externe sur la session TCP initialement ouverte entre le client et le serveur frontal de l'application. Pour y arriver, DISC nécessite la modification (bien que modeste) des applications, ce qui pourrait freiner son adoption.

L'objectif de notre deuxième contribution, On-Demand Buffer (ODB), est de proposer une solution zero-copie entre les noeuds d'une application multi-tier qui ne nécessite aucune modification des applications. ODB détecte de façon transparente le besoin d'exploitation d'une payload par les tiers intermédiaires et les rapatrie uniquement dans ce cas.

4.2 Principe de conception de ODB

4.2.1 Architecture générale

L'idée de base derrière ODB est d'intercepter de façon transparente les appels `read()`, `write()` (et leurs différentes variantes) sur les sockets. Lorsqu'un serveur A appelle `write()` pour envoyer une donnée à un serveur B, ODB intercepte l'appel afin de stocker la donnée à envoyer dans une zone de mémoire dédiée à ODB. Ensuite, ODB envoie tout simplement une référence vers cette zone au serveur B.

Lorsque ce dernier réalise un `read()` pour lire la donnée envoyée par A, ODB intercepte l'appel et stocke en local (sur B) la référence émise par A. Ensuite, sur la machine B, ODB protège contre les lectures et les écritures le buffer `buff` de destination passé en paramètre par l'application dans l'appel `read()`. ODB a préalablement positionné un traitant de défaut de pages sur B. De cette façon, toute tentative d'accès en lecture ou écriture à `buff` génère un défaut de page qui est attrapé par ODB. Le traitant déprotège `buff`, rapatrie les données nécessaires à l'application, et la copie dans `buff`.

Sur B, lorsque l'accès de l'application à `buff` se fait via l'appel à `write()` pour envoyer les données vers une machine interne au cluster, si `buff` n'a pas été accédé, ODB envoie tout simplement la référence initialement reçu de A. Aucun rapatriement de données n'est effectué depuis A dans ce cas.

Sur le serveur frontal (en liaison directe avec le client externe), deux options sont possibles : le rapatriement des données depuis le serveur (A) contenant les données ou l'utilisation de DISC pour l'émission de la donnée depuis A dans la session TCP qui existe entre la machine frontale et le client extérieur. Pour faciliter la compréhension de ce chapitre, nous présentons dans la suite ODB en considérant la première option.

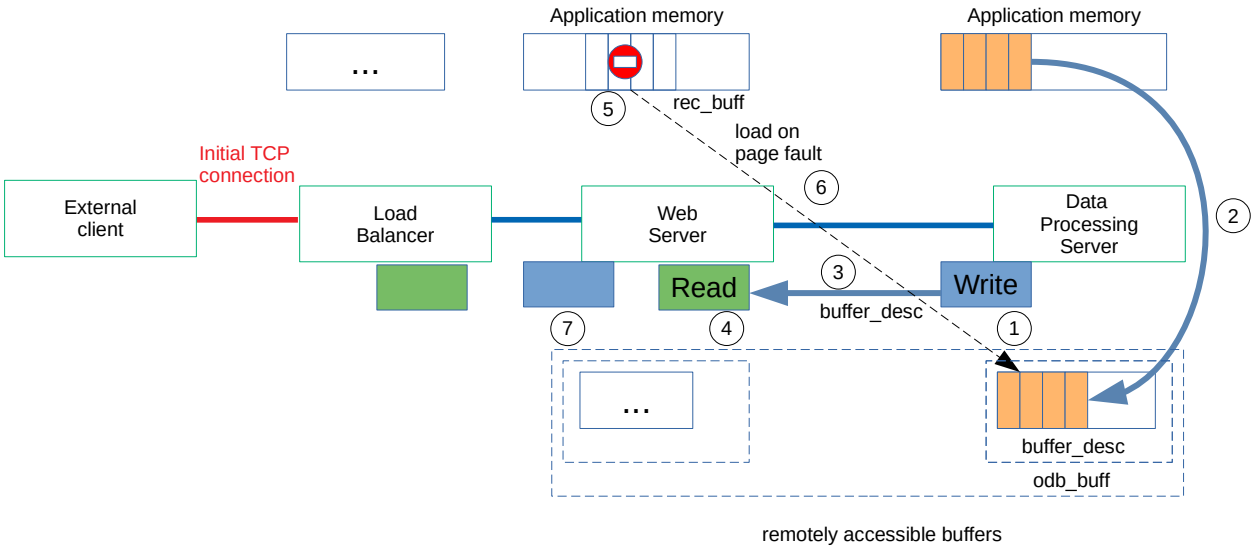


FIGURE 4.1 – *Architecture de ODB*

La Figure 4.1 présente l'architecture générale de ODB que nous illustrons avec le même exemple applicatif multi-tiers utilisé dans le chapitre consacré à DISC. Pour détailler le fonctionnement de ODB, nous allons nous focaliser sur l'émission de la réponse, à savoir de *Data Processing Server* vers *Load Balancer*, en passant par *Web Server*. Notons que ODB peut également être appliqué aux données envoyées dans le sens inverse (requête). La Figure 4.1 s'interpréte comme suit :

1. *Data Processing Server* émet des données en invoquant `write()`. Cet appel prend en paramètre un pointeur *buff* vers une zone mémoire située dans l'espace d'adressage de l'application. Cette zone mémoire contient les données à envoyer.
2. ODB intercepte `write()` de façon transparente et stocke les données émises dans un buffer dédié appelé *odb_buff*. Ce dernier sera utilisé par ODB pour tout envoi de données par cette application (processus). Pour localiser *buff* dans *odb_buff*, ODB construit un descripteur que nous appelons *odb_desc*.
3. Au lieu d'envoyer le contenu de *buff* à *Web Server*, ODB lui envoie *odb_desc*. On parle alors d'un envoi de donnée virtuelle dans le jargon ODB.
4. *Web Server* tente de recevoir des données en provenance de *Data Processing Server* en effectuant un appel `read()`. Ce dernier prend en paramètre un pointeur *rec_buff* vers une zone mémoire allouée par l'application. Cette zone mémoire est sensée être la destination des données reçues.
5. ODB intercepte l'appel `read()` afin d'interpréter *odb_desc*. ODB enregistre dans un annuaire le mapping *rec_buff* $\rightarrow$ *odb_desc*. Puis, il utilise les mécanismes de protection mémoire pour interdire tout accès à la zone mémoire pointée par *rec_buff*, qui n'a en réalité pas été modifiée par `read()`.
6. Si *Web Server* essaie d'accéder à *rec_buff*, une faute de page sera générée et le traitant de cette faute de page (qui fait partie de ODB) rapatrie les données. Pour cela, il se base sur l'annuaire de mappings *rec_buff* $\rightarrow$ *odb_desc*.
7. Lorsque *Web Server* émet des données, ODB intercepte l'appel `write()` et considère trois cas. (i) Soit les données à envoyer sont effectivement présentes sur la machine (il s'agit de nouvelles données générées par la machine locale) ; (ii) Soit les données proviennent de *Data Processing Server* mais ont déjà été rapatriées. Dans ces deux premiers cas, la reimplantation de `write()` par ODB procède comme en (2) et (3). (iii) Soit les données ne

sont pas locales à la machine. Il s'agit alors d'un *rec_buff* jamais accédé par *Web Server*. Dans ce cas, ODB modifie le comportement de `write()` pour forwarder *odb_desc* tel qu'il l'avait reçu. La primitive `write()` peut donc envoyer *virtuellement* des données sockées sur *Web Server*, sur *Data Processing Server*, ou un mix des deux.

