



Étude et développement d'une solution de filtrage anti-DDoS sur DPDK

MÉMOIRE DE STAGE

PRÉSENTÉ LE 1 OCTOBRE 2015

PAR

Romain Ly

DIRECTEURS DE STAGE :

Nicolas VIVET, Pierre LORINQUER et François CONTAT

SGDSN / ANSSI / SDE / ST / LRP

Table des matières

Table des matières	iv
Summary	vi
Préambule	ix
1. L'Agence Nationale de la Sécurité des Systèmes d'Informations	ix
2. Sujet du stage	xii
2.1. Temps alloué au sujet	xiii
i. Introduction	1
1. DDoS	2
1.1. DDoS et Internet	2
1.2. DDoS et mécanismes	3
1.3. DDoS et protection	6
1.4. Problématique	9
2. Solution de protection logicielle	10
2.1. Budget de 200 cycles	10
2.2. NAPI	11
2.3. Goulets d'étranglement	12
2.4. Améliorations nécessaires	14
2.5. En utilisant la pile Linux	14
2.6. Contourner la pile réseau Linux	15
3. DPDK	16
3.1. Gestion de l'affinité et du <i>mapping</i> (EAL LIBRARY)	17
3.2. Gestion des files d'attente (RING LIBRARY)	18
3.3. Gestion de la mémoire (MEMPOOL LIBRARY)	19
3.4. Gestion des tampons mémoires (MBUF LIBRARY)	19
3.5. Gestion du traitement des paquets (POLL MODE DRIVER)	20
3.6. Améliorations apportées par DPDK	20
ii. Pont filtrant	21
1. Approche générale du développement	21
2. Initialisation de l'application	22
2.1. Initialisation des processus	23
2.2. Configuration des processus	24
2.3. Communications inter processus	25
2.4. Statistiques	25
2.5. Système de récupération	26
3. Chemin des données	28
3.1. Réception des paquets	28
3.2. Traitement des paquets et transmission	29
3.3. Utilisation du jeu d'instructions SSE	29

4.	Algorithmes de filtrage	31
4.1.	Redirection de port	31
4.2.	Modulo	31
4.3.	Déchargement vers la carte réseau	31
4.4.	BPF	32
4.5.	Algorithmes de recherche de correspondances	34
4.6.	Algorithmes de recherche de chaînes de caractères	35
5.	Résultats	36
5.1.	Redirection de port	36
5.2.	Algorithmes	38
6.	sécurité	40
iii. Conclusion		41
Bibliographie		47
Liste des acronymes		47
Annexes		53

Table des figures

1.	Organigramme de l'ANSSI	ix
i.1.	Attaque DDoS par réflexion et amplification	4
i.4.	Système NAPI dans le noyau Linux	12
i.5.	Files d'attente dans la pile réseau du noyau Linux.	13
i.6.	DPDK : Gestion de la mémoire	17
i.7.	DPDK : Fonctionnement d'un anneau de tampons en lecture .	19
i.8.	DPDK : Structure d'une <i>mempool</i>	20
ii.2.	Architecture du <i>bridge</i> : initialisation des processsus	23
ii.4.	Architecture du <i>bridge</i> : système de récupération	27
ii.5.	Architecture du <i>bridge</i> : Traitement des paquets	30
ii.6.	BPF : dans linux	33
ii.7.	BPF : utilisation dans mon application	34
ii.8.	Performances : <i>testbed</i>	36
ii.9.	Performances : redirection de port	37
ii.11.	Performances : redirection de port, configuration avec perte de paquets	38

Summary

Distributed denial-of-service (DDoS) attacks disrupt Internet services by exhausting victims network bandwidth, or machine resources. DDoS attacks could be launched by leveraging machines and services available through Internet. Compromised hosts (botnets) are used to launch attacks. Misused or unprotected servers can be abused to amplify DDoS attacks. Nowadays, DDoS attacks are a major threat to the Internet and DDoS mitigation systems are becoming more common.

DDoS protection uses a diversity of means to mitigate attacks. Specialized DDoS mitigation technologies use a combination of detection and filtering mechanisms, to remove DDoS traffic from legit traffic, with hardware devices tailored to achieve a high-grade performance. Nonetheless, these solutions are expensive and lack flexibility or extensibility. As an alternative, one could leverage commodity hardware to build flexible and scalable DDoS-mitigation systems.

Current high-speed network drivers use New API (NAPI) approach to improve packet processing. However, Operating Systems (OS) NAPI mechanisms are insufficient to process packets with a rate of the same order of magnitude as most DDoS attacks rate. Several architectural problems hinder high-speed packet processing. Packets are copied multiple times through the kernel and userspace. Packets reception consumes a huge amount of CPU resources. Traffic are serialized through a unique queue wasting parallelization possibilities. To overcome these limitations, different libraries bypass the network stack to bring performance.

The objectives of the study is to examine the possibilities of one of these libraries (DPDK, Data Plane Development Kit) to build a proof of concept of a DDoS mitigation system.

With DPDK, I completed to build an application that follows these properties :

- Modularity : the application could use different algorithms to filter packet based on header or payload
- Availability : the architecture is based on multiple processus with a recovery mechanism
- Parallelism : multiple queues are used to distribute load on different cores
- Performance : forwarding or filtering are done at near line rate with 64 bytes packets size (14 millions packets per second)

The only constructive theory connecting
neuroscience and psychology will arise from the
study of software.

Alan J. Perlis

1. L'Agence Nationale de la Sécurité des Systèmes d'Informations

L'Agence nationale de la sécurité des systèmes d'information (ANSSI) est rattachée au secrétariat général de la défense et de la sécurité nationale (SGDSN). Celui-ci assiste le Premier ministre dans l'exercice de ses responsabilités en matière de défense et de sécurité nationale. L'ANSSI est l'autorité nationale en matière de sécurité et de défense des systèmes d'information. Elle a pour principales missions d'assurer la sécurité des systèmes d'information de l'État et de veiller à celle des Opérateurs nationaux d'Importance Vitale (OIV), de coordonner les actions de défense des systèmes d'information, de concevoir et déployer les réseaux sécurisés répondant aux besoins des plus hautes autorités de l'État et aux besoins interministériels, et de créer les conditions d'un environnement de confiance et de sécurité propice au développement de la société de l'information en France et en Europe ¹. L'ANSSI est actuellement dirigée par Guillaume Poupard et est organisée en quatre sous-directions.

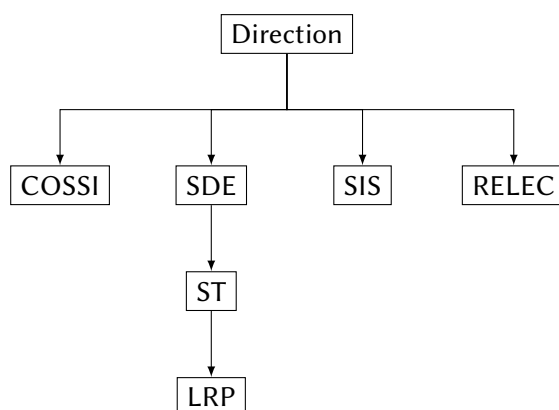


FIGURE 1. – *Organigramme de l'ANSSI.*

COSSI : Centre Opérationnel de la Sécurité et des Systèmes d'Information. Il est en charge de la défense des systèmes d'information, son action s'exerçant en priorité au profit des administrations de l'État et des OIV.

SDE : Sous-Direction Expertise. Elle apporte son soutien technique aux autres sous-directions de l'agence, aux ministères, aux industriels et prestataires de sécurité et aux OIV.

1. source : ssi.gouv.fr

SIS : Systèmes d'Information Sécurisés. Elle propose, conçoit et met en œuvre des produits et des systèmes d'informations sécurisés pour les ministères, les OIV et l'ANSSI.

RELEC : Relations Extérieures et Coordination. Elle est chargée des relations extérieures de l'agence, de la coordination des interventions et de l'élaboration de la réglementation.

Au sein de la SDE, la division Scientifique et Technique (ST) a pour rôle de définir les référentiels techniques de l'agence et d'apporter son soutien technique aux autres structures de l'ANSSI. À cette fin, elle doit connaître et maîtriser l'état de l'art de la sécurité des technologies et des systèmes d'information, anticiper les évolutions technologiques et proposer les innovations nécessaires.

Le laboratoire Sécurité des Réseaux et des Protocoles (LRP) constitue le pôle d'expertise de l'agence dans le domaine des réseaux (notamment l'internet, les réseaux de télécommunication fixes et les réseaux industriels). Il participe notamment, dans ces domaines, à la recherche, à l'analyse des besoins et à la conception des solutions propres à les satisfaire, à l'évaluation des produits et à l'élaboration et à la mise à jour de référentiels techniques.

Par exemple, dans le cadre de l'observatoire de la résilience de l'Internet français, l'ANSSI et l'Association Française pour le Nommage Internet en Coopération (AFNIC) publie annuellement un rapport détaillé sur la résilience de l'Internet en France en se basant sur les protocoles BGP (*Border Gateway Protocol*) ou DNS (*Domain Name System*).


Le laboratoire conseille aussi les entreprises et les structures gouvernementales. Les différents guides publiés sont disponibles sur le site internet de l'ANSSI².

Durant le stage, j'ai eu à disposition le matériel nécessaire pour l'étude et la réalisation des objectifs : ordinateurs, cartes réseaux, modules SFP. J'ai pu avoir accès à la documentation (articles et livres), et aux services communs de documentation de l'ANSSI pour l'impression de livrets. J'ai pu obtenir, pendant le stage, les éléments matériels supplémentaires pour compléter l'étude.

Le stage a été encadré par Nicolas Vivet, Pierre Lorinquer et François Contat. La discussion avec les tuteurs de stage et le directeur du laboratoire Guillaume Valladon, a été facilitée par la proximité, et leur disponibilité.

2. <http://www.ssi.gouv.fr/agence/rayonnement-scientifique/lobservatoire-de-la-resilience-de-linternet-francais/>

2. Sujet du stage

	Agence nationale de la sécurité des systèmes d'information	Fiche de stage SDE011
Etude et développement d'une solution de filtrage anti-DDoS		
<p>Description :</p> <p>Les attaques par déni de service distribué ou <i>DDoS</i> (<i>Distributed Denial of Service</i>) sont aujourd'hui fréquentes, notamment du fait de leur mise en œuvre relativement simple, et de leur efficacité contre des cibles non préparées. Il est donc nécessaire d'anticiper cette menace en prenant des mesures techniques et appropriées. Parmi celles-ci, les sociétés et les administrations ont parfois recours à des équipements de filtrage dédiés à la lutte contre les attaques DDoS. Cependant, ces équipements présentent souvent l'inconvénient d'être onéreux en termes d'achat et de maintenance.</p> <p>L'augmentation récente de la capacité de traitement du matériel destiné au grand public, ainsi que le développement de bibliothèques telles que <i>Data Plane Development Kit</i> (DPDK), permettent d'envisager la réalisation de solution logicielle de filtrage anti-DDoS à moindre coût basée sur du matériel non spécialisé.</p>		
<p>Localisation : 31, quai de Grenelle – 75015 Paris</p>		<p>Durée : 6 à 9 mois</p>
<p>Compétences requises :</p> <ul style="list-style-type: none"> • Maîtrise d'un langage C, C++ ou rust • Compréhension des protocoles exploités lors d'attaques par déni de service distribué • Connaissances théoriques et pratiques des systèmes d'exploitation (Linux ou FreeBSD) • Utilisation d'outils de développement collaboratif (git) • Maîtrise de l'anglais technique 		
<p>Formation :</p> <ul style="list-style-type: none"> • Bac +4/5 - École d'ingénieur ou formation universitaire en informatique <p>Qualités requises :</p> <ul style="list-style-type: none"> • Rigueur • Autonomie • Ouverture d'esprit 		

2.1. Temps alloué au sujet

Le temps consacré au projet (5 mois) :

- étude bibliographique (1 mois);
- développement (3 mois);
- expériences (1 semaine);
- rapport, présentations (1 mois).

Une attaque par déni de service ou *Denial of Service* (DoS) est une attaque informatique ayant pour but d'empêcher l'utilisation d'un service par des utilisateurs légitimes. Une attaque DoS est effectuée en épuisant une ressource nécessaire au bon fonctionnement du service : capacités des liens du réseau, ressources processeurs, nombre de connexions TCP, etc. Un autre moyen consiste à exploiter un bogue dans une application ou sur une machine pour forcer le service à s'arrêter ou à redémarrer. Les attaques par déni de service distribuées ou *Distributed Denial of Service* (DDoS) consistent à démultiplier la puissance d'une attaque DoS par l'utilisation de plusieurs machines. Dans ce rapport, je me focaliserai essentiellement sur les attaques DDoS.

La protection des attaques par déni de service distribué ou *Distributed Denial of Service* (DDoS) utilise souvent des *Intelligent DDoS Mitigation Systems* (IDMS) qui bloquent spécifiquement les paquets de déni de service. Ce sont des systèmes matériels et logiciels qui sont coûteux à l'achat et à l'entretien. Pour réduire ce coût, il serait possible d'utiliser une solution intégralement logicielle pour filtrer les paquets. Cependant, cela nécessiterait de pouvoir filtrer les paquets au niveau des en-têtes ou du contenu à très haut débit et pour des tailles de paquets variables. Or, les systèmes d'exploitation actuels ne sont pas capables de gérer les paquets de petite taille pour un débit à 14 millions de paquets par seconde (pour un lien à 10 Gbit.s⁻¹).

Pour surmonter cette difficulté, des bibliothèques de pile réseau « rapides » ont vu le jour. Elles permettent de contourner et de résoudre les problèmes structurels des piles réseau des systèmes d'exploitation. L'objectif de mon stage est d'étudier une de ces bibliothèques pour déterminer la possibilité d'y développer un système de filtrage anti-DDoS.

1. Les attaques par déni de service distribuées

Durant ces dernières années, les attaques DDoS sont devenues de plus en plus fréquentes. La durée et le débit utilisés lors des attaques DDoS ont augmenté. À titre d'exemple, on peut observer des attaques avec des volumes de l'ordre de la centaine de gigabits par seconde correspondant à des débits de plusieurs dizaines de millions de paquets par seconde. Ces attaques peuvent durer de plusieurs heures à plusieurs jours [Akamai Technologies 2015, Arbor Networks 2015].

Les victimes des attaques DDoS sont variées. On peut citer l'industrie vidéoludique qui représente la cible privilégiée. Il y a aussi les entreprises offrant des logiciels en tant que service (*Software as a Service* ou SaaS), les structures d'hébergement informatique, les institutions financières, les sites de divertissement ou de commerce électronique, les gouvernements, etc. [ANSSI 2015, Akamai Technologies 2015]. Les objectifs des attaquants sont multiples. Ils peuvent rechercher la notoriété en attirant l'attention des médias sur la perturbation d'un service important (par exemple, le cas d'une plateforme de jeux vidéo), revendiquer des positions idéologiques ou tenter d'extorquer des fonds à des entreprises. Les attaques DDoS sont aussi parfois utilisées pour détourner l'attention d'une autre cyberattaque.

Dans tous les cas, la perturbation de la disponibilité d'un service entraîne des dommages en terme de réputation, de productivité (l'attaque va drainer une partie des ressources de l'entreprise pour endiguer la cyberattaque) et de revenus (indisponibilité d'un site de commerce électronique ou violation d'une garantie de service (*Service Level Agreement* ou SLA)). Selon les facteurs pris en compte, l'estimation de la perte de revenus, consécutive à l'attaque DDoS, peut se chiffrer en dizaine de milliers d'euros pour les petites entreprises¹ à plusieurs millions d'euros pour les grandes compagnies de commerce électronique [Cisco Systems 2004].

Pour se protéger d'une attaque DDoS, il est nécessaire de comprendre la diversité des mécanismes et des vecteurs sous-jacents pour pouvoir anticiper les réponses à ce type de cyberattaque.

1.1. Comment les attaques par déni de service sont-elles possibles ?

La possibilité d'une attaque DDoS est liée à l'architecture des réseaux informatiques. La construction de l'Internet s'est focalisée dans l'efficacité de la distribution des paquets de la source vers la destination créant ainsi des réseaux intermédiaires simples et optimisés dans ce but.

Ce paradigme *de bout en bout* a fait que la fonction de routage du protocole IP n'impose pas la vérification de l'adresse IP source. Une des conséquences est l'existence de réseaux qui n'effectuent pas ce contrôle et qui peuvent donc être exploités pour usurper les adresses (IP *spoofing*).

De plus, la majeure partie de l'intelligence nécessaire à un service est située en bout de chaîne, ce qui laisse au réseau et à la source une quantité de travail relativement faible. Les ressources de ces entités terminales étant elles-mêmes limitées, elles peuvent être saturées. Ce traitement différentiel des paquets est renforcé par l'asymétrie entre la capacité des liens des réseaux de transit et celui des réseaux terminaux. Le réseau intermédiaire a une architecture

1. <https://twitter.com/kaspersky/status/582592074022838272>

permettant de soutenir de forts débits alors que le réseau final n'investit que les ressources nécessaires pour soutenir le service dans des conditions usuelles. Il existe donc un point de vulnérabilité dans les réseaux terminaux.

Finalement, la sécurité de l'Internet est interdépendante des entités qui l'habitent. Des systèmes comportant des vulnérabilités peuvent être compromis par des personnes malveillantes. Ces systèmes corrompus peuvent ensuite être utilisés contre un système hautement sécurisé. La gestion de l'Internet est distribuée ; or, la politique de sécurité est locale. Il est donc difficile de déployer des mécanismes sécurisés de manière globale.

1.2. Comment les attaques par déni de service sont-elles effectuées ?

Il existe une classification des attaques DoS ou DDoS où les taxons sont constitués selon le degré d'automatisme, les propriétés de la source et de la cible, l'impact généré sur la victime ... [Mirkovic et Reiher 2004]. Pour simplifier cette classification, nous proposons de considérer que deux classes d'attaques.

La première catégorie d'attaques vise à saturer les ressources réseau, les capacités des liens de la cible ou de son hébergeur en amont ou les ressources des éléments du réseau (pare-feu, routeurs, ...).

La seconde catégorie va cibler les ressources requises pour le fonctionnement d'un service, c'est-à-dire les serveurs *web*, d'authentification, etc.