Nous détaillons dans la suite les principes de mise en œuvre de chaque étape de phase.

4.2.2 Gestion des buffers

ODB implémente deux services de gestion des buffers. Le premier est un service d'accès à distance aux buffers (Remotely Accessible Buffers ou RAB). Ce service permet d'enregistrer (par copie) les buffers mémoire lorsqu'ils sont envoyés virtuellement (avec `write()`). RAB stocke en mémoire (dans *odb_buff*) ces buffers et retourne un identifiant *odb_desc*. Ce dernier est composé de l'adresse réseau de RAB (IP et port), l'offset/localisation du buffer dans *odb_buff* et la taille du buffer que l'application souhaitait envoyer.

Le second service, que nous appelons Remote Zone Mapping ou RZM, a deux tâches principales. Premièrement, il gère le mapping des zones mémoires locales (reçues de l'appel à `read()`) avec les *odb_desc*. Le mapping d'une zone mémoire locale avec un buffer distant comprend l'adresse de la zone mémoire locale, la taille que souhaitait lire l'application, et le *odb_desc* correspondant. Deuxièmement, RZM gère la protection du buffer local, les défauts de page et le rapatriement des données en cas de défaut de page.

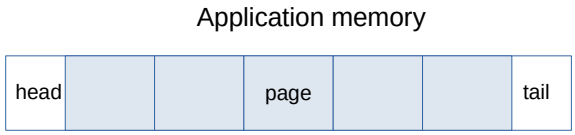


FIGURE 4.2 – *Nom alignement de pages*

Actuellement, les architectures matérielles ne permettent qu'une protection de la mémoire à la granularité d'une page (4KB) et l'adresse de la zone mémoire doit être alignée sur une adresse de page. Or le buffer utilisé par l'application lors de l'appel à `read()` peut ne pas être aligné. Cette situation nous pousse à organiser le buffer en trois parties : *head*, *tail* et *corps*. head est la portion se situant sur la première page du buffer lorsqu'il n'est pas aligné ; tail se situe sur la dernière page du buffer lorsque celle-ci n'est pas pleine ; corps est l'ensemble intermédiaire de pages contigües, dont la taille est multiple d'une page (Figure 4.2). Notre stratégie pour rapatrier le buffer dans cette situation est la suivante. Lors d'une lecture, ODB récupère les parties head et tail auprès de RAB, et protège uniquement corps. Les données correspondantes à ce dernier sont laissées distantes. Puis ODB met à jour RZM (Figure 4.3) en enregistrant le mapping corps $\rightarrow$*odb_desc*.

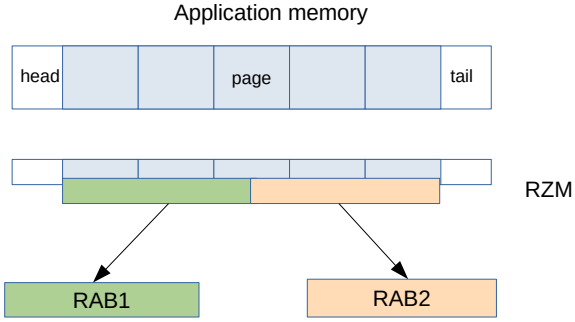


FIGURE 4.3 – *Association par le RZM*

Une optimisation que nous avons envisagée afin de minimiser le nombre de head dans une

application est l'utilisation d'un heap memory allocator qui privilégie l'allocation des buffers (réponse à `malloc()`) alignés. C'est le cas de plusieurs allocateurs tels que SlimGuard [LOR19]..

4.3 Détails d'implémentation

4.3.1 Interception des primitives systèmes

L'établissement d'une communication réseau, l'envoi et la réception des données sur le réseau se font à travers les primitives `write()`, `read()`, `send()`, `recv()` et `recv()` pour ne citer que ceux-là. L'invocation de ces appels systèmes est wrappée par la libc, ce qui permet leur interception de façon transparente (sans modifier les applications). Sous Linux, la variable d'environnement `LD_PRELOAD` permet de charger une librairie partagée avant l'exécution d'un binaire. Ainsi, tout symbole présent dans le binaire du programme à exécuter sera substitué par ceux présents et de même signature dans la bibliothèque partagée pointée par `LD_PRELOAD`. ODB utilise ce mécanisme pour s'interposer de façon transparente lors des appels `write()`, `read()`, etc. A l'ouverture d'une socket, ODB enregistre l'identifiant de la socket dans un annuaire afin de gérer son état (détaillé plus loin).

4.3.2 Implémentation de RAB

RAB est implémenté sous forme d'un agent qui gère un cache mémoire dans lequel sont sauvegardées les copies de données virtuellement envoyées. Nous utilisons un radix-tree pour fournir ce cache. Nous dédions une partie de la mémoire pour ce cache. Dans notre prototype actuel, nous le faisons en montant une mémoire persistante (pmem) émulée par la DRAM. Notons qu'une pmem réelle pourrait également être utilisée.

4.3.3 Implémentation de RZM

Il gère une table de descripteurs de données virtuelles et expose une interface de gestion de cette table avec les opérations de recherche, d'ajout, de modification et de suppression. RZM maintient également une table qui permet de mémoriser le statut et l'état des sockets.

4.3.4 Envoi de données

Au cœur de ODB se trouve un protocole qui consiste à encapsuler dans les données à envoyer sur le réseau leur statut réel. Une donnée peut être soit réelle (label R) ou virtuelle (label V), voir la Figure 4.4.

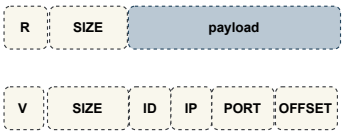


FIGURE 4.4 – *Encapsulation de la payload*

Un envoi réel correspond à l'envoi des données sur la connexion TCP, alors que l'envoi virtuel correspond à l'envoi d'une référence (odb_desc). Pour cela, le premier octet de la payload indique son type (R ou V). Les octets suivants dépendent du type de l'envoi. La tâche d'encapsulation est assurée lors de l'interception de `write()`. Les données de type V sont sauvegardées auprès du RAB et identifiées par un ID.

Envoi de données de type R

Toute payload dont la taille est inférieure à 4KB (taille d'une page mémoire) est réellement envoyée. Le champ type de notre protocole d'encapsulation est placé à R et les octets suivants représentent la taille de la payload, puis la payload proprement dite. Nous choisissons 4KB car c'est la taille par défaut d'une page mémoire, la granularité de protection de la mémoire. De ce fait, il n'est pas efficace d'envoyer virtuellement une payload dont la taille est inférieure à 4KB.

Envoi de données de type V

Toute payload dont la taille est supérieure à 4k (seuil configurable) est envoyée virtuellement. Dans ce cas, le champ indiquant le type d'envoi est placé à V. Les octets suivants indiquent (dans cet ordre) : la taille de la payload que l'application souhaitait envoyer (paramètre *count* de write()), ID identifiant cette payload dans RAB, IP et PORT de RAB, et offset localisant la payload dans le cache de RAB.

4.3.5 Lecture des données

Elle est réalisée par RZM. et chaque socket gérée par ODB est dans l'un des trois états suivants :

1. *virtual* : On a commencé à lire des données virtuelles sur la socket (provenant d'un envoi virtuel) et il reste des données virtuelles à lire pour cet envoi.
2. *real* : On a commencé à lire des données réelles sur la socket (provenant d'un envoi réel) et il reste des données réelles à lire pour cet envoi.
3. *none* : la socket n'est dans aucun des deux états précédents (pas de lecture en cours).

Son algorithme est guidé par l'état de la socket (*none*, *real*, ou *virtual*), voir l'automate d'état transition de la Figure 4.5.

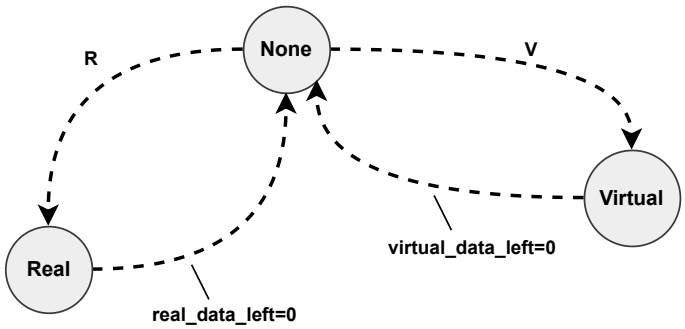


FIGURE 4.5 – *Diagramme d'état-transition d'une socket de lecture*

Dans cette dernière, *real_data_left* est la quantité de données réelles restantes à lire et *virtual_data_left* est la quantité de données virtuelles restant à lire.

L'appel à `read()` (intercepté) se comporte alors comme suit en fonction de l'état :

Etat *none*

Il s'agit de l'état initial de la socket. Dans cet état, RZM peut lire des données de n'importe quel type (R ou V). Si l'entête de la donnée lue est de type R, RZM positionne la socket dans l'état *real* et *real_data_left* prend la valeur de SIZE (contenue dans l'entête) de la donnée réelle du flux entrant. Si l'entête est de type V, ODB positionne la socket à l'état *virtual* et mémorise les informations méta-data envoyées (contenue dans l'entête) et *virtual_data_left* prend la valeur de

SIZE. Une fois dans un de ces deux états (*real* ou *virtual*), la lecture peut se poursuivre comme ci-dessous.

Etat *real*

Dans cet état RZM lit les données sur la connexion TCP. Soit *count* la quantité de données souhaitée par l'application (paramètre de *read()*). Si *count* est plus petite que *real_data_left*, alors *real_data_left* est décrémenté de *count* et cette quantité de données lue sur la socket et copiée dans le buffer applicatif. *count* est retournée à l'application et on reste dans l'état *real*, car il reste des données réelles à lire. Si *count* est plus grande que *real_data_left*, alors la quantité *real_data_left* est lue sur la socket et copiée dans le buffer applicatif. *real_data_left* est retournée à l'application et on passe dans l'état *none*.

Etat *virtual*

Une attention particulière est portée sur l'alignement du buffer de destination *buff* (paramètre de l'appel `read()`). Le lecteur peut se reporter à la section 4.2.2 ci-dessus afin de retrouver la stratégie pour l'alignement. Dans ce cas, RZM effectue la lecture des données (à distance auprès du RAB) qui ne seront pas sur des pages alignées ou qui ne feront pas la taille d'une page. Il s'agit de U1 et U2 dans la Figure 4.6.

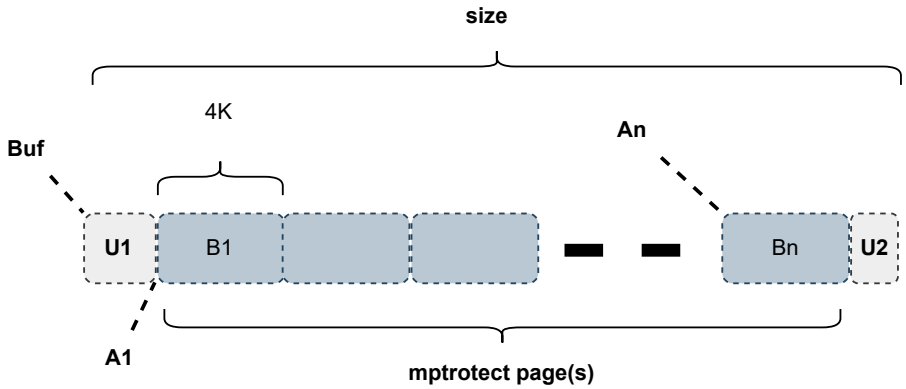


FIGURE 4.6 – *Protection des pages*

Il place les contenus U1 et U2 dans les endroits correspondants dans le buffer de destination *buff*. Puis la plage A1-An qui correspond aux pages alignés et de taille multiple de 4KB est protégée contre les lectures et les écritures. Comme précédemment (pour les lectures réelles), lorsqu'une donnée virtuelle est lue, RZM regarde la taille *virtual_data_left*. Si *count* est plus petite que *virtual_data_left*, alors *virtual_data_left* est décrémenté de *count* et cette quantité de données est traitée comme ci-dessus (U1 et U1 sont lues auprès du RAB et A1-An protégées). *count* est retournée à l'application et on reste dans l'état *virtual*, car il reste des données virtuelles à lire. Si *count* est plus grande que *virtual_data_left*, alors la quantité *virtual_data_left* traitée comme ci-dessus. *virtual_data_left* est retournée à l'application et on passe dans l'état *none*.

4.3.6 Protection mémoire

RZM utilise l'appel système `mprotect` et son flag *PROT_NONE* pour protéger les buffers applicatifs. Il place un gestionaire userspace pour les signaux *SIGSEGV*, ce qui lui permet de traiter les défaut de page. Lorsque l'application effectue un `read()`, nous déterminons à partir des paramètres *buff* et *size* les plages d'adresses mémoires à protéger. Comme présenté dans la

Figure 4.6, A1 désigne la première adresse de page alignée du buffer applicatif et An la dernière adresse de page alignée.

Une fois la protection effectuée, ODB constitue *odb_desc* (qui est composé de A1, ip, port, lenght, offset) associé aux blocs (B1,...Bn). *odb_desc* est ensuite enregistré auprès de RZM pour une utilisation lors d'une résolution de défaut de page. Lorsque ce dernier survient, un signal SIGSEGV est émis par le système d'exploitation et notre handler est appelé. Le handler récupère dans la variable context associée au gestionnaire, l'adresse (*si_addr*) fautive et détermine la page (Ai) contenant cette adresse. Ai déterminé, le handler lui restaure les droits de lecture/écriture. Ensuite, RZM est interrogé pour récupérer le *odb_desc* correspondant à Ai et les données correspondantes à Ai sont rapatriées depuis RAB via un appel RPC. Enfin, nous supprimons l'enregistrement correspondant à Ai dans RZM.

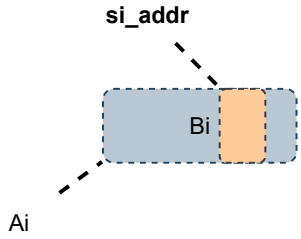


FIGURE 4.7 – *Faute de page*

Notons que la faute de page peut survenir à un emplacement quelconque comme on peut le voir sur la Figure 4.7. Si elle concerne les pages comprises entre B1 et Bn, la mise à jour auprès de RZM peut générer de nouveaux *odb_desc*, voir boîte verte sur Figure 4.8.