1.2.1. Attaques visant à épuiser les ressources réseau

L'attaque vise à saturer les liens des réseaux avec des paquets *poubelles* pour y noyer le trafic légitime. On parle souvent d'attaque volumétrique. La conséquence de l'attaque est une perte globale de paquets et *in fine* une forte baisse de la probabilité d'acheminement d'un paquet légitime jusqu'à son destinataire. Cette catégorie d'attaque utilise souvent la réflexion et l'amplification pour créer un effet de levier et démultiplier la puissance de l'attaque.

La réflexion utilise les machines disponibles dans l'Internet (appelées *réflecteurs*) pour générer du trafic non désiré vers la cible. L'attaquant va forger des requêtes à destination des réflecteurs en usurpant l'adresse IP de la cible. Les réflecteurs vont répondre à la requête et vont envoyer le paquet contenant la réponse vers l'adresse IP usurpée, c'est-à-dire la cible. Le protocole UDP est fréquemment utilisé pour mener cette étape de l'attaque.

L'amplification consiste à utiliser des protocoles qui génèrent une réponse de taille plus importante que celle de la requête. Par exemple, pour une requête DNS, la taille de la réponse peut être plus de cinquante fois plus importante que la taille de la requête.

La plupart des attaques volumétriques utilisent à la fois la réflexion et l'amplification. L'illustration i.1 représente schématiquement une attaque DDoS volumétrique par amplification et réflexion par l'intermédiaire du protocole DNS. L'attaquant va émettre une requête DNS à destination des résolveurs

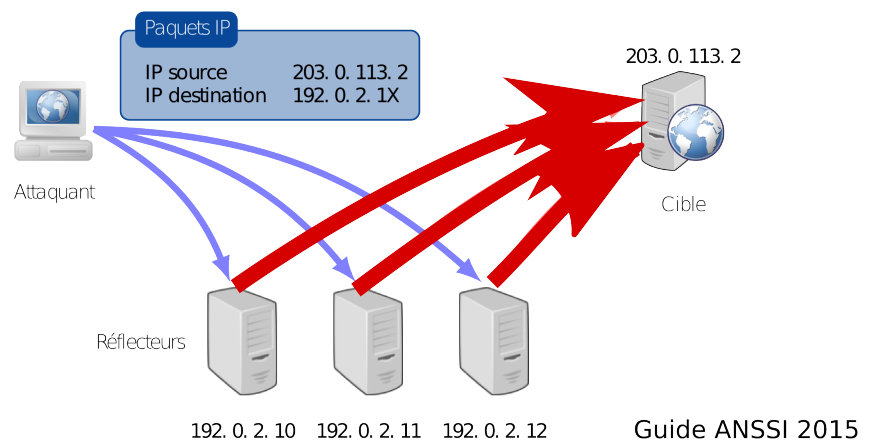


FIGURE i.1. – *Attaque DDoS par réflexion et amplification.*

DNS (réflecteurs) en utilisant l'adresse IP de la cible comme adresse IP source des paquets de la requête. Les résolveurs vont la résoudre en interrogeant des serveurs DNS faisant autorité, puis ils vont envoyer la réponse vers la cible. C'est l'étape de réflexion.

Le volume de ce type d'attaque est déterminé par plusieurs paramètres que l'attaquant va optimiser :

- l'accès à un *botnet*² ;
- le choix du protocole ;
- la requête pour engendrer la plus grande réponse ;
- le nombre de réflecteurs ;
- etc.

Il existe plusieurs exemples connus d'attaque DDoS par réflexion et amplification. On peut citer l'attaque contre Spamhaus qui a atteint un volume agrégé de 309 Gbit.s⁻¹, d'après la société Cloudflare [Prince 2013].

L'attaque par réflexion et amplification constitue une menace importante, car l'attaquant peut déclencher une attaque de grande envergure en utilisant les ressources disponibles dans l'Internet. Différents protocoles peuvent être exploités, ils sont choisis pour leur effet de levier (grand facteur d'amplification et grand nombre de serveurs exploitables sur Internet). Certains protocoles, comme DNS, sont nécessaires aux utilisateurs, il est donc impossible de bloquer le protocole. La table i.1 présente des protocoles qui sont utilisés dans les attaques DDoS [Akamai Technologies 2015].

1.2.2. Attaques visant les ressources machines

Ce type d'attaque vise à perturber ou arrêter le fonctionnement d'une ou plusieurs machines en exploitant un bogue ou en saturant certaines ressources critiques : le processeur ou *Central Processing Unit* (CPU), la mémoire, les tables d'états, etc.

2. réseau de machines compromises utilisables par l'attaquant, généralement à l'insu des utilisateurs légitimes, pour des activités malveillantes.

Protocoles	Facteur d'amplification ^a	Serveurs exploitables ^a
SNMP	11,3	4 832 000
NTP	4670	1 451 000
DNS _{SN}	98,3	255 000
DNS _{RO}	64,1	28 000 000 ^b
SSDP	75,9	3 704 000
Chargen	358	89 000

^a [Rossow 2014]

^b [Open Resolver Project 2015]

TABLE i.1. – **Protocoles couramment utilisés dans une attaque DDoS par réflexion.** Le facteur d'amplification correspond à la moyenne du premier décile des serveurs offrant l'amplification la plus forte. DNS_{RO} correspond à l'utilisation du protocole DNS avec résolveurs ouverts. DNS_{SN} correspond à l'utilisation du protocole DNS avec des serveurs de noms.

L'attaque la plus commune est l'attaque par inondation SYN qui vise les serveurs. Elle consiste à envoyer des paquets TCP SYN à la cible sans jamais conclure l'établissement de la connexion TCP (*Three-way handshake*). Le destinataire va répondre avec des paquets TCP SYN-ACK et va réserver des ressources (mémoires et CPU) pendant un certain temps. Ces ressources ne seront pas disponibles pour les utilisateurs légitimes. L'utilisation de la technique du SYN cookie permet notamment de se défendre contre ce type d'attaque.

Une autre attaque utilise la fragmentation des paquets IPv4. Un paquet peut transiter sur des réseaux dont la taille maximale de transmission ou *Maximum Transmission Unit* (MTU) des paquets peut être différente. Celui-ci peut donc être fragmenté. Le rassemblement des paquets est laissé à la charge du réseau de destination. Cette charge entraîne une allocation de ressources (mémoires et CPU) qui peut être exploitée par l'attaquant.

1.2.3. Les attaques applicatives

Ce type d'attaque vise à perturber le fonctionnement des applications. Par exemple,

- en saturant le nombre maximal de connexions concurrentes avec des requêtes HTTP avec une méthode de type GET ou POST ;
- en saturant les ressources CPU, avec par exemple une attaque sur le protocole TLS (*SSL exhaustion attack*) ou en effectuant des collisions dans les tables de hachage [Klink et Wälde 2011].
- exploiter un bogue pour perturber ou arrêter brutalement le service ;
- etc.

Dans la majorité des cas, il sera plus simple de forger un paquet frauduleux que de le traiter.

1.3. Comment se protéger d'une attaque DDoS ?

Les systèmes de protection utilisent plusieurs moyens pour endiguer les attaques DDoS. Dans un premier temps, il faut détecter l'existence d'une attaque DDoS et en trouver les origines. Les systèmes de protection font ensuite appel à l'ingénierie de trafic pour aspirer, bloquer ou faire de l'équilibrage de charge. Et finalement, la dernière étape consiste à filtrer les paquets pour en garder que le trafic légitime.

1.3.1. Détection du DDoS

La détection d'une attaque peut être difficile et constitue une thématique à part entière. C'est pourquoi elle ne le sera pas abordée de façon exhaustive. La difficulté dans la détection réside dans la capacité de pouvoir détecter le déni de service avant qu'il ne soit effectif.

Il existe plusieurs méthodes de détection [Prasad et coll., 2014]. La première méthode consiste à développer un modèle statistique du trafic habituel et de tester la significativité, au cours du temps, de chaque déviation du trafic réel par rapport au modèle initial. D'autres méthodes consistent à utiliser un système d'apprentissage automatique (*machine learning*).

Il est aussi possible d'utiliser la signature d'une attaque connue. Par exemple, le logiciel d'attaque DDoS appelé HOIC (*High Orbit Ion Cannon*) permet de mener une attaque applicative sur le protocole HTTP en envoyant des requêtes avec la méthode GET. L'analyse des paquets montre qu'il est possible de détecter une signature spécifique des requêtes provenant de cet outil. Par exemple, l'ordre des champs de l'en-tête HTTP peut être changé ou un caractère peut être répété. Dans la figure i.2, l'outil utilise deux espaces (caractère 0x20) après les deux-points dans le champ Keep-alive.

```

▼ Hypertext Transfer Protocol
  ▸ GET / HTTP/1.0\r\n
    Accept: */*\r\n
    Accept-Language: en\r\n
    Keep-Alive: 115\r\n
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
    Connection: keep-alive\r\n
    Referer: http://www.om.nl/vast_menu_blok/contact/\r\n
    User-Agent: Googlebot/2.1 ( http://www.googlebot.com/bot.html) \r\n
    If-Modified-Since: Wed, 30 Jan 2000 01:21:09 GMT\r\n
    Host: www.om.nl\r\n
    \r\n
    [Full request URI: http://www.om.nl/]
0050 2a 0d 0a 41 63 63 65 70 74 2d 4c 61 6e 67 75 61 *..Accept-Langua
0060 67 65 3a 20 65 6e 0d 0a 4b 65 65 70 2d 41 6c 69 ge: en.. Keep-Ali
0070 76 65 3a 20 20 31 31 35 0d 0a 41 63 63 65 70 74 ve: 115 ..Accept
0080 2d 43 68 61 72 73 65 74 3a 20 20 49 53 4f 2d 38 -Charset : ISO-8
0090 38 35 39 2d 31 2c 75 74 66 2d 38 3b 71 3d 30 2e 859-1,ut f-8;q=0.
00a0 37 2c 2a 3b 71 3d 30 2e 37 0d 0a 43 6f 6e 6e 65 7,*;q=0. 7..Conne

```

FIGURE i.2. – **Signature d'un paquet envoyé à partir du logiciel HOIC.** Source : [Barnett 2012]

1.3.2. Protection

La méthode naïve consiste à utiliser un système de protection en bout de chaîne, c'est-à-dire à la bordure du réseau de la cible. Ce type de solution permet de lutter contre des attaques dont le volume est inférieur à la capacité du lien réseau montant (*upstream*). Lorsque le volume de l'attaque dépasse cette capacité, la victime ne pourra pas intervenir, car il y aura une perte de paquets en amont.

La protection d'une attaque DDoS fait alors intervenir d'autres acteurs : l'hébergeur, l'opérateur ou un service de protection dédié. Voici une liste non exhaustive des solutions de protection utilisées.

Équilibrage de charge grâce à IP Anycast Cette méthode consiste à répartir le trafic sur plusieurs sites. Chaque site va recevoir un débit plus faible et pouvoir traiter localement l'attaque DDoS. Cette méthode utilise l'*anycast* qui permet d'annoncer une adresse IP à partir de plusieurs serveurs répartis géographiquement. Par exemple, Cloudflare a utilisé 23 sites différents pour contenir l'attaque contre Spamhaus [Prince 2013].

Proxy inverse DNS Cette méthode utilise un serveur de noms faisant autorité sur le domaine protégé. Les requêtes DNS des utilisateurs vont aller sur le réseau du *Content Delivery Network* (CDN) où les systèmes de protection vont pouvoir analyser et filtrer les paquets. Ce service est souvent offert par les CDN qui vont pouvoir équilibrer la charge sur plusieurs serveurs mandataires.

Black-holing Cette méthode permet de réduire efficacement le volume des attaques DDoS en contrepartie d'une limitation d'accès pour les utilisateurs légitimes. Cette solution de secours permet de limiter le trafic sans se soucier du caractère légitime ou illégitime.

La manipulation des routes BGP permet de mettre en œuvre cette solution, cette méthode s'appelle RTBH (*Remotely-Triggered Black Hole*) [Kumari et McPherson 2009]. L'idée est de manipuler les routes BGP pour bloquer les flux à destination de la cible en bordure du réseau du fournisseur d'accès.

La méthode *Destination-based RTBH* se base sur la destination. Tous les paquets à destination de la cible vont être jetés. Cette méthode rend efficace le déni de service, mais elle permet d'éviter les dommages collatéraux en désencombrant les liens qui peuvent être utilisés par les services des autres clients de l'opérateur/hébergeur.

Le *selective Black-holing* est une autre méthode qui permet de ne garder que le trafic provenant d'un territoire dont la portée est définie par la configuration. Cette portée peut être un pays, un continent, ou un cercle dont le rayon est de l'ordre du millier de kilomètres. L'idée sous-jacente est que la majorité des utilisateurs qui accèdent au service est localisée dans une même aire géographique que le serveur. On essaie donc de maximiser le nombre de personnes qui peuvent accéder au service tout en acceptant une partie du trafic illégitime [Snijder 2014, Contat 2015].

Flowspec Le protocole Flowspec permet de distribuer des listes de filtres aux routeurs en se basant sur les informations de la couche 3 et 4. Flowspec utilise BGP pour déployer les spécifications à tous les routeurs simultanément

[Serodi 2013]. Cette méthode n'est implémentée que dans un nombre restreint de routeurs (Alcatel Lucent, Juniper).

Systèmes de filtrage anti-DDoS intelligent ou Intelligent DDoS Mitigation System (IDMS) Les IDMS sont les solutions les plus utilisées pour contrôler une attaque DDoS. La méthode consiste à retirer le trafic illégitime sans toucher au trafic légitime.

L'objectif est de diriger le trafic d'une attaque vers le système de filtrage dédié. Le trafic y sera nettoyé et redirigé vers sa destination. La figure i.3 représente l'architecture de filtrage utilisé par OVH. Le trafic est aspiré dans le système, appelé VAC, pour y être filtré par divers équipements de filtrage haut débit.

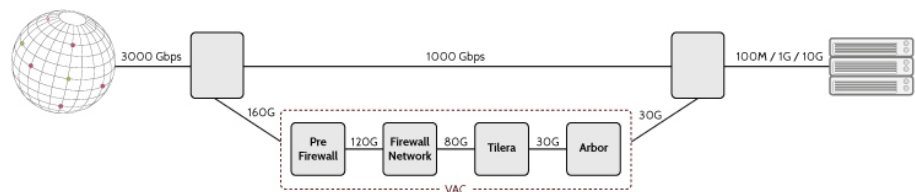


FIGURE i.3. – *Architecture de filtrage utilisé chez OVH. (Source : OVH).*

L'aspiration de trafic se fait en manipulant les routes BGP, ou en utilisant un tunnel dédié vers le système de filtrage.

Le filtrage utilise différents équipements qui par l'intermédiaire de listes de contrôle d'accès (ACL) vont autoriser ou bloquer des protocoles, des adresses IP ou des ports TCP ou UDP, etc. Ces équipements de filtrage se basent aussi sur le *checksum*, sur la fragmentation ou sur le contenu des paquets.

Cette méthode de filtrage requiert du matériel coûteux à acheter et à entretenir, mais les IDMS représentent la solution la plus utilisée pour l'accès au service aux utilisateurs légitimes tout en filtrant l'attaque DDoS.

1.4. Problématique

Pour réaliser une attaque DDoS, l'attaquant a besoin d'un ordinateur connecté à internet. Il y utilise l'infrastructure disponible (serveurs corrompus ou *botnets*) et un nombre suffisant de réflecteurs pour multiplier la puissance de l'attaque (par exemple en criblant les plages d'adresses IP pour trouver les résolveurs ouverts utilisables).

De l'autre côté, pour se défendre de l'attaque, la victime peut soit répartir la charge sur plusieurs centres de données (*data center*) disséminés dans le monde, soit utiliser un matériel coûteux dont la configuration peut être délicate pour éviter les dommages collatéraux sur les utilisateurs légitimes. Dans tous les cas, la défense d'une attaque DDoS est difficile, car elle fait appel à plusieurs métiers et demande la coordination de plusieurs services ou entreprises.

Une autre solution de protection envisageable serait d'utiliser des systèmes logiciels avec des produits de commodités tels que les processeurs x86 et les cartes Ethernet disponibles dans le commerce. Ces produits purement logiciels seraient beaucoup moins coûteux que du matériel ASIC. Ils seraient plus flexibles et extensibles, et permettraient d'exploiter la puissance des centres de données dans les fonctions de virtualisation.

Pour que cette solution soit efficace, le logiciel devrait traiter des volumes de paquets dépassant les 10 Gbit.s^{-1} avec des tailles de paquets variables. On peut donc se poser les questions suivantes : quelles sont les performances que l'on peut atteindre avec un système de traitement de paquet en logiciel ? Est-il possible d'y construire un système de filtrage anti-DDoS ? Est-il possible de passer à l'échelle ?

2. Solution de protection logicielle

En plus du coût, les systèmes de protection matériels manquent de flexibilité et d'extensibilité. Parmi les solutions de remplacement, les réseaux *Software-defined Networking* (SDN) permettent de gérer les fonctions des éléments réseau par l'intermédiaire d'une interface programmable et pourraient constituer un système de filtrage de remplacement. L'utilisation d'un circuit logique programmable ou *Field-Programmable Gate Array* (FPGA) dédié comme NetFPGA ou de matériel de commodité pourrait aussi constituer des systèmes de filtrage et de passer à l'échelle.

Les ordinateurs modernes partagent un ensemble d'instruction et une architecture souvent communs sur différents modèles, et permettent de produire un écosystème où le coût de production est compensé par le large volume d'unités fabriquées. Aujourd'hui, les processeurs possèdent plusieurs cœurs et sont construits avec une architecture (NUMA) permettant d'effectuer des traitements à haut débit. De plus, les cartes réseau disposent de fonctionnalités comme le *Receive-side Scaling* (RSS) qui permet de distribuer le trafic, selon des règles programmables, sur des cœurs différents, ou des fonctions de déchargement (*offloading*).