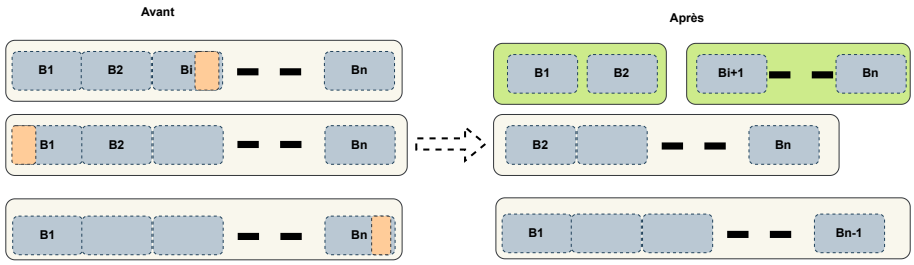


FIGURE 4.8 – *mise à jour RZM avant et après*

4.4 Evaluations

Cette section présente les résultats de l'évaluation de ODB. Notre évaluation porte sur les questions de recherche suivantes :

— **RQ1 - Avantages**. Quels sont les avantages de ODB par rapport à une architecture classique ?

— **RQ2 - Surcharge**. Quelle est la surcharge potentielle de ODB lorsque le serveur intermédiaire utilise la payload ?

Pour répondre à ces questions, nous utilisons les métriques suivantes : nombre de fautes de page, taux de données ayant contournées le serveur intermédiaire, impact de l'alignement des buffers applicatifs et charge CPU sur les serveurs applicatifs.

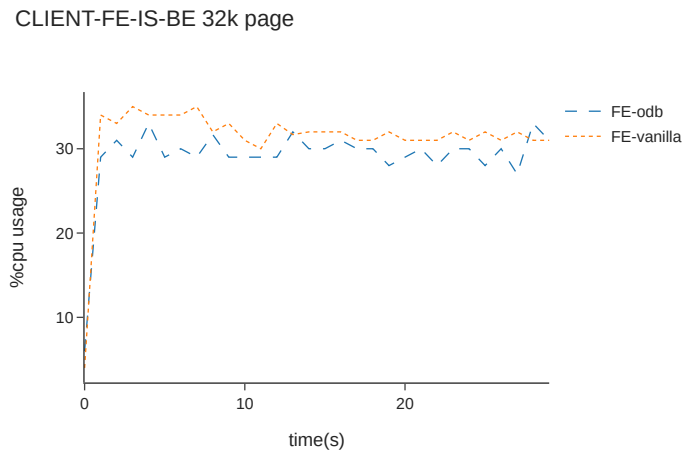
4.4.1 Méthodologie

L'environnement de test est le même que celui de DISC, voir Section 3.6.1. Dans ces expérimentations nous utilisons le microbenchmark Wrk et nous soumettons la charge suivante : 10 clients concurrents, chaque client émettant une requête de chargement d'une page web. Nous déployons une application web dans l'architecture multi-tiers composée d'un reverse proxy Nginx (FE), d'un loadbalancer Nginx (IS) et d'un backend Php (BE). L'application déployée est un script PHP qui revoie des pages dont les tailles varient de 1k à 64k. Chaque tier applicatif est pinné sur un seul CPU. Par défaut, l'allocation mémoire est alignée sur des frontières de page. Sauf indication contraire, nous utilisons cette configuration.

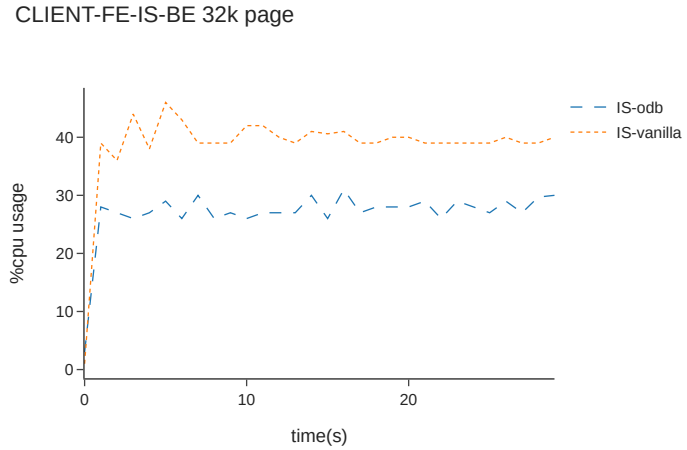
4.4.2 Evaluation de la charge CPU sur les serveurs

Taille de la payload fixée à 32k Ici, nous voulons observer l'impact de ODB sur la charge CPU des serveurs. Le microbenchmark est configuré de telle sorte que chaque client émet une requête pour une page de 32KB. La Figure 4.9 présente les résultats de la comparaison d'ODB par rapport à vanilla dans la configuration Client-FE-IS-BE.

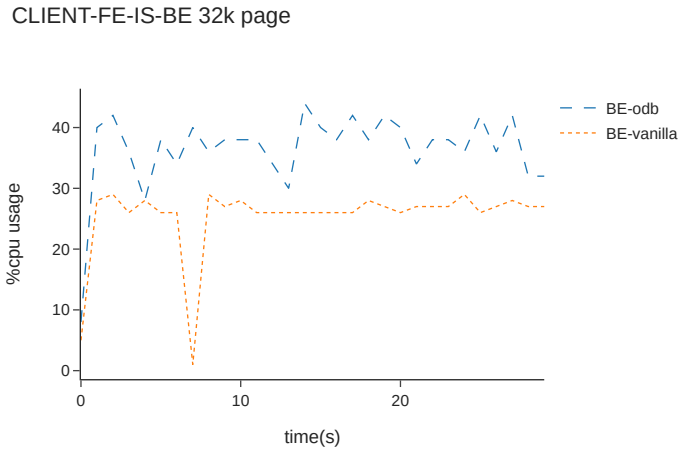
On peut observer sur la Figure 4.9a qu'il n'y a pas de baisse de charge significative sur FE dans la configuration ODB. FE étant le dernier maillon de la chaîne de traitement, RZM génère les requêtes pour récupérer les données virtuelles. Sur la Figure 4.9b, la charge sur IS pour la configuration ODB est significativement inférieure à celle de la configuration vanilla. Ces résultats s'expliquent par la dimunition des données en transit sur IS. Sur la Figure 4.9c, la charge de BE est plus élevée dans la configuration ODB par rapport à celle de vanilla. Cette augmentation s'explique par l'activité de RAB qui se produit lors de la résolution des fautes de page provoquées soit par IS, soit par FE.



(a) Charge CPU tier FE



(b) Charge CPU tier IS

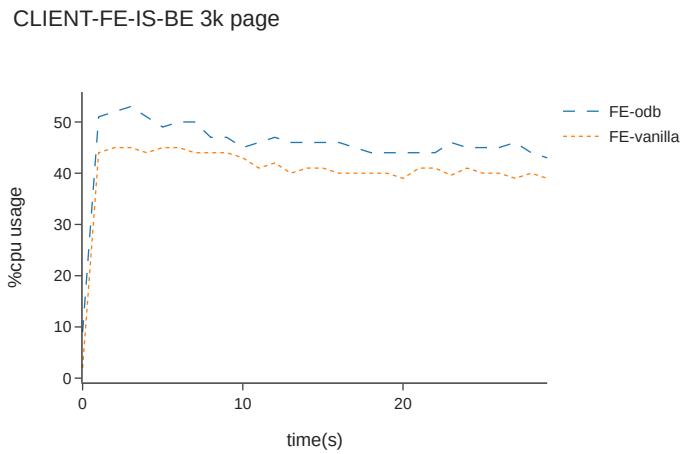


(c) Charge CPU tier BE

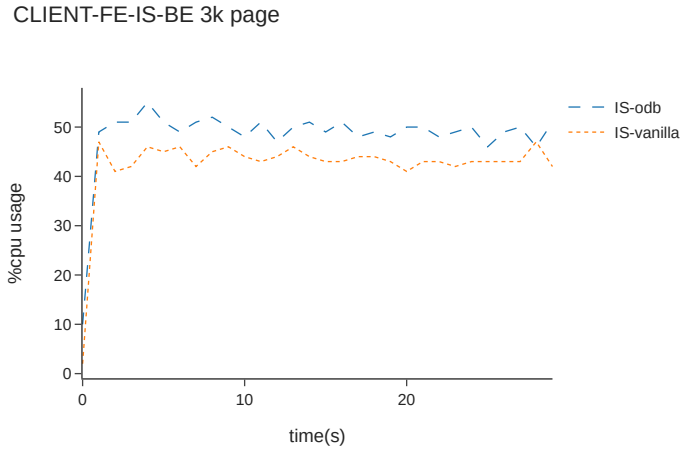
FIGURE 4.9 – *Charge CPU sur les tiers : configuration Client-FE-IS-BE, 32KB de données requêtées par chaque requête du client*