Cependant, le matériel a évolué plus rapidement que les systèmes d'exploitation qui ont mis la priorité sur la compatibilité plutôt que sur la performance. Le support réseau des systèmes d'exploitation actuel n'est pas optimal dans des environnements stringents tels que ceux à haut débit. Avec le noyau Linux de base, on atteint, sans optimisations, difficilement des débits supérieurs à plus de deux millions de paquets par seconde pour des paquets de 64 octets [Majkowski 2015, Emmerich et coll., 2015, Rizzo 2012].

On peut alors se poser plusieurs questions :

Quelles sont les limites du noyau Linux ? Quels sont les goulets d'étranglement présents dans la pile réseau du noyau ? Et comment les surmonter ?

Je me focaliserai ici sur le fonctionnement général de la pile réseau du noyau Linux car celui-ci, depuis la version 2.6, a incorporé un nouveau mécanisme appelé NAPI (*New API*) qui a inspiré les autres systèmes d'exploitation et qui constitue une des bases de la pile réseau actuelle.

2.1. Budget de 200 cycles

Avant de présenter le fonctionnement de la pile réseau, il faut noter que le traitement à haut débit de paquets nécessite un temps maximal de traitement par paquet qui est déterminé par la capacité du lien et la taille des paquets. Ce nombre constitue, en nombre de cycles CPU ou en temps absolu, le budget pour traiter un paquet.

Aujourd'hui, les cartes Ethernet peuvent atteindre 10 Gbit.s^{-1} avec des paquets de 64 octets (en comptant le contrôle de redondance cyclique (CRC) et sans l'espace inter trame, voir table i.2). Le débit maximal est donc de 14,88 millions de paquets par seconde (cf. table i.3). Ce débit correspond à un temps de traitement de 67,2 nanosecondes par paquet. Pour un CPU cadencé à 3 GHz, le budget par paquet est de 201 cycles³ environ. Ces 200 cycles ou

3. $3 \cdot 10^9 \times 67,2 \cdot 10^{-9}$

	octets
Espace inter trame	20
Préambule	8
En-tête Ethernet	14
Payload ^a	46
CRC ^b	4

^a Contenu minimal ;

^b Contrôle de redondance cyclique.

TABLE i.2. – *Détails d'une trame ethernet.*

67 nanosecondes constituent le budget maximal autorisé. Il est important que le système de filtrage puisse être capable de traiter aussi bien les petits paquets que les grands paquets, car l'attaquant est susceptible d'utiliser tous les moyens qu'il a à sa disposition.

budget (cycles)	Débit (Mpps)	taille (octets)
201	14,88	64
355	8,44	128
663	4,52	256
1282	2,34	512
2521	1,19	1024
3121	0,961	1280
3484	0,861	1518

TABLE i.3. – *Budget pour un lien à 10 Gbit.s⁻¹. Les tailles des paquets prennent en compte le CRC. Le budget en cycles est calculé pour un processeur cadencé à 3 GHz.*

2.2. NAPI

Dans les noyaux Linux dont la version est inférieure à 2.6, à l'arrivée de chaque paquet, la carte réseau transfère ce paquet dans une région de la mémoire réservée au *Direct Access Memory* (DMA). La fin de la copie du paquet dans cette zone mémoire entraîne une interruption matérielle ou *Interrupt Request* (IRQ). Le noyau va prendre en charge l'interruption et va gérer la copie du paquet du DMA dans un tampon (`sk_buff`) déjà alloué et situé dans une zone mémoire du noyau. Le paquet poursuivra son chemin dans le noyau pour être envoyé vers les applications.

Dans des conditions de fort débit, les interruptions incessantes et préemptives entraînent un interblocage actif (*livelock*). Le CPU est interrompu par l'IRQ du nouveau paquet, il arrête donc de traiter la tâche actuelle pour gérer le nouveau paquet. La gestion de l'IRQ coûte cher en terme de cycles CPU et décroît les performances globales. L'approche NAPI (*New API*, voir figure i.4) des noyaux Linux 2.6 et des versions supérieures permet de résoudre ce blocage par l'intermédiaire de deux principes : l'atténuation des interruptions et la limitation des paquets.

L'atténuation des interruptions. L'arrivée d'un nouveau paquet est gérée différemment par le système. Le premier paquet déclenche une IRQ. Cette

interruption active un mécanisme d'attente active (*polling*) qui vérifie la présence de paquets dans la file d'attente de la carte réseau, et se charge de les copier dans les zones mémoires du noyau. Ce mécanisme continue jusqu'à ce qu'il n'y ait plus de paquets dans la file d'attente. Les IRQ de la carte sont désactivées durant cette période. La consommation de CPU est plus élevée avec une charge réseau faible, cependant l'efficacité est plus grande lorsque la charge est élevée.

Limitation des paquets. Lorsqu'un flux de paquets surcharge les capacités du système, les paquets sont rejetés directement par la carte et non par le noyau. Cette méthode évite le surcoût lié à la copie du paquet le noyau et la gestion associée.

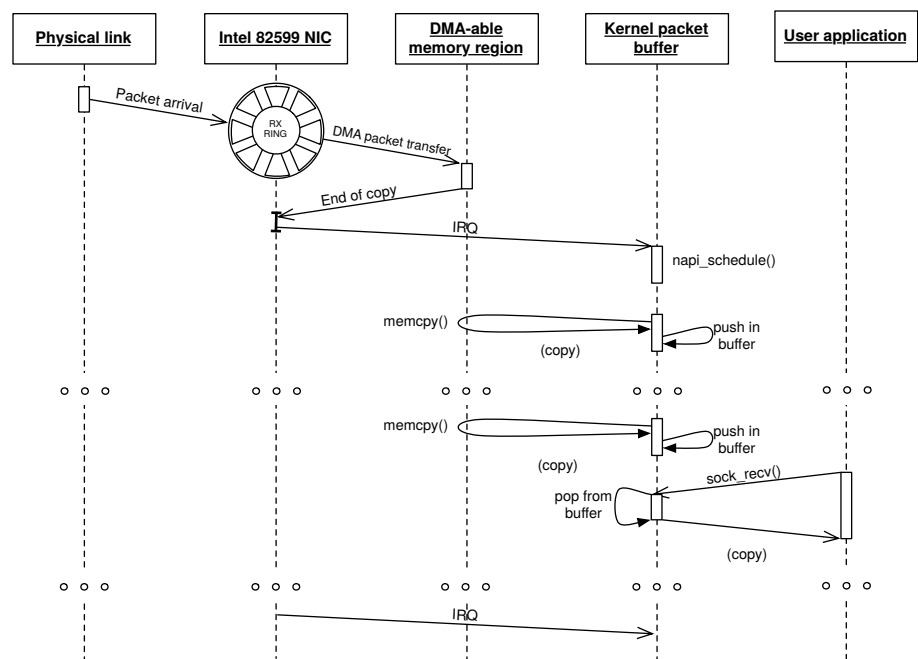


FIGURE i.4. – **Système NAPI dans le noyau Linux.** Tiré de [Garcia-Dorado et coll., 2013].

Le système NAPI permet d'accroître les performances de 20 % environ (Pentium II 350 MHz sur de l'Ethernet à 1 Gbit.s⁻¹ et des paquets de l'ordre de 1 Ko [Salim et coll., 2001]. Aujourd'hui, les goulets d'étranglement présents dans NAPI ne permettent pas d'atteindre les performances de 10 Gbit.s⁻¹ pour des paquets de 64 octets.

2.3. Goulets d'étranglement

Plusieurs études de mesures et de profilages ont mis en exergue les problèmes de la pile réseau [Garcia-Dorado et coll., 2013]. Seuls les goulets d'étranglement les plus importants sont décrits ci-dessous.

Allocation de ressources. Pour chaque paquet, un tampon est alloué puis désalloué dans le noyau. La table i.4 présente le coût en ressources CPU

pour chaque paquet. Cela représente plus de 50 % de la totalité du temps de traitement d'un paquet [Liao et coll., 2011, Han et coll., 2011]. De plus, pour des raisons de compatibilités protocolaires, la structure des métadonnées du `sk_buff`, qui contient un paquet, utilise plus de 208 octets en mémoire (soit 4 lignes de cache de 64 octets). La gestion de la mémoire n'est donc pas optimisée.

	cycles
Allocation	1200
Libération	1100

TABLE i.4. – *Coût de la gestion des ressources* [Liao et coll., 2011].

Multiple copies Chaque paquet est copié deux fois⁴ :

- Du DMA au noyau ;
- Du noyau en espace utilisateur.

Chaque copie nécessite environ 500 à 2000 cycles par paquet [Liao et coll., 2011, Han et coll., 2011]. Chaque paquet est copié individuellement, ce qui engendre un surcoût conséquent pour les petits paquets.

Sérialisation des files d'attente Les cartes réseau modernes peuvent recevoir et transmettre sur plusieurs files d'attente à la fois. À la réception, la carte réseau va distribuer les paquets sur des files d'attente qui sont liées à des CPU spécifiques.

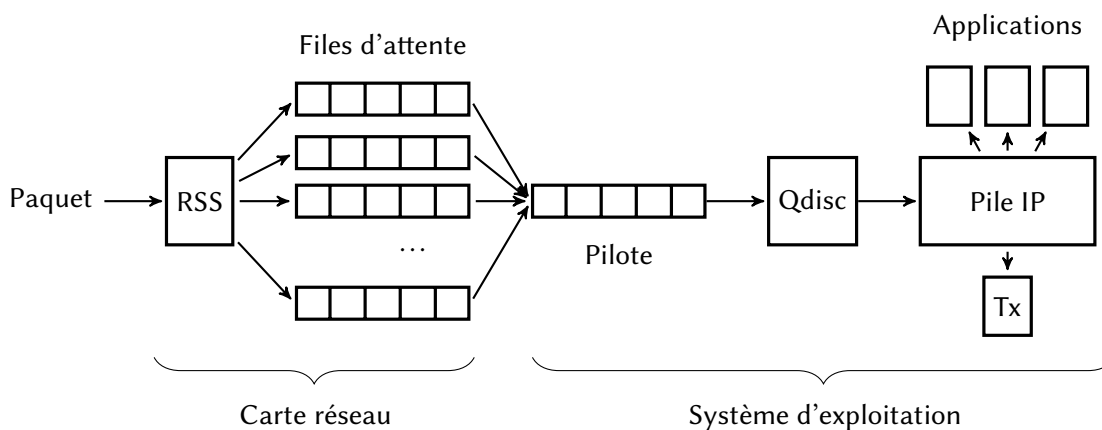


FIGURE i.5. – *Files d'attente dans la pile réseau du noyau Linux. En transmission, le chemin utilisé est similaire.*

Généralement, La carte réseau utilise une fonction de hachage sur plusieurs champs de l'en-tête (technologie *Receive-side Scaling* (RSS)). En fonction des empreintes calculées, les paquets vont être dirigés vers une file d'attente spécifique (dans le DMA, voir figure i.5) qui est assignée à une interruption dédiée permettant l'affinité entre un flux donné à un cœur donné. Sans ces technologies, tout le traitement des paquets va être effectué par le CPU qui a traité l'interruption.

4. La copie de la carte réseau dans le DMA n'est pas comptée.

Or dans le noyau Linux, les différentes files d'attente de la technologie RSS sont fusionnées en une seule file d'attente (voir figure i.5) ce qui annule l'avantage du RSS [Garcia-Dorado et coll., 2013].

Commutation de contexte. Pour une application en espace utilisateur, la commutation du noyau en espace utilisateur, et de l'espace utilisateur dans le noyau coûte environ 1000 cycles CPU [Liao et coll., 2011].

Mémoire Les accès mémoire constituent un des problèmes majeurs dans la gestion des paquets. Le premier accès au DMA entraîne un cache *miss* qui représente un surcoût en cycles CPU de 14 % [Han et coll., 2011]. De plus, la commutation de contexte entre applications sur le même CPU vide le cache de la *Translation Lookaside Buffer* (TLB) ce qui entraîne des caches *miss*. La TLB est la mémoire cache du processeur utilisée par l'unité de gestion mémoire (MMU) qui traduit les adresses de mémoires virtuelles en adresses physiques. Un cache *miss* correspond à un défaut d'entrée dans la TLB pour l'adresse virtuelle recherché, et induit un surcoût supplémentaire en cycles CPU. Selon le niveau de mémoire où se situe le cache *miss*, le surcoût est d'environ une dizaine à une centaine de cycles.

Multi-cœurs L'utilisation de plusieurs cœurs ne permet pas d'atteindre de meilleures performances, car le traitement des paquets est ralenti par des verrous tournants (*spinlocks*) dans le noyau. Pour accéder à la file de transmission de la carte, l'acquisition de deux verrous est nécessaire. Ce problème de synchronisation entraîne une non-linéarité entre le nombre de paquets traités à la seconde et le nombre de cœurs utilisés [Egi et coll., 2008].

2.4. Améliorations nécessaires

Pour surmonter ces difficultés, il faudrait effectuer plusieurs modifications :

- Pré allouer et réutiliser les tampons en mémoire ;
- Utiliser des files d'attente parallèles (pour exploiter le RSS) ;
- Accéder directement aux tampons dans le DMA ou dans le noyau (évite les copies) ;
- Traiter les paquets par lot (réduire le coût par paquet) ;
- Associer un processus à un CPU (affinité) pour éviter les problèmes liés à la gestion de la mémoire.

2.5. En utilisant la pile Linux

Dans le noyau Linux, il faut rajouter d'autres coûts. Le système de contrôle de trafic appelé Qdisc (*Queuing Disciplines*) engendre un surcoût de 70 ns par paquet (à cause des verrous). Le coût d'un appel système est d'environ 220 cycles.

Dans la configuration de base du noyau Linux, la fonction *bridge* permet d'obtenir un débit de 1,11 Mpps environ (millions de paquets par seconde). Différents moyens peuvent être utilisés pour optimiser le traitement des paquets dans la pile réseau. Parmi ceux-ci, on peut notamment citer :

	cycles
Appel système	220
qdisc ^a	100

^a contrôle de trafic dans Linux, règles vides.

TABLE I.5. – **Surcoût supplémentaire.** [Dangaard Brouer 2015].

- taskset ou cpuset qui permet d’assigner un processus à un proces-
seur ;
- ethtool qui permet d’activer et de configurer la technologie RSS de la
carte ;
- irqbalance qui permet d’associer l’IRQ de la file d’attente de paquets
avec le processeur qui va gérer les paquets de cette file d’attente ;
- packet mmap⁵, qui permet d’accéder directement aux tampons du
noyau, traiter par lot et limiter le nombre d’appels systèmes ;
- etc.

Jesper Dangaard Brouer⁶ réussit à atteindre les 14,8 Mpps en renvoyant toujours le même paquet avec l’outil pktgen avec un seul cœur. Le noyau a été modifié pour traiter les paquets par lot et pour réutiliser la même structure `sk_buff`, évitant ainsi les problèmes liés à l’allocation et la libération des tampons.

2.6. Contourner la pile réseau Linux

Pour éviter de rentrer dans l’optimisation de la pile réseau des systèmes d’exploitation, il est possible de les contourner et d’utiliser des bibliothèques qui vont permettre de traiter les paquets directement en espace utilisateur. Ces bibliothèques de programmation tiennent compte des difficultés actuelles de la pile réseau.

Ces bibliothèques sont destinées pour les applications nécessitant un traitement de paquets à haut débit : routeur, pare-feu, *Deep Packet Inspection*, système de détection d’intrusion ou *intrusion detection system* (IDS), surveillance (*monitoring*), virtualisation, ... et sont de plus en plus utilisées ou considérées aujourd’hui.

Dans notre cas, ces bibliothèques pourront être utiles pour développer un système de filtrage de paquet haut débit pour la protection contre les attaques DDoS.

5. http://lxr.free-electrons.com/source/Documentation/networking/packet_mmap.txt

6. [Dangaard Brouer 2015]

3. DPDK

Il existe plusieurs bibliothèques permettant de contourner la pile réseau du système d'exploitation :

- Intel DPDK
- Netmap
- PF-RING
- PacketShader (PSIO)
- PFQ
- etc.

La conception de ces bibliothèques s'articule sur les points névralgiques cités dans la section 2.4 (p. 14). Toutes les bibliothèques implémentent ces propriétés :

- Préallocation mémoire et réutilisation des tampons ;
- Plusieurs files d'attente parallèles ;
- Association mémoire en espace utilisateur et noyau ou DMA ;
- Affinité entre l'IRQ de la file d'attente et le CPU qui gère cette file ;
- Traitement par lot.

Malgré ces similitudes, il existe des différences entre ces bibliothèques :

PF_RING ⁷ est une solution commerciale qui offre la caractéristique du *zero copy*, qui permet de s'affranchir de toute copie inutile en permettant d'accéder directement à la zone mémoire du DMA. Cependant, cette méthode expose des vulnérabilités où l'application peut accéder à des zones mémoires incorrectes et faire arrêter brutalement le système. L'interface de programmation (API) de PF_RING offre des fonctions de filtrage avec BPF (voir 4.4 (p. 32)) [Garcia-Dorado et coll., 2013].

PacketShader utilise le *Graphic Processing Unit* (GPU) pour l'accélération du traitement des paquets [Han et coll., 2011].

DPDK ⁸ et Netmap partagent une architecture similaire : les applications en espace utilisateur accèdent directement aux tampons des paquets dans le noyau. Les métadonnées et les données du paquet sont situées dans le même tampon mémoire. DPDK expose le pilote en espace utilisateur alors que Netmap garde le pilote dans le noyau et expose seulement les tampons.

Les performances sont très similaires pour la plupart des bibliothèques, il est possible d'atteindre le débit en ligne (*line rate*) sur un seul cœur avec des paquets de 64 octets (14,88 Mpps). PF_RING, Netmap et DPDK semblent être les bibliothèques les plus utilisées dans les applications. Nous avons choisi de nous concentrer sur la bibliothèque DPDK.

L'API de DPDK propose plusieurs bibliothèques de fonctions qui sont optimisées pour le traitement de paquets. Les bibliothèques principales offrent les fonctions similaires à la pile réseau du noyau Linux avec les optimisations citées dans la section 2.4 (p. 14). Les principales bibliothèques concernent la mémoire, les files d'attente, la gestion des tampons, l'affinité mémoire et des IRQ. Seules les bibliothèques les plus essentielles à l'utilisation de DPDK seront décrites ci-dessous.

7. ZC pour Zero Copy, anciennement appelé PF_RING DNA.

8. anciennement appelé Intel DPDK

3.1. Gestion de l'affinité et du *mapping* (EAL LIBRARY)

La bibliothèque EAL (*Environment Abstraction Layer*) fournit aux applications un environnement qui va permettre de gérer les ressources dont voici une description des services les plus essentiels.

Initialisation et affinité à un cœur La bibliothèque EAL initialise les différents services offerts par l'API : fonctions de débogages, alarmes, verrous, fonctions atomiques, etc. Parmi les fonctions les plus importantes, il y a l'initialisation des cœurs. Chaque application peut utiliser plusieurs cœurs appelés par DPDK cœurs logiques (*logical core*). En effet, DPDK associe chaque fil d'exécution avec un cœur logique (avec la technologie de l'*hyper-threading* activé). Chacun de ces fils d'exécution sera épinglé sur un cœur logique spécifique, et y sera associé avec une cardinalité $<1 \dots 1>$. Cette structure évite les coûts liés à la commutation de contexte, et permet de garder le cache TLB du cœur cohérent.

Le processus va lancer la bibliothèque EAL qui initialise les différents services cités ci-dessus et va créer plusieurs fils d'exécution (avec au minimum un fil d'exécution). Ce même processus initialise aussi les ressources nécessaires à l'application (files d'attente, mémoires, carte réseau, ...). À la fin de l'initialisation, les différents fils d'exécution créés vont exécuter l'application.

Gestion de la mémoire DPDK utilise des segments de mémoire de 2 Mo ou 1 Go (*HugePages*) à la place des 4096 octets utilisé couramment. L'utilisation de pages de grande taille permet de diminuer le temps pour la recherche de correspondance entre l'adresse virtuelle et l'adresse physique dans la table TLB.

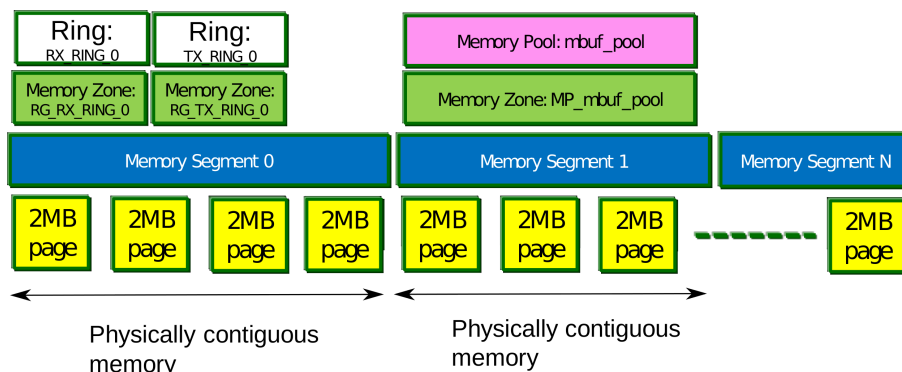


FIGURE i.6. – *Gestion de la mémoire par DPDK*. Source : [Intel 2012]

Chaque page va prendre une place dans la table TLB. Plus le nombre de pages d'un processus est élevé, plus la place prise dans la table TLB est grande. Dans ce cas, un accès mémoire aura une plus grande probabilité de tomber sur un cache *miss*, car une des pages associées au processus pourra ne pas être présente dans la table TLB. Plus on augmente la taille des pages, plus le nombre de pages associées au même processus diminue et plus la place prise par le processus dans la table TLB est faible. Concrètement, un processus avec un espace d'adressage de 2 Mo est composé d'une page avec les *Hugepages*

ou de 512 pages avec des pages de 4 Ko ce qui correspond à 1 entrée ou 512 entrées respectivement dans la table TLB.

L'utilisation des *Huge Pages* permet d'augmenter la *portée* de la TLB, c'est-à-dire la quantité de mémoire pouvant être traduite par la TLB et permet l'optimisation de la gestion de la mémoire par le CPU. L'EAL va allouer toute la mémoire nécessaire pour l'application dans les *HugePages* par l'intermédiaire d'un appel système à `mmap`.

La cartographie de la mémoire physique se fait par l'intermédiaire d'une table de descripteurs où chaque descripteur décrit une portion contiguë de la mémoire. Ces segments mémoires (*Memory segment*) sont constitués de pages provenant des *HugePages* et sont subdivisés en zones mémoires (*Memory zones*) qui constituent la base mémoire des objets créés par DPDK (voir illustration i.6). Les zones mémoires sont accessibles par d'autres applications DPDK, on peut donc créer un *mempool* pour la réception et la transmission de paquets et y greffer plusieurs applications.

3.2. Gestion des files d'attente (RING LIBRARY)

Les files d'attente sont conçues à partir d'un anneau de tampons (*ring buffer* [Rostedt 2009]) et suivent les propriétés suivantes :

- Taille fixe ;
- Sans verrous ;
- FIFO (*First in, First out*) ;
- Autorise plusieurs consommateurs/producteurs ;
- Structure en anneau, avec un pointeur de tête et de queue ;
- Traitement par lot.

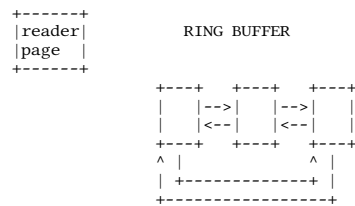
Par rapport à une liste doublement chaînée, l'intégralité de cette structure peut être préalloué dans la mémoire et les tampons de cette structure sont réutilisables. Par conséquent, les coûts des allocations et des libérations des tampons sont inexistantes. Les opérations de lecture et d'écriture se font simplement avec une fonction atomique de comparaison de drapeaux (*flags*) et d'échange de tampons.

Lorsqu'un lecteur demande la lecture d'une page, celle-ci est retirée de l'anneau de tampon si cette page est accessible en lecture. Lorsque le même lecteur demande une nouvelle page en lecture, il échange la page précédemment lue avec la nouvelle page désirée (cf. figure i.7).

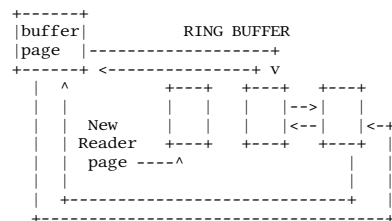
La page qui est en cours d'écriture, n'est pas accessible en lecture. Lorsqu'un écrivain désire écrire sur la page n , le pointeur de tête (écriture) va aller pointer sur la page suivante ($n+1$). Si un autre producteur désire écrire, il écrira à partir de la page ($n+1$).

La gestion peut être plus complexe dans d'autres conditions. Par exemple, lorsque les producteurs rattrapent les consommateurs, c'est-à-dire que le pointeur de tête rattrape le pointeur de queue. Dans ce dernier cas, les opérations d'écriture échouent (perte de paquets).

Cette gestion des tampons est plus rapide, car il y a besoin que d'une seule opération atomique : il est possible d'ajouter ou de retirer en une seule opération plusieurs pages dans l'anneau sans utiliser de verrous. Cette structure est utilisée dans la gestion des tampons des paquets et peut être réutilisée dans la gestion des communications inter-processus (IPC).



(a) À l'initialisation.



(b) Lecture d'une nouvelle page.

FIGURE i.7. – **Fonctionnement d'un anneau de tampons en lecture.** Source : [Rostedt 2009].

Cependant, l'utilisation de cette structure présente quelques désavantages : la taille de l'anneau est fixe et configurée à l'avance, ce qui peut être très coûteux en espace mémoire puisqu'elle n'est pas dynamiquement allouée et contrairement à l'algorithme RCU (*Read Copy Update*) présent dans le noyau Linux, il n'existe pas de preuve formelle de l'algorithme de fonctionnement de l'anneau de tampons de Stevend Rostedt [Rostedt 2009].

3.3. Gestion de la mémoire (MEMPOOL LIBRARY)

Parmi les objets contenus dans les segments mémoires, la *mempool* est la plus importante. Elle contient des objets de tailles fixes structurés en anneaux. L'allocateur de mémoire de la *mempool* maintient un cache exclusif à chaque cœur. Lorsque ce cache est vide ou rempli, le cœur va échanger des objets du cache spécifique vers la collection d'objets libres de la *mempool*.

La mémoire est structurée par du bourrage pour que chaque objet soit aligné avec le cache et que le début de chaque objet (paquet) commence sur un canal (*memory channel*) et un rang mémoire (*memory rank*) différent. Cette structure ainsi que l'entrelacement de la mémoire permettent d'accéder aux 64 premiers octets des paquets rapidement et de pouvoir classifier les flux à partir d'un seul accès mémoire.

3.4. Gestion des tampons mémoires (MBUF LIBRARY)

Contrairement à la pile Linux qui sépare les métadonnées et les données des paquets dans deux tampons séparés⁹, DPDK stocke l'intégralité dans le même tampon de taille fixe. L'avantage est de pouvoir allouer le tampon

9. `skb` et `skb->head` pour les métadonnées et les données respectivement.

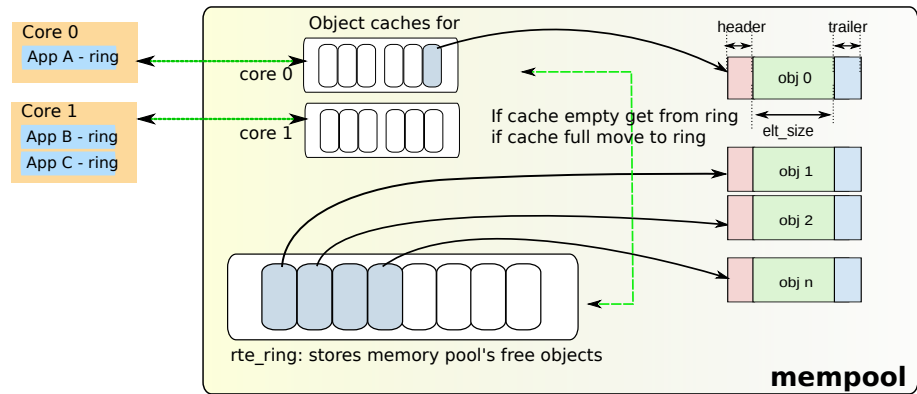


FIGURE i.8. – **Structure d'une mempool.** Source : [Intel 2014]

pour un paquet en une seule fois. La gestion des pointeurs et la structure d'un paquet DPDK est similaire à la structure `sk_buff`. Cependant, cette méthode est moins flexible que la solution de la pile Linux qui permet de séparer la gestion des structures des métadonnées et des données des paquets.

3.5. Gestion du traitement des paquets (POLL MODE DRIVER)

Le *poll mode driver* est constitué d'API qui permettent de gérer le pilote de la carte réseau en espace utilisateur. Cela permet notamment d'accéder directement aux descripteurs des files d'attente de réception et de transmission sans interruption. Cependant, l'application qui utilise les fonctions de cet API doit être développée avec les principes de précaution suivants :

- Cardinalité $<1 \dots 1>$ entre un cœur logique et une file d'attente en réception ou en transmission (pour éviter d'utiliser des verrous sur une ressource qui est partagée);
- Recevoir un maximum de paquet, les traiter et les envoyer après accumulation (traitement par lot);
- etc.

3.6. Améliorations apportées par DPDK

En résumé, les différentes bibliothèques de DPDK citées précédemment répondent aux besoins d'une pile réseau « rapide » (cf. section 2.4 (p. 14)). DPDK permet de

- de préallouer et de réutiliser les tampons en mémoire grâce à la *mempool*;
- d'utiliser efficacement les différentes files d'attente de la technologie RSS;
- d'éviter de copier les paquets du noyau vers l'espace utilisateur;
- d'effectuer des traitements par lot (gestion de la mémoire, réception et envoi des paquets);
- etc.

L'objectif du stage est de tester les possibilités d'un système de filtrage de paquets avec la bibliothèque DPDK. Cette solution de filtrage fonctionne comme un pont filtrant, il reçoit les paquets sur une interface, il les traite puis renvoie les paquets considérés comme « légitimes » sur une autre interface.

J'ai choisi le langage C pour des raisons de performances et de compatibilité avec DPDK et j'ai écrit l'intégralité du code de l'application. Sauf indication contraire, je ne fais état dans ce rapport que des parties de mon code qui sont fonctionnelles.

1. Approche générale du développement

Le système de filtrage anti-DDoS devra avoir les propriétés suivantes :

- l'application doit agir comme un pare-feu, il ne modifie pas les paquets.
- l'application regarde chaque paquet et selon les règles portant sur les informations de la couche IP, de la couche de transport ou dans le contenu décider de laisser passer le paquet ou le jeter.
- on choisit de réaliser un système sans états : les paquets sont traités indépendamment les uns des autres.
- l'application doit atteindre un objectif de performance en terme de nombre de paquets par seconde.
- l'application doit permettre l'implémentation de tests de manière flexible.

Pendant le développement, j'ai gardé aussi à l'esprit les caractéristiques suivantes :

Flexibilité : Le système de filtrage pourra utiliser plusieurs algorithmes de recherche différents ;

Multi processus : À la place de l'utilisation classique de DPDK comportant plusieurs fils d'exécution, l'application utilisera plusieurs processus pour plus de sécurité et de fiabilité ;

Fiabilité : Pour limiter le temps moyen entre pannes, l'application disposera d'un système de récupération .

Architecture du pont filtrant Il existe plusieurs modèles d'architecture possibles : du modèle simple où chaque cœur va traiter un flux de paquet (de la réception à la transmission), à une architecture plus complexe permettant de faire de la programmation parallèle sur le même flux de paquets.

La programmation parallèle nécessite l'utilisation d'une structure en tube, où les flux de paquets passent par plusieurs processus entre la réception et la transmission. Ce changement de processus nécessite un changement de cœur logique¹ ce qui diminue les performances à cause de la localité de la mémoire (le cœur logique suivant ne possède pas le paquet dans sa table TLB, ce qui engendre au minimum un *cache miss*), et des accès concurrentiels au niveau de la mémoire (lorsque celle-ci est partagée [Egi et coll., 2008]).

J'ai donc choisi d'utiliser le modèle simple pour éviter de rentrer dans les optimisations de mémoire liées aux changements de cœurs. Chaque processus ne s'occupe que d'un flux de paquets : de la réception, au traitement par l'algorithme de filtrage, jusqu'à la transmission (cf. figure ii.1).

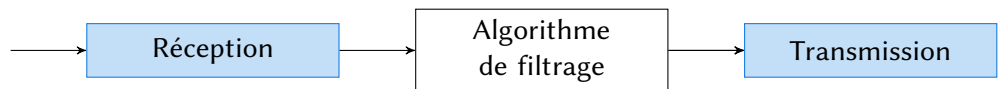


FIGURE ii.1. – **Traitement d'un flux de paquets par un cœur.** La partie bleue correspond à la base de ma solution de filtrage.

Mon application peut être divisée en trois parties : l'initialisation, le chemin des données et les algorithmes de filtrages.

La première partie permet d'initier et de configurer les différentes structures nécessaires à l'application.

Le chemin de données correspond à la structure qui va prendre en charge les paquets (réception, transmission ou libération du tampon). Il a pour but d'amener ces paquets aux systèmes de filtrage.

Les algorithmes de filtrage correspondent aux systèmes de décision des flux de paquets.

2. Initialisation de l'application

L'initialisation est une étape importante pour pouvoir optimiser les structures de données pour le traitement des paquets. J'ai essayé de limiter l'utilisation de la bibliothèque DPDK. L'initialisation est aussi très longue, car il faut aussi configurer la carte réseau. Voici une liste non exhaustive des étapes à réaliser :

Initialisation des objets liés à DPDK :

- création et configuration *mempool* pour les paquets ;
- configuration des ports ;
- configuration des files d'attente de réception et de transmission ;
- configuration de la technologie RSS.

Initialisation de l'application :

- historique d'évènements par processus ;
- affinité ;

1. Chaque processus est associé à un cœur logique.

Processus esclaves Chaque fil d'exécution va créer un seul processus esclave grâce à un `fork()` (sauf le premier fil qui deviendra le processus maître). Les processus ainsi créés gardent toujours l'associativité 1:1 avec un cœur logique. Cependant pour des raisons de flexibilité et de compatibilité avec le système de récupération (cf. section 2.5), les nouveaux processus vont être dissociés de leur affinité avec leur cœur logique pour être réassociés avec un nouveau cœur à partir d'une collection de cœurs logiques libres. Cette réassociation permet de créer des processus « flottants » qui auront la possibilité de changer de cœurs logiques et de configuration à la demande.

Ces processus esclaves auront accès à deux structures partagées :

- La configuration des algorithmes en lecture seule ;
- Les statistiques du cœur logique pour lequel il est associé, en lecture et en écriture.

Processus maître Le processus maître, ainsi que les n fils d'exécution ont accès aux mêmes structures :

- La structure de configuration en lecture et en écriture ;
- Toutes les statistiques en lecture seule.

Fils d'exécution Les fils d'exécution vont attendre la fin de l'exécution de leurs processus fils.

2.2. Configuration des processus

L'application utilise deux fichiers de configuration, voir illustration ii.3.

Le premier fichier permet de configurer les différents processus (algorithmes, files d'attente, et port en réception et transmission).

Le code ii.1 illustre un exemple de configuration. Trois processus sont créés (donc 3 cœurs logiques utilisés). Le premier processus utilise la file d'attente 0 du port 0 en réception et la file d'attente 1 du port 0 en transmission. Il exécutera l'algorithme BPF_PCAP et utilisera pour cela le fichier `bpf.conf`. En résumé, ce fichier exemple de configuration va me permettre d'utiliser deux files d'attente RSS, en réception sur le port 0, qui vont être associés aux processus 0 et 1. Le troisième processus sert uniquement pour le débogage et permet d'avoir une liaison bidirectionnelle.

```
3 -> nombre de lignes de commentaires
Exemple de fichier de configuration
proc    pRx pTx qRx qTx Engine      args
0       0  1  0  0  BPF_PCAP    bpf.conf
1       0  1  1  1  HASH        tuple.conf
2       1  0  0  0  DEFAULT
```

Code ii.1 – **configuration des processus**, *pRx* et *pTx* correspondent aux ports en réception et transmission respectivement, *qRx* et *qTx* correspondent aux files d'attente en réception et transmission respectivement.