Taille de la payload fixée à 3k Ici, nous voulons voir l'impact de ODB sur la charge CPU des serveurs lorsque la taille de la payload est inférieure à 4KB, la taille d'une page memoire. Le microbenchmark est configuré de telle sorte que chaque client émet une requête pour une page de 3KB. Dans ce cas, toutes les données sont envoyées en rééL. Nous mesurons donc l'overhead de ODB en mode réel par rapport à vanilla qui est aussi en mode réel. Par comparaison, l'expérimentation avec une payload de 32k comparait ODB virtuel à vanilla réel.

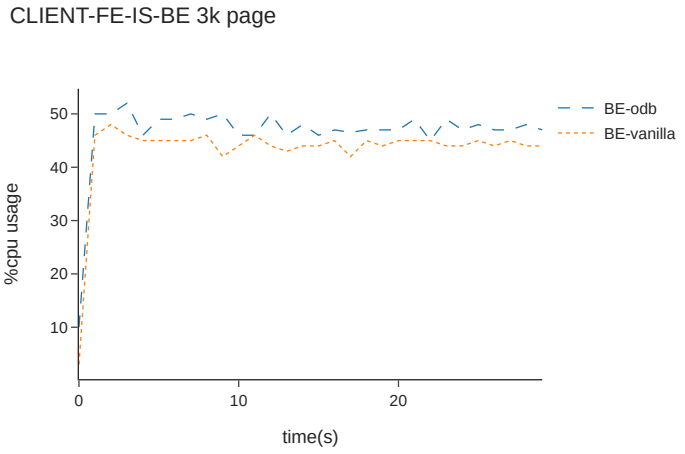
La Figure 4.10 présente les résultats. Nous pouvons observer que la charge CPU pour la configuration ODB est globalement (1.12 fois, 1.13 fois et 1.07 fois) plus élévée sur les figures 4.10a, 4.10b et 4.10c.



(a) Charge cpu tier FE



(b) Charge tier IS



(c) Charge cpu tier BE

FIGURE 4.10 – *Charger CPU sur les tiers : configuration Client-FE-IS-BE, 3KB de données requêtées par chaque requête du client*

Variation de la taille de la payload Ici, nous faisons varier la taille de la payload. Le but est de voir à partir de quelle taille de payload ODB est bénéfique (baisse significative de la charge CPU sur IS). La Figure 4.11 présente les résultats. On peut observer une baisse significative (51%) de la charge CPU sur le tier IS à partir de 16x4KB. Par contre, sur le tier BE, la charge dans la configuration ODB est élévée par rapport à vanilla. Ceci est dû à l'activité de RAB pour la gestion des pages mises en cache. L'implementation du RAB mériterait d'être optimisée.

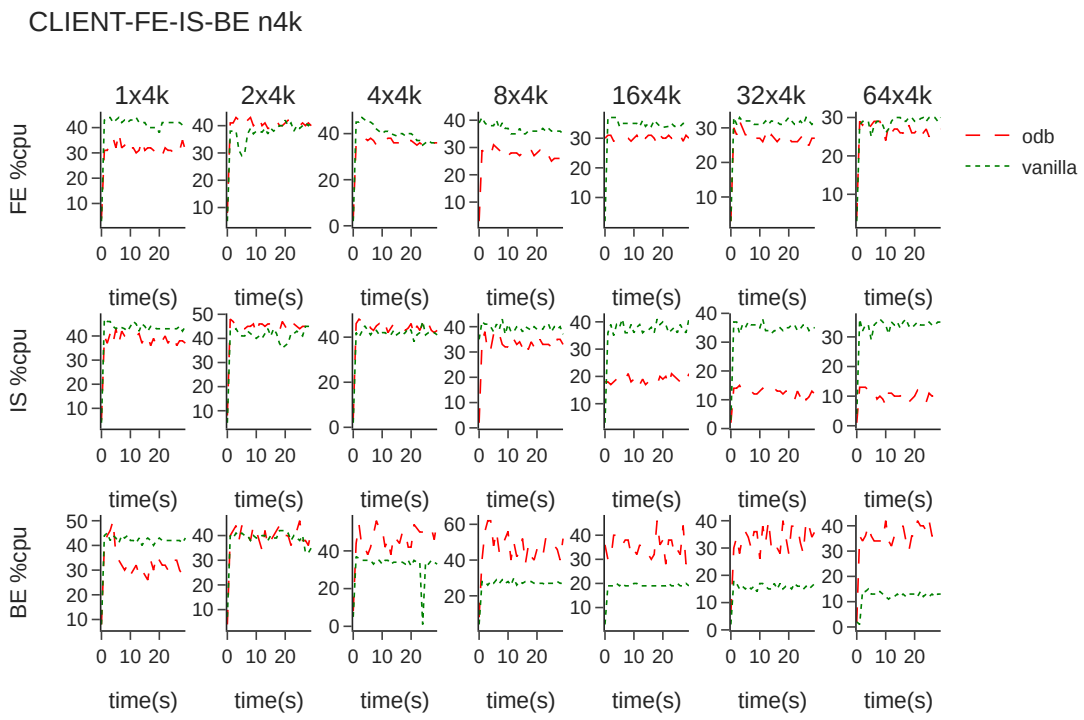


FIGURE 4.11 – *Utilisation du CPU sur les FE, IS et BE*

4.4.3 Alignement des buffers

Nous voulons voir l'impact de l'alignement de buffers sur la charge CPU des tiers. Ici, nous désactivons l'alignement buffers applicatif, la taille des payloads renvoyées par BE est de 32KB. La Figure 4.12 présente les résultats de l'évaluation d'ODB avec et sans alignement des buffers sur la frontière 4KB. Nous observons globalement une charge (0.93 fois, 1.14 fois et 2.21 fois) plus élévée lorsque les pages ne sont pas alignées. Cela s'explique par le fait que les tiers intermédiaires IS génèrent du traffic supplémentaire pour récupérer les zones head et tail.