Le second fichier permet de configurer les files d'attente RSS. J'ai utilisé le langage YAML et j'ai créé un analyseur syntaxique pour la configuration de ces files d'attente.

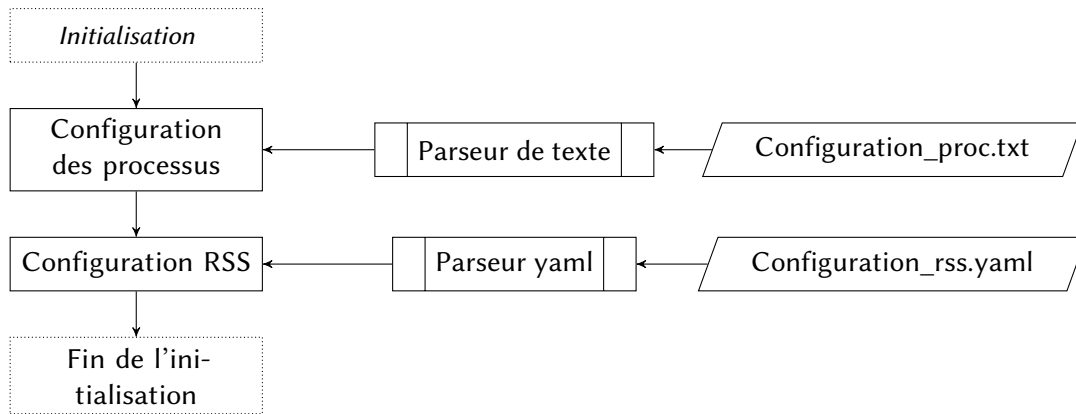


FIGURE ii.3. – **Diagramme de flux récapitulant l'initialisation de la configuration.**

2.3. Communications inter processus

Il est nécessaire de synchroniser les différents processus entre eux puisqu'ils ne partagent pas le tas. Cependant, l'utilisation des IPC peut être sujet à des problèmes de performances si elles utilisent des verrous ou des appels systèmes.

Pour éviter ces verrous, j'ai utilisé une *mempool* dédiée et des anneaux de tampons. Chaque processus esclave dispose de deux tubes de communication unidirectionnels et privatisés avec le processus maître. Le processus maître gère la synchronisation et les processus esclaves ne communiquent pas entre eux.

2.4. Statistiques

Les statistiques vont nous permettre d'obtenir des informations sur le traitement des paquets effectué par les différents cœurs. Plusieurs types de statistiques sont collectés par l'application.

L'application va compter le nombre de paquets reçus, transmis, non transmis² et filtrés.

L'application va aussi récupérer les statistiques de la carte réseau. La carte va mettre à jour plusieurs compteurs : le nombre de paquets reçus qui ne sont pas pris en charge par l'application, le nombre de paquets reçus qui comportent des erreurs, le nombre de paquets envoyés par la carte, etc. Grâce à ces valeurs, il est possible de détecter le lieu de la perte de paquet le cas échéant.

Le processus maître centralise les statistiques de chaque cœur logique et les affiche. En même temps, il récupère les statistiques de la carte réseau.

Les statistiques obtenues par cœur logique et par interface réseau sont affichées selon une période de temps configurable, elles peuvent être enregistrées dans un fichier csv (*comma-separated values*). Un exemple de sortie dans le terminal est illustré en annexe iii.1, page 54.

Pour avoir un aperçu des statistiques, j'ai écrit un script en R³ pour lire le fichier csv et dessiner dynamiquement le graphique. Un exemple est illustré

2. paquets qui ne sont pas pris en charge par la carte, c'est à l'application de libérer les tampons.

3. [R Core Team 2015]

en annexe 1, page 55.

2.5. Système de récupération

Les systèmes de filtrage de l'application sont susceptibles de s'arrêter brutalement, par exemple, à cause d'une mauvaise configuration d'un algorithme. Dans ce cas, l'application doit continuer de s'exécuter pour éviter de perdre des paquets. J'ai donc étudié les possibilités pour améliorer la fiabilité de l'application en utilisant plusieurs processus sur DPDK.

J'ai donc créé un système de récupération à partir des processus flottants (cf. figure ii.4). Les fils d'exécution vont se réveiller lorsque leurs processus fils (esclaves) s'arrêtent volontairement ou anormalement. En utilisant un verrou tournant, ils vont modifier la structure de configuration des processus (qui n'est normalement pas accessible pour les processus esclaves) pour signaler qu'un cœur logique s'est arrêté puis ces fils d'exécution vont aussi s'arrêter.

Le processus maître vérifie à chaque tour de boucle les drapeaux reportant l'état d'exécution du processus. Si un processus a besoin d'être relancé, la sous-routine associée va recréer un unique fil d'exécution qui va suivre le même chemin dans le diagramme des flux que les fils d'exécution initiaux.

Du point de vue de la bibliothèque DPDK, la création d'un nouveau processus affilié à un cœur logique DPDK (lcore) permet à ce processus de pouvoir accéder aux tampons de mémoires déjà existantes. Le nouveau processus peut donc accéder aux files de tampons des paquets du processus qui s'est arrêté.

Pour aller plus loin, ce système permettra au processus maître de lancer des sous-routines particulières en fonction de l'état des processus esclaves. Pour l'instant, seule la sous-routine qui permet de relancer les processus esclaves est développée et fonctionnelle.

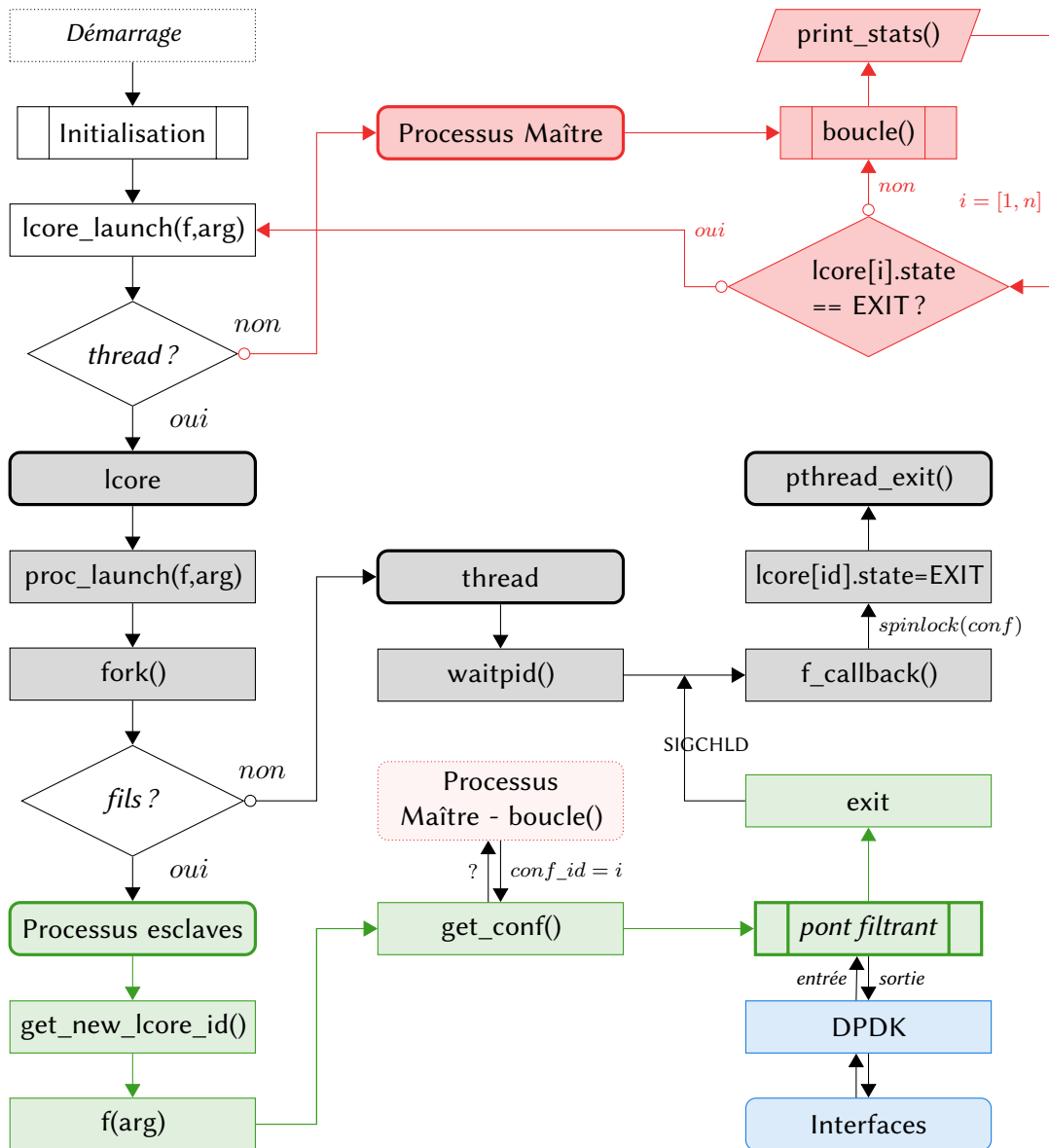


FIGURE ii.4. – **Diagramme de flux de récapitulant l'initialisation des processus esclaves et du système de récupération.** En rouge, vert et gris, flux correspondant aux processus maître, esclaves et aux fils d'exécution du maître respectivement.

3. Chemin des données

Le chemin des données correspond à la partie du code de l'application qui va traiter les paquets (de la réception à la transmission). La performance de l'application dépend en partie de l'optimisation de cette partie du code. J'ai donc suivi les directives suivantes :

- Éviter l'utilisation d'entrées-sorties mémoires comme *malloc()* qui ne sont pas gérées par DPDK ;
- Éviter les fonctions de la libc qui ne sont pas optimisées lorsque cela est possible et utiliser les alternatives optimisées dans DPDK [Intel 2014] ;
- Utiliser la *mempool* pour les allocations mémoires, dans le but de profiter des mécanismes de gestion de la mémoire de DPDK et pour éviter les verrous associés à certaines fonctions comme *malloc()*.
- Éviter l'utilisation des structures partagées. Les seules structures partagées existantes sont
 1. Les statistiques ;
 2. La configuration ;
 3. La *mempool* ;
 4. Les IPC.

À part la *mempool* qui est gérée par DPDK. Les structures partagées utilisées ne disposent pas de verrous, car chaque écrivain n'écrit que dans sa partie privée.

L'optimisation du code est réalisée via l'utilisation :

- d'une directive de compilation *inline* qui permet d'étendre le code d'une fonction dans celui de la fonction appelante et permet d'éviter le coût d'un appel de fonction ;
- d'un alignement des structures par rapport au cache et d'une augmentation de la localité des données dans la même ligne de cache. Cette optimisation est aidée par l'intermédiaire d'une directive de compilation qui va utiliser du bourrage entre les variables ;
- du préchargement de données dans les caches L1 ou L2 du processeur (pour éviter les caches *miss*) ;
- d'une prédiction de branchement par l'intermédiaire d'une directive de compilation. La prédiction de branchement permet au processeur de favoriser la partie la plus « probable » d'une condition et d'exécuter spéculativement le saut d'instruction associé avant la fin de l'exécution de la condition.

3.1. Réception des paquets

La technologie RSS permet de distribuer la charge de travail sur plusieurs processus. Ce travail de distribution est effectué dans la carte réseau.

Une fois que les processus ont été initialisés, chaque cœur logique va lire sur une file d'attente RSS spécifique. Il est nécessaire que chaque file d'attente RSS ne soit associé qu'à un seul processus esclave, car elle ne permet pas un accès concurrentiel.

La carte Intel permet d'utiliser deux systèmes de filtrage : à base de fonctions de hachage (signatures) ou *via* l'utilisation d'un masque. J'ai utilisé ce dernier pour pouvoir filtrer directement les paquets dans la carte réseau, cf. 4.3 (p. 31).

3.2. Traitement des paquets et transmission

Le diagramme des flux de paquets dans l'application est illustré dans la figure ii.5. Dans un premier temps, chaque processus esclave demande au processus maître la configuration des files d'attente et de l'algorithme utilisé, cf. section 2.2 (p. 24). Ensuite, tous les processus esclaves rentrent dans une boucle correspondant à l'attente active nécessaire pour éviter les appels systèmes.

Si on suit le diagramme des flux à partir de la réception des paquets, on utilise une fonction de l'API (`rx_packets`) qui nous renvoie un tableau de descripteur des tampons contenant les paquets reçus. L'idée générale est de traiter plusieurs paquets avant de les envoyer. Pour chaque paquet reçu, le processus va décider le sort du paquet par l'algorithme de filtrage. Dans le cas où le paquet devrait être transmis, la fonction `add_packet` permet d'ajouter le paquet dans la liste des paquets prête à être transmise. Si cette liste est remplie (`n_tx_pkts > max`), tous les paquets de la liste sont transmis par l'intermédiaire de la fonction `send_packets()`. Pour éviter d'attendre l'arrivée de nouveaux paquets pour remplir la liste, un compte à rebours est utilisé pour envoyer les paquets de la liste à la fin de ce compte à rebours.

Ce système permet de traiter les paquets par lot et de réduire les coûts de transmission. Ce système introduit aussi une latence supplémentaire qui est limitée à 100 μ s dans mon application. Cette valeur est configurable.

3.3. Utilisation du jeu d'instructions SSE

Dans l'optique d'améliorer les performances de certains algorithmes, j'ai développé des processus esclaves dont la boucle de traitement des paquets permet d'utiliser les jeux d'instructions à architecture instruction unique, données multiples ou *Single Instruction Multiple Data* (SIMD). En effet, certains algorithmes de filtrage (dans mon cas, l'algorithme de hachage utilise un crc sur 32 bits) effectuent de nombreuses opérations qui peuvent être effectuées sur plusieurs jeux de données simultanément.

Dans le chemin des données de l'algorithme de hachage, le traitement des paquets s'effectue par groupe de 4 (la taille des registres des instructions SSE est de 128 bits). Lorsque le nombre de paquets est inférieur à quatre, chaque paquet est traité individuellement.

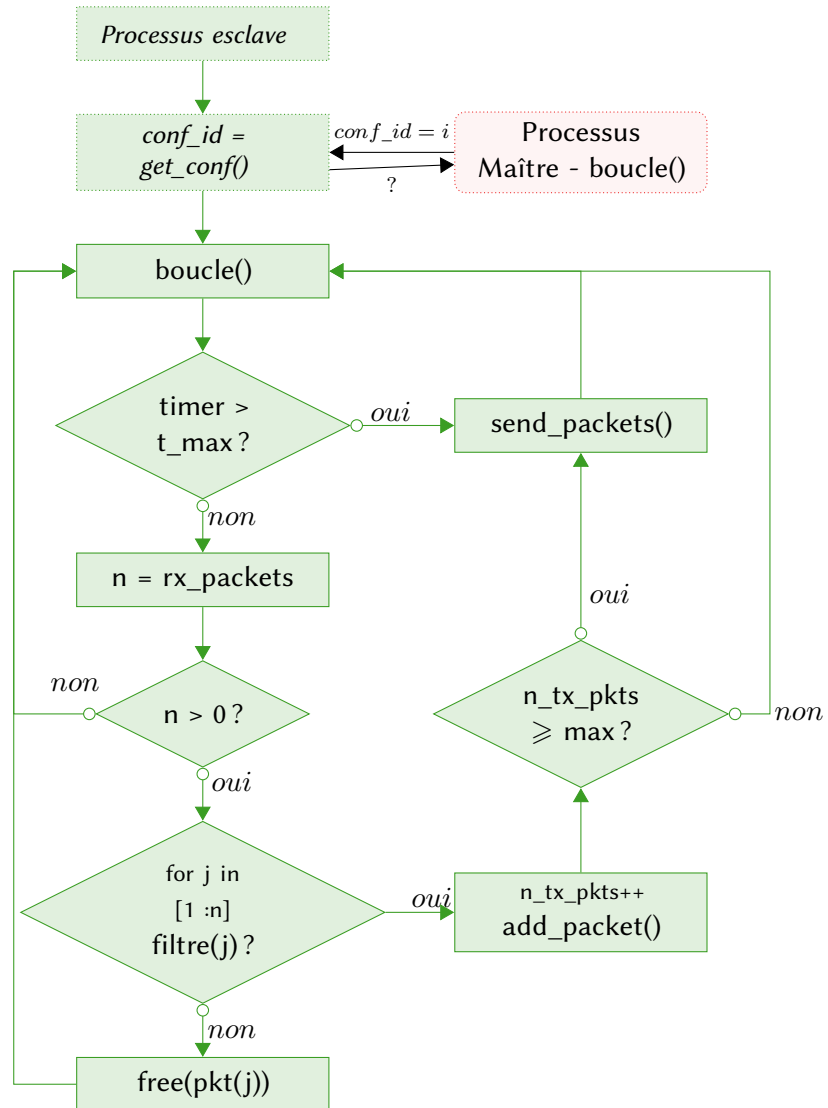


FIGURE ii.5. – **Diagramme de flux du traitement des paquets.** En rouge et en vert : flux correspondant aux processus maître et esclaves respectivement.

4. Algorithmes de filtrage

Il existe plusieurs systèmes de filtrage, j'ai choisi d'utiliser un modèle similaire à un pare-feu sans états (*stateless*) pour des raisons de simplicité et de contraintes temporelles de développement. Un système sans états est plus performant, car il n'a pas besoin de garder en mémoire les connexions en cours, sa protection reste valable pour se prémunir de certaines attaques DDoS. Cependant, certaines attaques nécessitent des solutions de protections plus intelligentes.

J'ai choisi d'implémenter divers algorithmes de filtrage permettant d'appliquer des règles de filtrage sur les en-têtes (couche réseau et transport) et sur le contenu des paquets. La table ii.1 résume les différents algorithmes implémentés.

Algorithme	Vecteurs	Commentaires
L2_forward	-	Redirection de port
Modulo(n)	-	Jeter ou garder un paquet tous les n paquets
RSS	-	Fonction de hachage
BPF	en-tête et contenu	Redirection de port
fast-str	contenu	Recherche de chaîne de caractères
CRC-32	quintuplet	Fonction de hachage

TABLE ii.1. – *Algorithmes implémentés.*

4.1. Redirection de port

La redirection de port redirige tous les paquets reçus provenant d'un port vers un autre port en laissant passer tous les paquets sans distinction. Ce module permet d'effectuer des tests de performance et de débogage. Dans une situation de filtrage d'attaques DDoS, la redirection de port conjointement avec la technologie RSS peut être utilisée pour laisser passer le trafic en suivant une liste blanche.

4.2. Modulo

La fonction modulo est utilisée pour filtrer les paquets de manière déterministe. L'utilisation de cet algorithme pour filtrer des paquets de déni de service n'est pas utile, mais elle permet d'effectuer des tests de filtrage.

4.3. Déchargement vers la carte réseau

L'utilisation des fonctions de déchargement représente une économie de traitement par le processeur. Les cartes réseau offrent plusieurs fonctionnalités pouvant être utiles dans la solution de filtrage, dans notre cas, trois d'entre elles peuvent nous intéresser.

Par exemple, il est possible de vérifier le somme de contrôle (CRC) des paquets IP. La carte peut aussi détecter la nature du protocole utilisé dans la couche 2, 3 ou 4 du modèle OSI. Cependant, il est nécessaire de recevoir le paquet avant de le filtrer : la carte réseau active des drapeaux dans la structure de métadonnées du paquet.

La fonction de déchargement la plus intéressante est la technologie RSS. Outre la possibilité déjà évoquée, de pouvoir distribuer les paquets sur plusieurs processeurs. Il est aussi possible de configurer une file d'attente où les paquets sont directement rejetés par la carte réseau sans jamais rentrer dans le système d'exploitation. La possibilité de filtrer directement à partir des cartes pourrait être utile pour décharger les processeurs et économiser les coûts de réception, de traitement et de transmission des paquets.

Cependant, il faut garder à l'esprit que le nombre de règles est limité et qu'il dépend de la carte réseau.

Mon application permet de configurer cette fonction de filtrage par l'intermédiaire du fichier de configuration en YAML.

4.4. BPF

BSD *Packet Filter* (BPF) est usuellement un agent utilisé dans les systèmes UNIX pour filtrer les paquets directement dans le noyau ce qui évite le surcoût lié au passage dans l'espace utilisateur. Il existe des accroches dans la pile réseau dans laquelle BPF peut s'exécuter (par exemple sur une interface ou sur une socket). BPF peut même filtrer les paquets à partir de la mémoire DMA de la carte réseau ce qui évite les coûts liés à la gestion de ces paquets [McCanne et Jacobson 1993]. BPF est un mécanisme destiné à spécifier aux couches réseau du noyau les paquets qui nous intéressent.

BPF est implémenté sous la forme d'un interpréteur d'une machine virtuelle avec une pile spécialisée. Le code octal (*bytecode*) BPF est interprété dans cette machine virtuelle et est constitué d'un ensemble déterminé d'instructions. Il est compilé à la volée ou *just-in-time* (JIT) en code machine et est exécuté par le processeur.

Dans le noyau Linux, BPF est appelé *Linux Socket Filtering* (LSF) [Linux BPF]. Il contient un jeu d'instructions étendu, appelé eBPF, et dispose d'un interpréteur et d'un compilateur JIT intégré dans le noyau. BPF est utilisé dans de nombreuses fonctions réseaux comme les applications qui utilisent la bibliothèque pcap (*wireshark*, *tcpdump*). BPF est aussi utilisé par *netfilter* et le pare-feu du noyau linux, dans *dhclient* ou dans le système de contrôle de trafic qdisc, etc.

BPF est réutilisé dans *seccomp* pour filtrer les appels systèmes d'un processus dans le noyau et il est aussi utilisé pour filtrer les informations provenant de différents points de traçages du noyau.

```
> tcpdump -i eth0 arp -ddd
(000) ldh      [12]
(001) jeq      #0x806          jt 2      jf 3
(002) ret      #-1
(003) ret      #0
```

Code ii.2 – génération de bytecode formaté pour la lisibilité.

Le code ii.2 est un exemple de *bytecode* généré avec l'application *tcpdump*. Ici, le *bytecode* permet de filtrer les paquets qui utilisent le protocole *Address Resolution Protocol* (ARP) sur l'interface *eth0*. L'expression pcap pour filtrer les paquets (ici "ARP") est compilée par la bibliothèque pcap en *bytecode*. Ici le *bytecode* est facile à lire :

- (000) charger le demi-mot à l'octet 12 dans l'accumulateur. C'est-à-dire lire le champ `EtherType` de la couche Ethernet ;

- (001) aller à (002) si égal à 0x806 (ARP), sinon aller à (003);
- (002) retourne -1;
- (003) retourne 0.

Le *bytecode* est compilé à la volée dans le noyau et est vérifié pour que le programme BPF ne puisse pas créer de boucles dans l'exécution et que les instructions lisent à des endroits valides de la mémoire. Le nombre d'instructions est limité à 4096. La figure ii.6 illustre le fonctionnement général de BPF dans le système d'exploitation Linux.

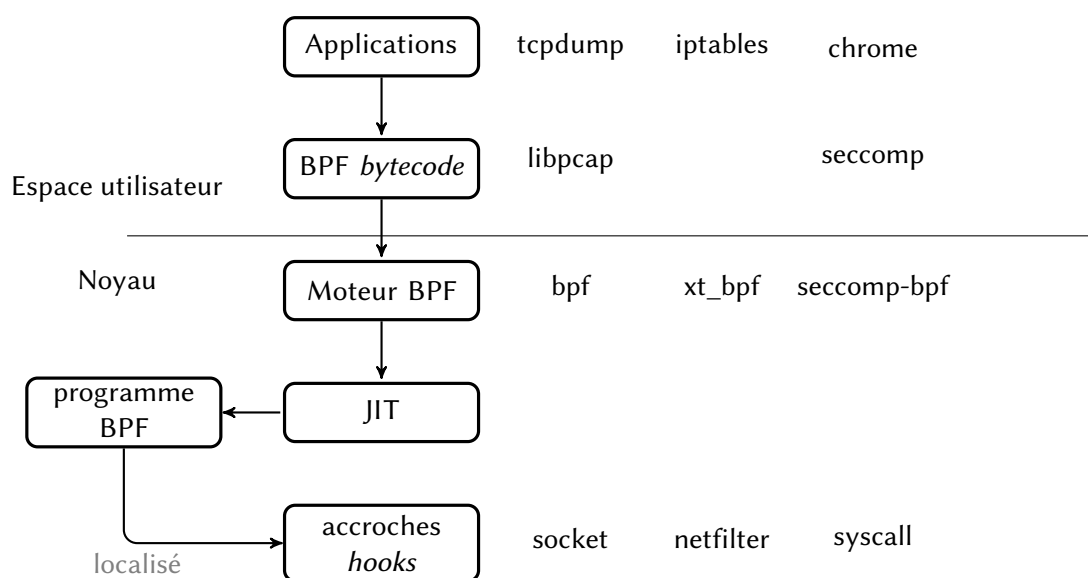


FIGURE ii.6. – *Utilisation de BPF dans le kernel Linux 3.18.*

BPF est un outil puissant pour filtrer des paquets dans le noyau et pourrait être réutilisé pour faire du filtrage en espace utilisateur. Certains CDN utilisent BPF pour lutter contre les attaques volumétriques utilisant le protocole DNS. C'est pourquoi j'ai décidé de réutiliser ce système pour tester son efficacité dans l'application de filtrage. J'ai modifié une bibliothèque de BPF⁴ pour être utilisable avec le langage C. Cette méthode a plusieurs avantages. La première est de pouvoir utiliser BPF en espace utilisateur, évitant ainsi les problèmes de copies de paquets ou les appels systèmes coûteux. L'autre méthode possible consiste à utiliser le *back-end* du compilateur LLVM.

4.4.1. Intégration

La figure ii.7 (p. 34) représente l'architecture de BPF que j'ai choisi d'utiliser dans le *bridge* filtrant (en bleu). La configuration des filtres BPF est obtenue soit par une expression pcap qui est compilée par la bibliothèque libpcap, soit en injectant directement le *bytecode* désiré. La seconde méthode permet d'obtenir une plus grande flexibilité du *bytecode* car certains filtres ne peuvent être exprimés simplement avec des expressions *tcpdump*. J'ai réutilisé les outils provenant de la société CloudFlare pour créer les *bytecode* nécessaires pour filtrer des paquets de requêtes DNS contenant des noms de domaine particulier.

4. Cette bibliothèque est celle utilisée par Linux dans le Noyau. Elle a été modifiée pour être utilisable en espace utilisateur pour comparer les performances entre BPF et PFLua, un système de filtrage de paquet en Lua : <https://github.com/Igalia>.

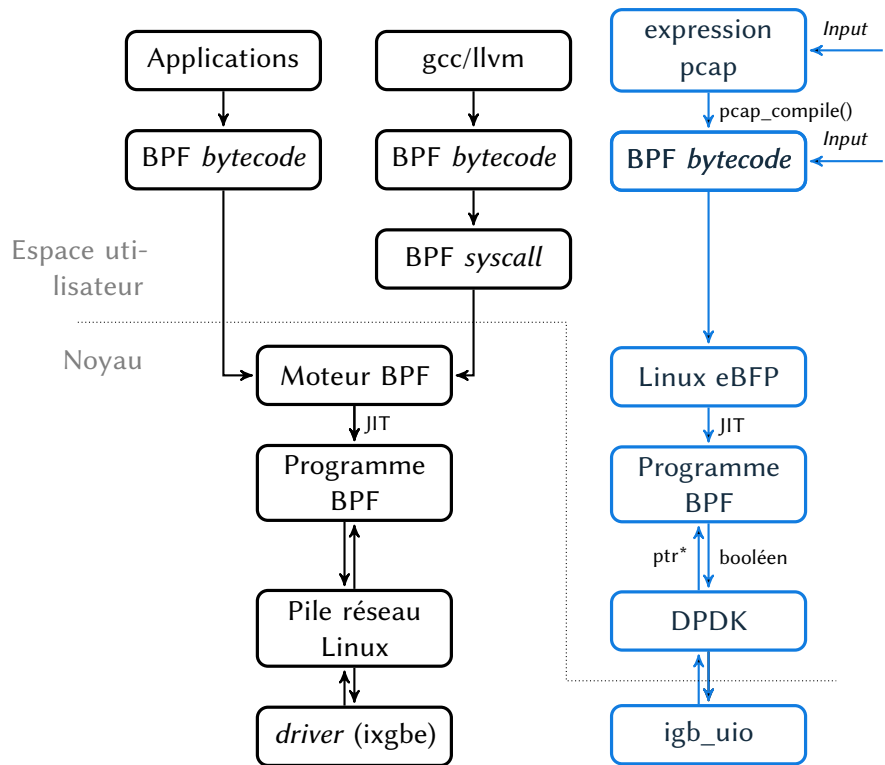


FIGURE ii.7. – **Utilisation de BPF dans mon application.** Utilisations possibles de BPF. En noir, BPF est utilisé classiquement par les applications ou par l'intermédiaire d'un appel système BPF (noyau > 3.18). En bleu, architecture utilisée dans le bridge.

4.5. Algorithmes de recherche de correspondances

Pour détecter les attaques DDoS au niveau des en-têtes, plusieurs champs peuvent être intéressants. Par exemple, les adresses IP peuvent être utilisées comme règles de filtrage pour se protéger d'une attaque volumétrique par réflexion qui utiliserait des résolveurs ouverts. Dans ce cas, il peut être avantageux de bloquer ces adresses IP si la proportion du trafic légitime provenant de ces résolveurs est faible. De même, il est possible d'utiliser les ports comme règles de filtrage : par exemple dans le cas d'une inondation par des paquets UDP sur le port 80.

Il existe de nombreux algorithmes de recherche. Pour les adresses IP, on peut citer l'arbre radix ou le LC trie (*Level compressed Trie*) qui sont des arbres de recherche utilisés par le noyau Linux. Ces algorithmes ont le défaut de nécessiter plusieurs accès mémoires pour effectuer la recherche de correspondance. DIR-24-8 est un algorithme de recherche du préfixe le plus long (*Longest Prefix Match*) [Gupta et coll., 1998] et semble être le plus adaptée dans notre solution de filtrage. Il permet de faire une recherche en temps constant avec seulement un accès mémoire dans le meilleur des cas et deux à trois accès mémoires dans le pire des cas. Son principal désavantage est l'espace pris en mémoire et sa complexité de mise à jour qui est en $O(n^2)$.

Si l'on utilise plusieurs champs à la fois, la classification des flux peut être réalisé sur un n-uplet avec une fonction de hachage ou par l'intermédiaire de classificateur plus complexe comme l'algorithme *Tuple space search* qui permet de rechercher l'ensemble de règles qui partagent la même spécificité

et de choisir celle qui possède la plus grande priorité [Gupta 2000].

Pour des contraintes temporelles, je n'ai implémenté que le filtrage par fonction de hachage en se basant sur le quintuplet :

- adresse IP source
- adresse IP destination
- port source
- port destination
- protocole

J'ai utilisé l'API de DPDK pour créer et rechercher dans la table de hachage. J'ai utilisé comme fonction de hachage la fonction CRC en l'intégrant dans le chemin de données optimisé qui utilise les instructions SSE pour pouvoir traiter 4 paquets à la fois (voir section 3.3 (p. 29)).

4.6. Algorithmes de recherche de chaînes de caractères

Pour certains cas d'attaques où l'en-tête des paquets n'est pas suffisant pour distinguer les paquets légitimes des paquets illégitimes, il peut être intéressant de rechercher dans le contenu des paquets pour trouver des signatures d'attaques. Par exemple pour le protocole DNS, les serveurs de noms faisant autorité peuvent recevoir des requêtes DNS pour des sous-domaines qui n'existent pas. Dans ce cas, il est possible de filtrer un paquet si le contenu du paquet contient une chaîne de caractères particulière. Par exemple, `.www.exemple.fr`.

Parmi les différents algorithmes de recherche de sous-chaînes de caractères, Rabin Karp et Aho-corasick semblent les plus adaptés pour rechercher plusieurs sous-chaînes de caractères en même temps. Pour n la taille du texte cible, k le nombre de sous-chaînes à rechercher et m la taille combinée de ces sous-chaînes, le premier offre une complexité moyenne de $O(n + k)$ et le second de $O(n + m)$.

Je n'ai eu le temps d'implémenter l'algorithme de Rabin-Karp que dans sa version à une seule sous-chaîne de caractère. La version permettant de rechercher plusieurs sous-chaînes est codée mais n'est pas encore fonctionnelle. Rabin-Karp utilise une fonction de hachage sur une sous-chaîne de caractère du texte et compare le résultat avec l'empreinte de la chaîne de caractère recherché. Lorsqu'on décale la fenêtre de recherche, les propriétés de cette fonction de hachage permettent de recalculer la nouvelle empreinte avec seulement la valeur de l'ancienne empreinte, du caractère retiré et du caractère nouvellement ajouté. Ce processus est analogue à une moyenne glissante et permet de recalculer très rapidement l'empreinte.

J'ai utilisé une version de Rabin Karp qui utilise la somme des chaînes de caractères comme fonction de hachage. Cette implémentation⁵ semble être l'une des plus rapides pour la recherche de chaînes de caractère de petite taille. Je l'ai donc réutilisé et adapté pour ma solution de filtrage en réécrivant l'algorithme et en l'adaptant à mon application

5. https://github.com/RaphaelJ/fast_strstr

5. Résultats

Pour mesurer les performances de l'application, j'ai utilisé deux machines comportant une carte ethernet avec 2 ports 10 Gbit.s⁻¹. La figure ii.8 illustre l'architecture du banc de test. La première machine envoie des paquets sur le premier lien qui relie deux ports des deux machines. La seconde machine va exécuter une application qui va rediriger les paquets sur le second lien qui relie les deux autres ports des machines.

La première machine utilise *netmap*, une autre pile réseau « rapide » qui permet de contourner la pile du noyau Linux. Elle envoie les paquets sur le premier port par l'intermédiaire de *pkt-gen* ou de *tcpreplay* (avec la bibliothèque *netmap*). Le nombre de paquets est mesurée sur le second port. La seconde machine exécute soit l'application de test de DPDK, soit mon application avec différents algorithmes. Les caractéristiques des machines sont données en annexe iii (p. 55).

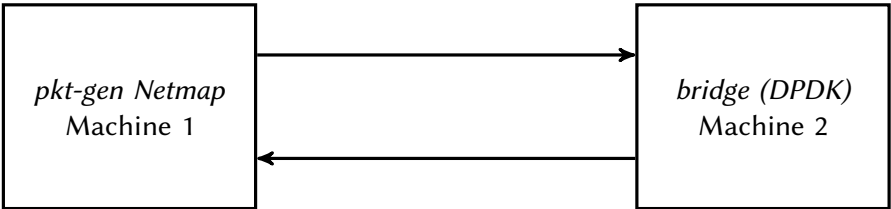


FIGURE ii.8. – Configuration de test.

5.1. Redirection de port

Dans un premier temps, j'ai testé les performance de mon *bridge* en redirection de port (sans filtrage) pour mesurer les performances maximales. La machine 1 envoie des paquets de tailles différentes au débit maximal.

<i>bridge</i> (Mpps)	testpmd (Mpps)	Transmis (Mpps)	Taille (octets)
14.58	14.49	14.88	64
13.22		14.705	65
12.50		13.02	80
8.454		8.454	128
2.352		2.352	512
0.813		0.813	1518

TABLE ii.2. – Débit de réception mesuré par la machine 1. Une redirection de port est utilisé avec la machine 2. La taille des paquets prend en compte le CRC (4 octets).

La table ii.2 résument les mesures des débits envoyés et reçus avec le *bridge* filtrant et l'application de test de DPDK. La figure ii.10 illustre graphiquement les mesures de performance de l'application. Pour comparaison, la performance du routeur virtuel Open vSwitch est de 0,475 Mpps soit 203 Mbps (paquets de 64 octets, en redirection de port et sans DPDK).