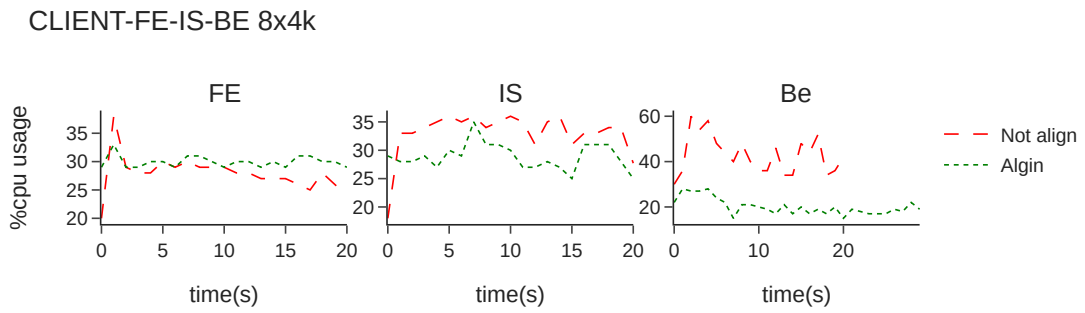


FIGURE 4.12 – *Impact de l'alignement des buffers sur l'utilisation CPU*

4.4.4 Le cas de Nginx

Ici, nous nous intéressons à analyser les payloads envoyées par notre serveur Php (BE), ainsi que les fautes de page générées sur le serveur intermédiaire Nging (IS). Des fautes de page sont générées lorsque IS parse les headers. L'expérience consiste à faire varier la taille des pages Php retournées. Nous injectons une seule requête et observons les payloads envoyées et fautes de page générées.

La Table 4.1 présente les quantités de données virtuelles ou réelles émises par le BE en fonction des tailles des payloads. Rlen désigne la taille des émissions de type R et Vlen la taille des émissions de type V. La colonne Fautes donne le nombre de fautes de page sur le IS. La colonne Php donne une interprétation de ce qu'à envoyé le moteur Php.

Au cours de cette expérimentation, nous avons découvert trois choses :

— le moteur Php est accédé à travers FastCGI qui ajoute ses propres headers (il a son propre protocole).

— FastCGI a une taille limite de trame retournée, ce qui fait qu'une grosse payload est découpée en plusieurs trames.

— à partir de 16KB, FastCGI passe en mode stream : il envoie une trame avec le header HTTP seul, puis la totalité de la payload et enfin une trame de fin (footer).

Ceci permet d'expliquer ce que nous observons dans la Table 4.1.

— pour une page (1x4k), FastCGI effectue une écriture d'une trame sur la socket (4184 bytes), la trame incluant le header de la trame, le header HTTP et la payload. Cette écriture est virtuelle et on a une faute de page lorsque IS parse les deux headers (HTTP et trame).

— pour deux pages (2x4k), FastCGI effectue deux écritures de trame sur la socket, car la payload excède la taille de trame (qui semble être de taille 8k). La première trame (8192 bytes) inclut le header de la trame, le header HTTP et le début de la payload. La deuxième trame (96 bytes) inclut le reste de la payload. La première écriture est virtuelle et la deuxième est réelle (car elle est petite). Il y a une faute de page lorsque IS parse les deux headers (HTTP et header de trame). La deuxième trame est envoyée en réel, donc pas de faute de page.

— pour trois pages (3x4k), FastCGI effectue deux écritures de trame sur la socket. Ces écritures sont virtuelles (8192 bytes et 4200 bytes) et il y a deux fautes de pages, la première lorsque IS parse les deux headers (HTTP et trame) de la première trame et la seconde lorsque IS parse le header de la seconde trame.

— pour quatre pages (4x4k), FastCGI passe en mode stream : il effectue une première écriture de trame incluant seulement le header HTTP (et le header de la trame) et cette écriture étant petite (88 bytes), elle est réelle. Il écrit ensuite toute la payload (16384 bytes) sans header

de trame en virtuel. Enfin, il écrit une trame de fin (16 bytes) en réel. Comme les headers sont envoyés en réel et la payload sans headers, il n'y a pas de faute de page.

— les envois plus gros se comportent comme pour le cas de quatres pages.

Cette analyse montre le comportement de Nginx avec FastCGI, que nous n'avons pas modifiés. Cela montre que ODB est particulièrement efficace pour les grosses payloads, puisque les headers sont envoyés en réel et la payload en virtuel sans header.

Taille	Fautes	Rlen	Vlen	FastCGI
1x4k	1	0	4184	1 write (V)
2x4k	1	96	8192	2 writes (V+R)
3x4k	2	0	8192, 4200	2 writes (V+V)
4x4k	0	88, 16	16384	3 writes (R+V+R)

TABLEAU 4.1 – *Impact des fautes de pages. Rlen désigne la taille des paquets de type R et Vlen la taille des paquets type V.*

4.4.5 Taux de données qui contourne le tier IS avec ou sans alignement des buffers

Ici, nous souhaitons observer le pourcentage de données qui contournent le tiers intermédiaire. Nous utilisons le microbenchmark décrit en Section 4.4.1. BE envoie des données de taille variable.

Buffer allligné La Table 4.2 présente les statistiques sur les mouvements de données avec alignement des buffers. On peut observer dans la Table 4.2 colonne Pourcentages, qu'une grande quantité de données ne sont pas exploitées par le tier IS et contournent donc IS, à partir 4x4KB. Avec les tailles inférieures 4KB, il n'y a pas de contournement possible.

Tailles	Reçues	Court circuit	Pourcentages
1x4k	1380352	0	
2x4k	2572288	1286144	50
4x4k	4079616	3059712	75
8x4k	5636096	4931584	87.5
16x4k	6760148	6332416	93.67
32x4k	9699328	9093120	93.75
64x4k	12582912	11796480	93.75

TABLEAU 4.2 – *Taux de court-circuit lorsque les buffers sont allignés.*

Buffer non-allligné La Table 4.3 présente les résultats. Nous faire les mêmes observations que précédemment. On peut donc conclure que statistiquement, le coutournement de IS, en terme de volume de données, est faiblement impacté par le non alignement des buffers. En fait, l'alignement a un impact sur la charge CPU étant donné que les parties head et tail doivent être rapatriées, mais l'incidence est faible sur le volume de données qui contournent le IS.

Tailes	Reçues	Court circuit	Pourcentages
1x4k	1504723	0	
2x4k	2646513	1331200	50.30
4x4k	3915776	2936832	75.00
8x4k	5767168	5046272	87.50
16x4k	7274496	6819840	93.75
32x4k	9175040	8601600	93.75
64x4k	10485760	9830400	93.75

TABLEAU 4.3 – *Taux de court-circuit lorsque les buffers sont non allignés.*

4.5 Conclusion

Dans ce chapitre nous avons présenté ODB comme approche permettant de mettre en place des raccourcis sans modification des applications. ODB propose une telle optimisation à travers la notion de pagination, en permettant à un serveur dans une architecture multi-tiers d'envoyer virtuellement les données aux tiers qui le précédent dans cette architecture. Un système de pagination est mis en place pour les réponses retournant une payload qui n'est pas exploitée (lue et modifiée) par les tiers précédents. L'implantation de ODB nécessite un préchargement d'une librairie fournie qui surcharge les primitives d'appels systèmes relatives aux entrées-sorties. Ceci permet de garder les applications intactes. Les évaluations conduites montrent les bénéfices qu'apportent ODB en terme d'utilisation CPU sur les serveurs court-cicuités.

5 Conclusions et perspectives

Nous résumons ici les contributions de cette thèse et dressons quelques perspectives à ces travaux.

5.1 Bilan

L'objectif de nos travaux était de réduire les communications dans les datacenters pour économiser des ressources. Nous avons fait l'hypothèse d'une structuration des applications suivant une architecture multi-tiers. Dans ces architectures d'application, les tiers interagissent suivant le modèle cient-serveur et s'appuient généralement sur le protocole TCP en utilisant des protocoles de plus haut niveau comme HTTP ou IMAP.

Nous avons observé que de nombreuses communications dans ces architectures sont inutiles dans le sens où elles véhiculent des données qui ne sont que relayées de tiers en tiers pour être renvoyées telles quelles au client externe. Ces données sont qualifiées de données finales. Pour ces données finales, nous proposons la mise en place de raccourcis qui visent à contourner les tiers intermédiaires qui n'exploitent pas les données finales.