On peut observer une légère déviation du débit envoyé par rapport au débit théorique pour les paquets de petites tailles. Elle n'est pas liée à mon application car l'application de test de DPDK (*testpmd*) n'arrive pas non plus à retransmettre tous les paquets. L'observation des statistiques de la carte montrent que l'application ne perd pas de paquets. Il est probable que cette

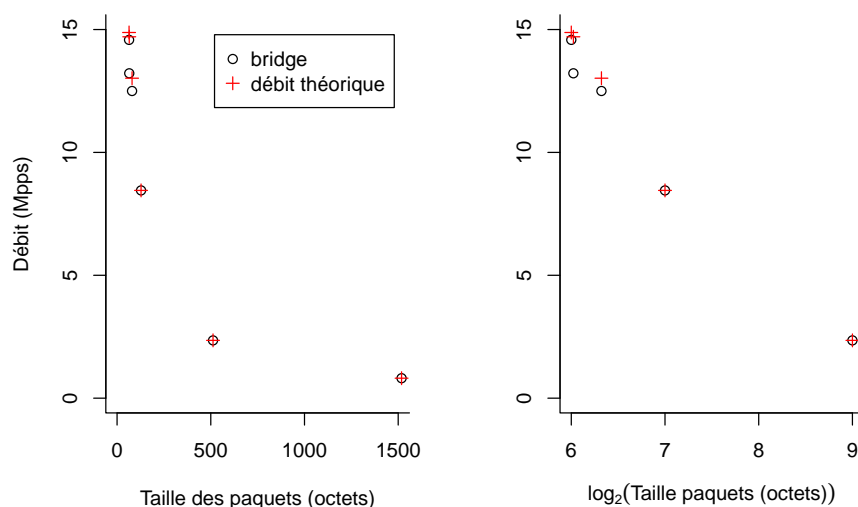


FIGURE ii.9. – **Performance en redirection de port.** Débit de réception mesuré sur la machine 1 avec une redirection de port sur la machine 2 pour des tailles variables de paquets (configuration matérielle 1). L'illustration de droite est une autre représentation des données.

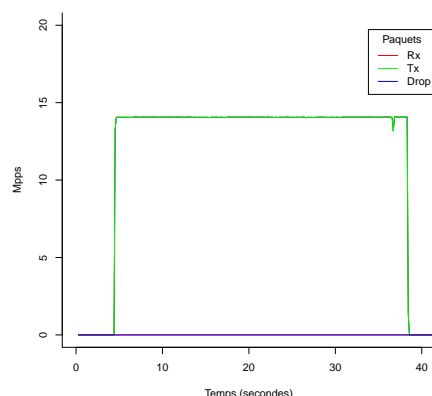


FIGURE ii.10. – Graphique des statistiques pendant une expérience avec des paquets de 65 octets (configuration matérielle 1). La ligne rouge (paquets reçus) est confondue avec la ligne représentant les paquets transmis (verte).

légère déviation soit d'origine matérielle. J'ai effectué des tests avec une machine différente. Les résultats montrent des problèmes de performances plus drastiques (cf annexe iii (p. 56) et figure ii.11). Pour comparaison, en utilisant Open vSwitch et la pile réseau Linux, le débit mesuré est de 500 000 paquets par seconde environ sans utiliser d'optimisations particulières.

Toutes les cartes réseaux ne peuvent pas atteindre les 10 Gbit.s^{-1} avec des paquets de petites tailles. Certaines cartes ne peuvent atteindre le débit maximal en réception avec des petits paquets dont la taille n'est pas multiple de 64. Certaines cartes peuvent transmettre à haut débit mais ne peut pass

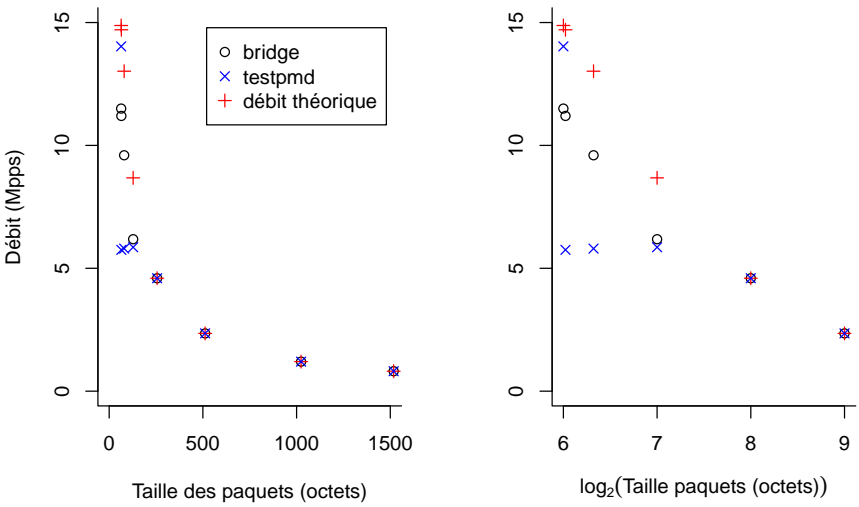


FIGURE ii.11. – **Performance en redirection de port.** Débit mesuré par la machine 1 avec une redirection de port sur la machine 2 pour des tailles de paquets différentes (configuration matérielle 2). L’illustration de droite est une autre représentation des données.

recevoir à la même vitesse [Rizzo 2012]. Les résultats illustrés dans la figure ii.11 sont similaires à ceux obtenus par Luigi Rizzo avec *netmap* [Rizzo 2012]. Cette perte drastique de performance pour des paquets de 65 octets (avec le CRC) est expliquée par un problème de dépassement de la taille de la ligne de cache. Des investigations plus poussées sont nécessaires pour confirmer cette cause dans nos conditions expérimentales.

5.2. Algorithmes

Tous les algorithmes n’ont pas pu être testés par manque de temps. Pour effectuer les tests, nous avons forgé différents paquets qui sont susceptibles d’être utilisés pour des attaques DDoS. Les algorithmes de filtrage ont été testés avec un autre outil (sans DPDK) que j’ai développé, qui compte le nombre de paquets filtrés en rejouant un fichier pcap. La table ii.3 récapitule les exemples d’attaque, avec les signatures qui ont été utilisées pour les détecter.

Protocoles	Attaques mimées	Signatures	Proportion
DNS	Réflexion	noms de domaine dans le contenu du paquet	1
ICMP	ICMP FLOOD	adresses IP et protocole	0.5
Bittorrent	Réflexion	contenu du paquet	0.95

TABLE ii.3. – **Exemples d’attaques utilisées.** La proportion représente la quantité de paquets de paquets à filtrer par rapport à la quantité totale de paquets (qui contiennent des paquets légitimes provenant de connexions à Internet).

L’utilisation des algorithmes de filtrage avec les exemples donnés n’entraîne pas de pertes de paquets de la part de l’application.

Profilage Pour avoir un aperçu des goulets d'étranglements de l'application, j'ai utilisé l'outil de profilage de Linux.

```
# Samples: 244K of event 'cycles'
# Event count (approx.): 149861413262
#
# Overhead Command Shared Object Symbol
# .....
#
# 44.80% bridge bridge [.] ixgbe_rcv_pkts_lro_bulk_alloc
# 33.25% bridge bridge [.] bridge_loop_optimized
# 19.05% bridge bridge [.] get_cmd
# 1.06% bridge bridge [.] rte_hash_lookup
# 0.91% bridge bridge [.] ixgbe_xmit_pkts
# 0.35% bridge bridge [.] hash_pkt_filter
# 0.18% bridge bridge [.] ipv4_hash_crc
# 0.16% bridge bridge [.] bridge_send_burst
# 0.15% bridge bridge [.] rte_hash_k16_cmp_eq
# 0.04% bridge bridge [.] rte_pktmbuf_free
```

Code ii.3 – *Profilage de la fonction de hachage*

On peut observer que les fonctions les plus coûteuses sont celles permettant de recevoir ou de transmettre les paquets (fonctions avec le préfixe `ixgbe`).

`bridge_loop_optimized` est la fonction de boucle qui traite les paquets par groupe de 4. La probabilité d'échantillonnage de cette fonction augmente lorsque l'application passe moins de temps pour le traitement des paquets (réception, traitement, transmission ou libération des tampons).

`get_cmd` est une fonction qui vérifie la présence de messages du processus maître, elle est appelée à chaque tour de boucle ce qui explique son coût. Pour la suite, j'ai retiré cette fonction en attendant de l'améliorer.

Le coût de la fonction de filtrage est faible par rapport aux coûts de la réception et de la transmission. La table ii.4 résume le pourcentage d'utilisation des fonctions de filtrage en fonction de l'algorithme testé. On peut noter le coût élevé de la recherche de chaîne de caractère. En effet, il va parcourir l'intégralité du contenu du paquet (si la signature n'est pas présente). BPF est plus performant pour la même signature car sa recherche est plus ciblée.

Algorithme	Vecteurs de filtrage	Utilisation
BPF-pcap	ICMP	1 %
BPF	Domaine DNS	1,2 %
fast-str	Domaine DNS	11,7 %
fonction de hachage		1 %

TABLE II.4. – *Profilage des algorithmes. Pourcentage d'évènements collecté pour la fonction de filtrage par rapport à l'ensemble des évènements collectés. BPF_pcap correspond à l'utilisation de BPF avec le bytecode généré par la bibliothèque pcap.*

Les accès mémoires ont été mesurés avec l'outil `perf` pendant le filtrage des paquets. Le nombre de cache *miss* est environ 1 % pour tous les algorithmes de filtrage.

```
17 217 404 345 L1-dcache-loads [100,00\%]
180 461 084 L1-dcache-misses 1,05% of all L1-dcache hits [100,00\%]
61 929 955 440 cycles
20,523470969 seconds time elapsed
```

Code ii.4 – *Exemple de profilage de la fonction de hachage*

6. sécurité

D'un point de vue de la performance, DPDK permet d'atteindre les débits désirés. Cependant, est-ce que son utilisation peut entraîner des problèmes de sécurité sur la machine ? En effet, puisque cette bibliothèque permet d'exposer une partie des registres de la carte réseau, on peut donc se demander si son utilisation entraîne des failles de sécurité.

hugepages La gestion de la mémoire est centralisée dans les *hugepages*. Lors de la configuration de la carte réseau, les descripteurs des différentes files d'attente des interfaces de la carte sont associés à une *mempool* qui est accessible en écriture et en lecture à tous les utilisateurs qui ont les mêmes droits pour les *hugepages*. Dans ce cas, il est possible de lire ou d'écrire dans toutes les files d'attente associées à la *mempool*.

Privilèges DPDK nécessite les privilèges *root*. Il est possible d'exécuter les applications DPDK en diminuant les privilèges par la modification des permissions du point de montage des *hugepages*, et des fichiers concernant les pilotes en espace utilisateur (*uio*) dont a besoin l'application.

Address space layout randomization (ASLR) C'est une technique utilisée par défaut dans les systèmes d'exploitation actuels qui permet de placer de façon aléatoire les zones de données dans la mémoire virtuelle pour limiter les attaques du type exécution de code arbitraire (qui peut être déclenché par l'exploitation du *buffer overflow*). L'utilisation de plusieurs processus DPDK requiert que les mêmes associations avec les *hugepages* soient présentes dans tous les processus. L'ASLR peut interférer et empêcher l'initialisation de l'application. La documentation de DPDK suggère de désactiver l'ASLR pour permettre la reproductibilité de l'initialisation. La désactivation de cette technologie peut entraîner des problèmes de sécurité.

Si l'ASLR est activé, il est possible de contourner ce problème d'initialisation en relançant plusieurs fois l'application. Pour éviter ce problème dans mon application, la création des processus est réalisée par l'intermédiaire d'un *fork* ce qui permet de garder les mêmes associations avec la *mempool*.

IVSHMEM Lors de l'utilisation de machines virtuelles, les applications peuvent utiliser la bibliothèque « IVSHMEM » à la place de celle utilisée dans notre application (*Poll Mode Driver*). Cette bibliothèque permet d'associer des zones mémoire (DPDK) de la machine virtuelle avec celles de la machine invitée. Cette technologie permet de partager des données entre l'hôte et les invités en « zéro copie ». Ce mode ne doit pas être utilisé dans des machines virtuelles dont on n'a pas confiance car les données corrompues par une machine affecteront toutes les machines.

En conclusion, j'ai répondu aux objectifs du stage en développant une application de filtrage avec la bibliothèque DPDK. Cette application peut filtrer des paquets à des débits relativement élevés. J'ai créé un système modulaire et configurable en construisant une architecture basée sur la fiabilité. J'ai implémenté pour les essais plusieurs algorithmes de filtrage qui se base sur les en-têtes ou le contenu des paquets. Cette application pourrait être utilisée en situation réelle comme le premier élément d'un système de filtrage des attaques DDoS.

Architecture Mon application de filtrage reste une étude de faisabilité et il est possible de l'améliorer sur plusieurs points.

L'architecture de notre application est efficace mais simple. Les paquets restent dans le même cœur logique de la réception à la transmission. Ce modèle n'est pas optimisé pour certains usages. Par exemple, si l'on désire utiliser un filtrage en se basant sur les en-têtes et sur le contenu, il faudrait utiliser deux algorithmes de recherche différents (par exemple, une fonction de hachage et l'algorithme aho-corasick). Or l'utilisation de deux algorithmes sur le même cœur peut engendrer des problèmes liés à la mémoire. Le changement d'un algorithme à l'autre va modifier les données en cache par celles de l'algorithme en cours d'exécution. Lors du traitement du prochain paquet, les données du premier algorithme seront invalidées ce qui génère des caches *miss*.

Pour améliorer l'architecture, on peut se baser sur un cadre de travail qui améliore le traitement en parallèle des paquets. Le même flux RSS peut être pris en charge par plusieurs processus. Un des processus va distribuer les paquets de ce flux à des travailleurs. Ces travailleurs vont récupérer les paquets, les traiter puis renvoyer les résultats au distributeur ou à un autre processus qui poursuivra la chaîne de traitements. L'avantage de cette architecture est de pouvoir multiplier les processus esclaves pour les algorithmes qui nécessitent beaucoup de ressources processeurs. Cependant, dans la conception de l'architecture il faudrait faire attention au réordonnancement possible des paquets.

Cette architecture est similaire au modèle utilisé par FastFlow, un programme d'inspection de paquets en profondeur ou *Deep Packet Inspection* (DPI) qui peut traiter 10 Gbit.s^{-1} en utilisant 8 fils d'exécution et la pile réseau « rapide » PF_RING [Danelutto et coll., 2014]. Le modèle utilisé par mon application actuelle est analogue à celui utilisé dans une étude utilisant Suricata pour traiter 10 Gbit.s^{-1} [Suricata 2012].

Dans tous les cas, il faudra toujours considérer les problèmes liés à la mémoire, c'est-à-dire, d'une part, essayer de garder le flux des paquets (de la réception à la transmission) dans les cœurs du même processeur et d'autre part, résoudre les problèmes de cache *miss* qui peuvent survenir lorsqu'un autre cœur va prendre en charge les paquets d'un autre cœur.

Gestion de la récupération des processus Lorsqu'un processus se termine anormalement, le relancement des cœurs logiques peut prendre du temps s'il faut réinitialiser les structures des algorithmes de filtrage. Dans ce cas, il est susceptible d'y avoir des pertes de paquets. Pour améliorer la récupération, on peut créer de nouveaux processus qui seront des copies dormantes des processus existants. Lors de l'arrêt d'un processus, la copie dormante prendra le relai.

Fragmentation La fragmentation IP n'est pas prise en charge par mon application. Les attaquants peuvent utiliser la fragmentation pour saturer la mémoire des serveurs de la victime ou pour obfusquer leurs attaques. Si on reprend l'exemple du domaine DNS de la section 4.6, il ne sera ni détecté par BPF ni par l'algorithme de recherche de caractères. La gestion de la fragmentation peut être délicate, car il faut garder en mémoire le paquet et ensuite le réassembler. Un attaquant peut viser spécifiquement cette gestion pour que le système garde artificiellement en mémoire des paquets. Il est donc possible d'effectuer un déni de service sur le pare-feu ou sur le système de protection.

Algorithmes Le choix de l'algorithme peut dépendre de la nature de l'attaque. J'ai choisi d'avoir un mécanisme qui accepte différents algorithmes pour laisser le choix à l'utilisateur.

Les algorithmes utilisés dans les solutions commerciales de filtrage d'attaques DDoS ne sont pas connus. Cependant, pour la comparaison on peut s'inspirer du fonctionnement des applications de traitement de paquets. Open vSwitch est un routeur virtuel utilisé dans les réseaux SDN. Il permet de classer les flux en fonction de la couche 2, 3 et 4 du modèle OSI, et possède de base de meilleures performances que le *bridge* de Linux [Emmerich et coll., 2015]. Open vSwitch permet d'accepter un grand nombre de règles (plusieurs millions). Open vSwitch utilise une amélioration d'un algorithme appelé *tuple space search* qui consiste à utiliser une table de hachage pour chaque type de champs utilisé dans les règles de filtrage. Un tuple est défini par un vecteur de longueur k , où k correspond au nombre de champs définis dans les règles de filtrage. Cet algorithme permet d'obtenir des temps de recherche et de mise à jour linéaire pour k inférieur à 2, cependant pour des k plus élevés, la complexité semble ne pas être déterministe [Srinivasan et coll., 1999].

Ce routeur virtuel n'inspecte pas le contenu des paquets. Des études récentes ont été réalisées pour y intégrer le module *conntrack* du noyau de Linux et L7-filter¹ en espace utilisateur [Baudin 2014, Graf et Pettit 2014].

Pour comparaison, FastFlow, un outil d'inspection de paquets en profondeur ou *Deep Packet Inspection* (DPI) ouvert, utilise une fonction de hachage sur un quintuplet (adresses IP, ports et protocole) pour classer les flux par rapport aux en-têtes [Danelutto et coll., 2014].

Pour l'instant, dans mon application chaque processus ne partage pas les structures de l'algorithme, il pourrait être intéressant de partager une structure unique pour plusieurs processus. Cela nécessite que le processus maître mette à jour la structure pour éviter les problèmes liés aux verrous.

1. module de classification de Netfilter.

RSS Les possibilités de la technologie RSS dépendent de la carte utilisée. Le nombre maximal de règles ainsi que le nombre de n-uplets utilisé dans la fonction de hachage dépendent de la carte et du constructeur. L'utilisation de la technologie des cartes Intel *Flow directory*, qui permet de distribuer les paquets ou de les filtrer par l'intermédiaire d'un masque n'est pas flexible : il faut arrêter et redémarrer la carte si on désire changer le masque. Cette opération dure environ 5 secondes, ce qui n'est pas tolérable si on désire éviter les pertes de paquets.