Dans un premier prototype appelé DISC, nous introduisons un mécanisme de raccourci dans lequel nous distinguons dans les réponses retournées au client une partie header et une partie payload. Le header suit le chemin normal dans l'architecture multi-tier, alors que la payload finale est envoyée directement sur la connexion avec le client. Ce mécanisme permet de réduire significativement la charge CPU sur les tiers intermédiaires. Ce mécanisme nécessite une adaptation dans les systèmes permettant de distribuer la session TCP pour qu'elle soit utilisable par différents tiers, et une modification des applications pour prendre en compte le fait que la payload contourne les tiers intermédiaires.

Dans un second prototype appelé ODB, nous montrons comment on peut mettre en place des raccourcis sans modification des applications. ODB s'appuie sur la notion de buffer à la demande reposant sur les mécanismes de protection mémoire. La protection mémoire permet de détecter les accès aux buffers utilisés pour lire les données sur une socket et ainsi de ramener ces données à la demande, et ainsi de ne pas ramener ces données si elles ne sont pas utilisées. Les raccourcis peuvent ainsi être mis en place sans modification des applications. Ce mécanisme fonctionnant à la granularité des pages mémoires, il induit un léger surcoût.

5.2 Perspectives

Les perspectives à ces travaux peuvent se décliner en deux catégories : à court terme et à long terme.

5.2.1 Court terme

La première perspective de ses travaux concerne les performances du prototype DISC. Ses performances peuvent être significativement améliorées, principalement suivant deux axes :

— Meilleure intégration de TLS. Nous avons montré avec DISC qu'il était possible d'implanter des raccourcis y compris lorsque la connexion avec le client est chiffrée avec TLS. La session TLS peut être sérialisée et transportée entre les tiers émetteurs. Cependant, nous avons observé que le support de TLS avait un fort impact sur le tier émetteur en terme de charge CPU. Nous suspectons une mauvaise méthode de chiffrement dans notre implémentation où nous chiffrons paquets par paquets alors qu'il est possible de chiffrer des records TLS plus grands. Nous comptons reconsidérer nos choix d'implémentation du support de TLS pour obtenir de bien meilleurs résultats.

— Meilleure gestion des acks. Nous avons également observé qu'une charge CPU significative sur le tier émetteur est due à la gestion des acks. Dans le prototype actuel, tous les acks reçus sur le FE sont traduis en acks à destination du BE lorsqu'un raccourci est en cours. Nous comptons étudier la possibilité d'aggréger des acks, surtout des acks positifs, au pris d'une consommation mémoire légèrement supérieure sur le BE.

Le seconde perspective concerne l'intégration des deux prototypes DISC et ODB. Faute de temps, nous n'avons pu réaliser cette intégration. Elle ne pose pas de problème particulier et permettrait des raccourcis impliquant tous les tiers (y compris le FE) sans aucune modification des applications.

5.2.2 Long terme

A plus long terme, nous envisageons deux perspectives principales :

— Evaluation sur d'autres catégories d'application dans les datacenters. Dans cette thèse, nous nous sommes concentrés sur les applications multi-tiers déployées dans les datacenters. Ces applications sont contruites comme un assemblage de composants logiciels déployés sur des machines différentes. Ces composants logiciels (les tiers) interagissent suivant un modèle client-serveur. Nous avons utilisé un mécanisme de raccourci pour contourner des tiers lors du retour d'une réponse incluant une payload volumineuse. Nous pensons que ce même mécanisme peut être exploité pour d'autres types d'application, par exemple des applications de workflow où toutes les données véhiculées ne sont pas forcément exploitées par tous les composants du workflow.

— Utilisation de raccourci de facons plus générale. La notion de raccourci dans les commmunications peut potentiellement être utilisée dans des application en dehors des datacenters. Par exemple des applications déployées à plus large échelle sur Internet pourraient être étudiées. Nous avons envisagé l'utilisation de raccourci pour l'infrastructure de transfert des emails sur Internet, l'envoi d'un email passant par un serveur SMTP, puis par une chaîne de serveurs MTA pour arriver sur une serveur IMAP. Alors que logiquement, les emails doivent suivre une chemin logique en fonction du destinataire du message, des payloads comme les attachements aux emails pourraient faire l'objet de raccourcis pour soulager les serveurs intermédiaires (MTA).

Bibliographie

[AGH+15] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab Hamid, Muhammad Shiraz, Abdullah Yousafzai, and Feng Xia. A survey on virtual machine migration and server consolidation frameworks for cloud data centers. *Journal of network and computer applications*, 52 :11–25, 2015.

[AMMR92] R Ananthanarayanan, Sathis Menon, Ajay Mohindra, and Umakishore Ramachandran. Experiences in integrating distributed shared memory with virtual memory management. *ACM SIGOPS Operating Systems Review*, 26(3) :4–26, 1992.

[BCT03] Novella Bartolini, Emiliano Casalicchio, and Salvatore Tucci. A walk through content delivery networks. In *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 1–25. Springer, 2003.

[bpf06] BPFTOOL - tool for inspection and simple manipulation of eBPF programs and maps. https://man.archlinux.org/man/bpftool.8.en, 2006. Accessed : 2022-07-14.

[bpf07] High-level tracing language for Linux eBPF. https://github.com/iovisor/bpftrace, 2007. Accessed : 2022-07-14.

[CCD+05] Franck Cappello, Eddy Caron, Michel Dayde, Frédéric Desprez, Yvon Jégou, Pascale Primet, Emmanuel Jeannot, Stéphane Lanteri, Julien Leduc, Nouredine Melab, et al. Grid'5000 : A large scale and highly reconfigurable grid experimental testbed. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.*, pages 8–pp. IEEE, 2005.

[CFF14] Antonio Corradi, Mario Fanelli, and Luca Foschini. Vm consolidation : A real case based on openstack cloud. *Future Generation Computer Systems*, 32 :118–127, 2014.

[CGY01] Jeffrey S Chase, Andrew J Gallatin, and Kenneth G Yocum. End system optimizations for high-speed tcp. *IEEE Communications Magazine*, 39(4) :68–74, 2001.

[Clo53] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2) :406–424, 1953.

[Cor12] Jonathan Corbet. TCP connection repair. https://lwn.net/Articles/495304/, 2012. Accessed : 2022-07-14.

[dpd10] DPDK. https://www.dpdk.org/, 2010. Accessed : 2022-07-14.

[DRM+19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[Dru99] Peter Druschel. Tcp connection handoff. USENIX Association, 1999.

[DUB+11] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. {ETTM} : A scalable fault tolerant network manager. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[ES05] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 74–85. IEEE, 2005.

[FAP+14] Fahimeh Farahnakian, Adnan Ashraf, Tapio Pahikkala, Pasi Liljeberg, Juha Plosila, Ivan Porres, and Hannu Tenhunen. Using ant colony system to consolidate vms for green cloud computing. *IEEE Transactions on Services Computing*, 8(2) :187–198, 2014.

[GYLW20] Jingbo Gao, Wenbo Yin, Wai-Shing Luk, and Lingli Wang. Scalable multi-session tcp offload engine for latency-sensitive applications. In *2020 China Semiconductor Technology International Conference (CSTIC)*, pages 1–3. IEEE, 2020.

[hap11] Layer 4 Load Balancing direct server return mode. https://www.haproxy.com/fr/blog/layer-4-load-balancing-direct-server-return-mode/, 2011. Accessed : 2022-07-14.

[HHSE21] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. Prism : Proxies without the pain. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 535–549, 2021.

[iou19] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf, 2019. Accessed : 2022-07-14.

[KGWB08] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 123–134. IEEE, 2008.

[KIB20] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the load balancer without regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 193–207, 2020.

[KMS+09] Rupa Krishnan, Harsha V Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize cdn performance. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 190–201, 2009.

[KR06] Hyong-youb Kim and Scott Rixner. Tcp offload through connection handoff. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 279–290, 2006.

[KRR02] Jiantao Kong, Daniela Rosu, and Marcel Rosu. Towards enabling web proxy control of tcp splice transfer rates. In *2nd New York Metro Area Networking Workshop*. Citeseer, 2002.

[lin05] Linux and tcp offload engines. https ://lwn.net/Articles/148697/, 2005.

[LOR19] Beichen Liu, Pierre Olivier, and Binoy Ravindran. Slimguard : a secure and memory-efficient heap allocator. In *Proceedings of the 20th International Middleware Conference*, pages 1–13, 2019.

[MB99] David A Maltz and Pravin Bhagwat. Tcp splice for application layer proxy performance. *Journal of High Speed Networks*, 8(3) :225–240, 1999.

[MLJP20a] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. {AccelTCP} : Accelerating network applications with stateful {TCP} offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, 2020.

[MLJP20b] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP : Accelerating network applications with stateful TCP offloading. In

17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 77–92, Santa Clara, CA, February 2020. USENIX Association.

[Mog03] Jeffrey C Mogul. {TCP} offload is a dumb idea whose time has come. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[ngi] Accelerating networking with af_xdp. https://www.nginx.com/blog/ip-transparency-direct-server-return-nginx-plus-transparent-proxy/. Accessed : 2022-07-14.

[pac14] Packet_mmap. https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt, 2014.

[Pas12] Andrea Passarella. A survey on content-centric technologies for the current internet : Cdn and p2p solutions. *Computer Communications*, 35(1) :1–32, 2012.

[PBY+13] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta : Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4) :207–218, 2013.

[pfr08] High-speed packet capture, filtering and analysis. https://www.ntop.org/products/packet-capture/pf_ring/, 2008. Accessed : 2022-07-14.

[Riz12] Luigi Rizzo. netmap : a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[RW06] William Rice and H William. *Moodle*. Packt publishing Birmingham, 2006.

[S+21] Rajath Shashidhara et al. *TASNIC : a flexible TCP offload with programmable SmartNICs*. PhD thesis, 2021.

[SH02] Heinz Stockinger and Andrew Hanushevsky. Http redirection for replica catalogue lookups in data grids. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 882–889, 2002.

[SH04] Sandhya Senapathi and Rich Hernandez. Tcp offload engines. *Network AND Communications magazine pp103-107*, 2004.

[SHHP00] Oliver Spatscheck, Jorgen S Hansen, John H Hartman, and Larry L Peterson. Optimizing tcp forwarder performance. *IEEE/ACM Transactions on networking*, 8(2) :146–157, 2000.

[Spe09] SPECweb. https://www.spec.org/web2009/, 2009. Accessed : 2022-07-14.

[SSKP22a] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. {FlexTOE} : Flexible {TCP} offload with {Fine-Grained} parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, 2022.

[SSKP22b] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. {FlexTOE} : Flexible {TCP} offload with {Fine-Grained} parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, 2022.

[Sta03] Dragan Stancevic. Zero copy i : user-mode perspective. *Linux Journal*, 2003(105) :3, 2003.

[TK95] Moti N Thadani and Yousef A Khalidi. *An efficient zero-copy I/O framework for UNIX*. Citeseer, 1995.

[tls06] The TLS alert protocol. https://www.gnutls.org/manual/html_node/The-TLS-Alert-Protocol.html, 2006. Accessed : 2022-07-14.

[Tow79] Don Towsley. The stutter go back-n arq protocol. *IEEE transactions on Communications*, 27(6) :869–875, 1979.

[uio06] The Userspace I/O HOWTO. https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html, 2006. Accessed : 2022-07-14.

[VCP+20] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with ebpf and xdp : Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1) :1–36, 2020.

[wE18] SSL wolfSSL Embedded. Tls library, 2018.

[wrk] wrk2 Http Load generator. https://github.com/giltene/wrk2. Accessed : 2022-07-14.

[YLZ+09] Hao Yin, Xuening Liu, Tongyu Zhan, Vyas Sekar, Feng Qiu, Chuang Lin, Hui Zhang, and Bo Li. Design and deployment of a hybrid cdn-p2p system for live video streaming : experiences with livesky. In *Proceedings of the 17th ACM international conference on Multimedia*, pages 25–34, 2009.

[zim02] Zimbra. https://www.zimbra.com/, 2002.

Table des figures

Liste des tableaux

Résumé

Pour des raisons de modularité et de scalabilité, la plupart des services en ligne déployés dans des datacenters reposent sur une architecture multi-tiers dans laquelle plusieurs composants logiciels sont déployés sur des machines différentes . Ces tiers interagissent à travers des communications réseaux, le plus souvent reposant sur le protocole TCP/IP. Un client externe au datacenter peut se connecter à un tier frontal et soumettre une requête, provoquant un traitement distribué entre les tiers et retournant au client une réponse incluant des données. Les communications nécessaires pour faire remonter ces données dans l'architecture multi-tiers sont inutiles et sources d'une charge de calcul significative dans le datacenter. L'objectif de cette thèse est de réduire ces communications inutiles en introduisant un mécanisme de raccourci. Lorsque dans l'architecture multi-tiers, un tier génère des données qui ne sont pas exploitées par les tiers qui le précédent avant leur retour au client, alors un raccourci permet à ce tier de retourner directement les données au client sans passer par les tiers intermédiaires. Deuxièmement, les raccourcis bousculent la cohérence interne des applications et il faut modifier les applications dans ce sens. Les solutions actuelles nécessitent soit une modification du protocole TCP, soit des modifications importantes des applications. Nous montrons comment il est possible de distribuer une session TCP comme évoqué précédemment sans modification du protocole TCP.

Mots-clés : gestion de ressources, entrées-sorties, optimisation, achitectures multi-tiers

Abstract

A client, external to the data center, can connect to a frontal tier (the entry point) and submit a request, triggering a distributed execution For modularity and scalability reasons, most online services deployed in datacenters rely on a multi-tier architecture where several software components are deployed on different machines (e.g. between tiers which returns to the client a response that includes data. Interactions between tiers follow the classical client-server model.

We observe that in such services, a significant amount of data returned to the client are emitted by tiers in this architecture, without any handling by other tiers that lie between the emitting tier and the frontal tier. The tiers which precede the emitting tier only forward such data until the frontal tier returns the data to the client. Therefore, network communications used to transfer such data through the multi-tier architecture until the frontal tier are needless and a significant source of CPU waste in the data center. The introduction of a shortcut mechanism is equivalent to making the TCP connection with the client remotely accessible from any tier in the architecture. Two problems make it hard to implement such shortcuts. First, emitting tiers have to be coordinated in order to enforce the TCP internal consistency . Second, shortcuts compromise the internal consistency of applications which have to be modified accordingly. Current solutions require either a modification of the TCP protocol or important modifications in applications.

We show that modifications to applications can be minimal and that it is even possible to implement shortcuts without any modification to applications. Finally, we conducted a performance evaluation that shows that shortcuts allow a significant reduction of the CPU load in the data center, without degrading communication performance in terms of latency and throughput.

Keywords : resource management, input-output, optimization, multi-tiers architectures