La distribution des paquets par la technologie RSS nécessite aussi un système de surveillance pour connaître l'état du trafic et des flux. Ces informations sont nécessaires pour configurer les files d'attente RSS.

Performances Les performances obtenues par mon application avec la redirection de port sont similaires à celles de la littérature. À cause de contraintes matérielles et temporelles, je n'ai pas pu faire toutes les mesures.

Il aurait été intéressant de savoir si toutes les fonctions de déchargement de la carte réseau sont capables de traiter 14 Mpps.

Certaines applications ont besoin d'une qualité de service qui impose un faible délai ou gigue. La mesure du délai doit être réalisée à différents débits. En effet, si la file d'attente de transmission se remplit rapidement, l'application enverra les paquets aussi rapidement. Si le débit de réception est plus faible alors, il est possible que les paquets doivent attendre la fin du compteur pour être transmis.

La bibliothèque DPDK L'utilisation de DPDK n'est pas aussi aisée que celle d'une *socket* dans la libc.

L'initialisation de toutes les structures nécessaires à l'application est simple et linéaire, mais elle est aussi très longue avec de nombreux paramètres à configurer. Environ 700 lignes de code sont utilisées dans l'application exemple fourni par DPDK pour faire une simple redirection de port. La configuration de ces paramètres est importante pour le fonctionnement de l'application. Par exemple, il faut déterminer à l'avance la taille de la zone mémoire allouée aux tampons, car toutes les structures qui vont contenir les paquets sont préallouées. Si mon application devait gérer la fragmentation, la mémoire (*mempool*) nécessaire pour garder en mémoire les fragments serait au minimum de 3,5 Go² environ. La gestion des paquets fragmentés de petite taille nécessite paradoxalement plus de mémoire que celle des paquets de grande taille puisque la taille des tampons est fixe.

Pour atteindre les performances maximales, il faut prendre en compte l'architecture matérielle dans le développement du logiciel. En effet, il faut considérer l'architecture NUMA (*Non Uniform Memory Access*) pour définir les processus et les cœurs logiques. Il est nécessaire d'associer le processeur (les cœurs logiques), avec le bus PCIe 16x connecté à ce processeur (et donc la carte réseau, et les files d'attente associées). Il faut aussi configurer le nombre de canaux mémoires disponibles dans la machine et il n'existe pas de moyens fiables de déterminer cette valeur dans le noyau Linux [Heminger 2015]. Les performances du processeur dépendent de sa température or DPDK nécessite l'utilisation d'une attente active dans tous ses cœurs logiques, ce qui entraîne une utilisation du processeur à 100 % pour chaque processus même si ces

2. $0,861 \text{ Mpps} \times 0,500 \text{ ms} \times 2000 \text{ octets} \times 4 \text{ fragments}$

processus ne traitent aucun paquet. L'utilisation de *sleep* de très courte durée, pour rendre un cœur oisif, permet d'augmenter les performances du cœur juxtaposé [Heminger 2015].

La bibliothèque DPDK est toujours en cours de développement. La documentation de l'API disponible sur Internet n'est pas dans l'intégralité à jour et il est parfois nécessaire de regarder dans le code source pour y comprendre le fonctionnement des structures ou des fonctions. Selon les versions de DPDK, les noms des variables peuvent être modifiés ou supprimés et d'autres problèmes de compilation peuvent apparaître. Le problème majeur que j'ai pu observer est l'apparition d'une erreur de segmentation qui se déclenche de manière chaotique à l'exécution en utilisant une bibliothèque plus récente.

Est-ce qu'une application qui utilise DPDK pourrait être vendue comme un produit d'étagère ? Oui et non. Il est nécessaire de considérer l'architecture matérielle et logicielle pour configurer l'application ; or la machine utilisée pour le développement sera différente de celle utilisée pour l'exécution de l'application. Il est donc difficile de produire un logiciel générique qui fonctionnera sur tous les types de matériels. De plus, la configuration de DPDK est statique, il est nécessaire à l'utilisateur de comprendre les mécanismes intrinsèques à DPDK pour profiter des meilleures performances.

Néanmoins, les performances obtenues avec DPDK sont excellentes sur des plateformes matérielles connues. Les difficultés liées à la configuration peuvent être compensées par l'utilisation d'un logiciel adapté à du matériel spécifique. Les outils ayant testé l'intégration de DPDK sont de plus en plus nombreux. DPDK est utilisé dans les routeurs virtuels comme Open vSwitch ou Lagopus³ ; dans les outils d'inspection de paquets en profondeur ou *Deep Packet Inspection* (DPI) : FastFlow un DPI ouvert, ixEngine le DPI de la société Qosmos ou celui de la société WindRiver et dans les générateurs de paquets comme pktgen ou Moongen. DPDK est aussi utilisé dans une solution commerciale de pile réseau « rapide » (6WIND).

3. <https://lagopus.github.io/>

Bilan personnel

Les objectifs de mon stage étaient la réalisation d'une preuve de concept d'un outil de protection contre les attaques DDoS en utilisant la bibliothèque DPDK. J'ai pu développer une application modulaire avec plusieurs algorithmes de filtrage qui peut traiter des débits de 10 Gbit.s^{-1} ou de 14 Mpps.

Ce stage a été très enrichissant d'un point de vue technique et humain. Il m'a permis de découvrir l'environnement de travail l'ANSSI. J'ai pu y acquérir et y approfondir mes connaissances dans les systèmes informatiques, dans le fonctionnement des piles réseau des systèmes d'exploitation et aussi dans divers algorithmes de recherche en m'aidant des bases de compétences acquises durant le Master dans les domaines des noyaux des systèmes d'exploitation et des protocoles réseaux. J'ai pu y rencontrer des personnes de divers horizons que ce soit chez les permanents de l'ANSSI ou chez les stagiaires, avec qui j'ai pu discuter et où j'ai pu enrichir ma vision dans d'autres spécialités de l'informatique.

Les sphères d'applications de ces piles réseaux « rapides » sont aussi utiles pour la virtualisation, et sont en étroite relation avec les réseaux SDN (*Software-Defined Networking*) qui est une thématique que j'apprécie. C'est pourquoi j'aimerais continuer vers ces thématiques en gardant l'aspect des réseaux informatiques et de la sécurité.

Remerciements

Je remercie Nicolas Vivet, Pierre Lorinquer, François Contat pour l'encadrement du stage, le temps consacré et l'aiguillage tout le long du stage.

Je remercie Guillaume Valadon pour m'avoir accueilli au sein de son laboratoire et je remercie aussi Florian Maury et Maxence Tury pour les discussions entre deux portes ou entre deux stations de métro.

Je remercie Gregory Fresnais pour l'aide apportée.

Je remercie aussi tous les stagiaires pour les discussions. Je remercie tout autant ceux qui ont réalisé l'exploit d'y faire baisser la productivité tous les jours.

Au final, je remercie aussi Anne Feltz et Boris Barbour pour l'accompagnement au début de cette transition et je m'excuse d'avance pour ma contribution minimale à l'article final.

Liste des acronymes

- ACL** liste de contrôle d'accès ou *Acces Control List*. 8
- API** interface de programmation. 16
- ARP** *Address Resolution Protocol*. 32
- BPF** *BSD Packet Filter*. 32, 33, 39
- bytecode** code octal. 32, 33, 39
- CDN** *Content Delivery Network*. 7
- CPU** processeur ou *Central Processing Unit*. 4, 5, 10–14, 16, 18
- CRC** contrôle de redondance cyclique. 10, 35
- DDoS** par déni de service distribué ou *Distributed Denial of Service*. 1–4, 6–9, 15, 21, 31, 34, 38, 42, 45
- DMA** *Direct Access Memory*. 11, 13, 14, 16, 32
- DoS** par déni de service ou *Denial of Service*. 1, 3
- DPI** inspection de paquets en profondeur ou *Deep Packet Inspection*. 41, 42, 44
- FPGA** circuit logique programmable ou *Field-Programmable Gate Array*. 10
- GPU** *Graphic Processing Unit*. 16
- IDMS** *Intelligent DDoS Mitigation System*. 1
- IDS** système de détection d'intrusion ou *intrusion detection system*. 15
- IPC** communications inter-processus. 18, 22, 25, 28
- IRQ** interruption matérielle ou *Interrupt Request*. 11, 12, 15, 16
- JIT** à la volée ou *just-in-time*. 32
- LSF** *Linux Socket Filtering*. 32
- MTU** taille maximale de transmission ou *Maximum Transmission Unit*. 5
- RSS** *Receive-side Scaling*. 10, 13, 14, 20, 22, 24, 28, 31, 32, 41, 43
- SDN** *Software-defined Networking*. 10, 42
- TLB** *Translation Lookaside Buffer*. 14, 17, 18, 22

Bibliographie

Note: Le signe * indique que le contenu de l'article mérite l'attention dans le cadre de notre étude.

[Akamai Technologies 2015] Akamai Technologies. *Q1 2015 State of the Internet – Security Report*. <https://www.stateoftheinternet.com/resources/web-security-2015-q1-internet-security-report.html>, 2015. **Note:** Statistiques des attaques DDoS récoltées tous les trimestres par la société Akamai Technologies. (Cité en pages 2 et 4.)

[ANSSI 2015] ANSSI. *Comprendre et anticiper les attaques DDoS*. <http://www.ssi.gouv.fr/guide/comprendre-et-anticiper-les-attaques-ddos>, 2015. (Cité en page 2.)

* **Note:** Le guide publié par l'ANSSI fait état des attaques DDoS pour mieux les comprendre et mieux s'en protéger.

[Arbor Networks 2015] Arbor Networks. *When the Sky is Falling*. http://www.cisco.com/c/en/us/products/collateral/security/traffic-anomaly-detector-xt-5600a/prod_white_paper0900aecd8011e927.html, APRICOT 2015. (Cité en page 2.)

[Barnett 2012] Ryan Barnett. *HOIC DDoS Analysis and detection*, SpiderLabs. Article de blog : <https://www.trustwave.com/Resources/SpiderLabs-Blog/HOIC-DDoS-Analysis-and-Detection/>, 2012. (Cité en page 6.)

[Baudin 2014] Franck Baudin. *Open vSwitch 2014 Fall Conference : L7-filter and Open vSwitch*, Qosmos. <https://www.youtube.com/watch?v=TH3yedYXDuw>, 2014. (Cité en page 42.)

[Cisco Systems 2004] Cisco Systems. *Defeating DDOS Attacks*. http://www.cisco.com/c/en/us/products/collateral/security/traffic-anomaly-detector-xt-5600a/prod_white_paper0900aecd8011e927.html, 2004. (Cité en page 2.)

[Contat 2015] François Contat. *Selective BlackHoling*. In FRNOG 24, 2015. (Cité en page 7.)

[Danelutto et coll., 2014] M Danelutto, L Deri, D De Sensi et M Torquati. *Deep packet inspection on commodity hardware using fastflow*. Parallel Computing : Accelerating Computational Science and Engineering (CSE)(Proc. of PARCO 2013, Munich, Germany), vol. 25, pages 92–99, 2014. (Cité en pages 41 et 42.)

[Dangaard Brouer 2015] Jesper Dangaard Brouer. *Network stack challenges at increasing speeds*, Red Hat Inc. http://people.netfilter.org/hawk/presentations/LCA2015/net_stack_challenges_100G_LCA2015.pdf, 2015. (Cité en page 15.)

[Egi et coll., 2008] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerd, Felipe Huici et Laurent Mathy. *Towards high performance virtual routers on commodity hardware*. In Proceedings of the 2008 ACM CoNEXT Conference, page 20. ACM, 2008. (Cité en pages 14 et 22.)

* **Note:** Une étude expérimentale exhaustive sur les problèmes de mémoires liés au routage virtuel.

[Emmerich et coll., 2015] Paul Emmerich, Daniel Raumer, Florian Wohlfart et Georg Carle. *Assessing Soft-and Hardware Bottlenecks in PC-based Packet Forwarding Systems*. ICN 2015, page 90, 2015. **Note:** Test de performances avec différentes piles réseaux. (Cité en pages 10 et 42.)

[Garcia-Dorado et coll., 2013] José Luis Garcia-Dorado, Felipe Mata, Javier Ramos, Pedro M Santiago del Río, Victor Moreno et Javier Aracil. *High-performance network traffic processing systems using commodity hardware*. Data Traffic Monitoring and Analysis, pages 3–27, 2013. (Cité en pages 12, 14 et 16.)

* **Note:** Une revue exhaustive et très intéressante sur les goulets d'étranglement de la pile réseau du noyau Linux.

- [Graf et Pettit 2014] Thomas Graf et Justin Pettit. *Open vSwitch 2014 Fall Conference : Connection Tracking & Stateful NAT*, VMware and Noiro Networks. <https://www.youtube.com/watch?v=O-6Ksl4-JjA>, 2014. (Cité en page 42.)
- [Gupta et coll., 1998] Pankaj Gupta, Steven Lin et Nick McKeown. *Routing lookups in hardware at memory access speeds*. In INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, volume 3, pages 1240–1247. IEEE, 1998. **Note** : *Un algorithme de recherche d'adresse IP performant* (Cité en page 34.)
- [Gupta 2000] Pankaj Gupta. *Algorithms for routing lookups and packet classification*. Thèse de Doctorat, Stanford University, 2000. (Cité en page 35.)
- * **Note** : *Cette thèse constitue une revue complète des différents algorithmes de recherche pour classifier les flux.*
- [Han et coll., 2011] Sangjin Han, Keon Jang, Kyoungsoo Park et Sue Moon. *PacketShader : a GPU-accelerated software router*. ACM SIGCOMM Computer Communication Review, vol. 41, no. 4, pages 195–206, 2011. (Cité en pages 13, 14 et 16.)
- [Heminger 2015] Stephen Heminger. *DPDK performance How to not just do a demo with DPDK*. In FOSDEM 2015, 2015. (Cité en pages 43 et 44.)
- * **Note** : *Cette conférence expose les difficultés liées à l'utilisation de DPDK.*
- [Intel 2012] Intel. *Intel DPDK : Packet Processing on Intel Architecture*. In Presentation slides, 2012. (Cité en page 17.)
- [Intel 2014] Intel. *Intel® Data Plane Development Kit (Intel® DPDK) Programmer's Guide*. 2014. (Cité en pages 20 et 28.)
- [Klink et Wälde 2011] Alexander Klink et Julian Wälde. *Efficient Denial of Service Attack on Web Application Platforms*. In Chaos Communications Congress, 2011. **Note** : *Des méthodes peu communes pour effectuer un déni de service applicative* (Cité en page 5.)
- [Kumari et McPherson 2009] W. Kumari et D. McPherson. *Remote Triggered Black Hole Filtering with Unicast Reverse Path Forwarding (uRPF)*. RFC 5635, RFC Editor, August 2009. (Cité en page 7.)
- [Liao et coll., 2011] Guangdeng Liao, Xia Zhu et Laxmi Bnuyan. *A new server I/O architecture for high speed networks*. In High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on, pages 255–265. IEEE, 2011. **Note** : *Cet article mesure les goulets d'étranglements lors du traitement des paquets avec un outil de profilage*. (Cité en pages 13 et 14.)
- [Linux BPF] Linux BPF. *Linux Socket Filtering aka Berkeley Packet Filter (BPF)*. Documentation du noyau linux : <https://www.kernel.org/doc/Documentation/networking/filter.txt>. **Note** : *Page de documentant de la version de BPF implémentée dans le noyau Linux*. (Cité en page 32.)
- [Majkowski 2015] Marek Majkowski. *Cloudflare blog notes*, CloudFlare. Article de blog : <https://blog.cloudflare.com/author/marek-majkowski/>, 2015. (Cité en page 10.)
- * **Note** : *Séries d'articles qui introduisent de manière expérimentale les problèmes des piles réseau actuelles et les moyens d'y remédier. Premier article : https://blog.cloudflare.com/how-to-receive-a-million-packets/.*
- [McCanne et Jacobson 1993] Steven McCanne et Van Jacobson. *The BSD packet filter : A new architecture for user-level packet capture*. In Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings, pages 2–2. USENIX Association, 1993. (Cité en page 32.)
- [Mirkovic et Reiher 2004] Jelena Mirkovic et Peter Reiher. *A taxonomy of DDoS attack and DDoS defense mechanisms*. ACM SIGCOMM Computer Communication Review, vol. 34, no. 2, pages 39–53, 2004. (Cité en page 3.)
- [Open Resolver Project 2015] Open Resolver Project. <http://openresolverproject.org/>, Août 2015. (Cité en page 5.)
- [Prasad et coll., 2014] K Munivara Prasad, A Rama Mohan Reddy et K Venugopal Rao. *DoS and DDoS Attacks : Defense, Detection and Traceback Mechanisms-A Survey*. Global Journal of Computer Science and Technology, vol. 14, no. 7, 2014. (Cité en page 6.)

* **Note** : Cette conférence expose les coûts de traitement cachés du noyau Linux.

- [**Prince 2013**] Matthew Prince. *Lessons from Surviving 300 Gbps Denial of Service Attack*, CloudFlare. https://www.youtube.com/watch?v=w04ZAXftQ_Y, 2013. (Cité en pages 4 et 7.)
- [**R Core Team 2015**] R Core Team. *R : A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015. (Cité en page 25.)
- [**Rizzo 2012**] Luigi Rizzo. *netmap : A Novel Framework for Fast Packet I/O*. In USENIX Annual Technical Conference, pages 101–112, 2012. (Cité en pages 10 et 38.)
- [**Rossow 2014**] Christian Rossow. *Amplification hell : Revisiting network protocols for DDoS abuse*. In Symposium on Network and Distributed System Security (NDSS), 2014. (Cité en page 5.)
- * **Note** : Une étude expérimentale et exhaustive des vecteurs d'attaques DDoS par réflexion et amplification.
- [**Rostedt 2009**] Steven Rostedt. *Lockless Ring Buffer Desing*, Red Hat Inc. <http://lwn.net/Articles/340443/>, 2009. (Cité en pages 18 et 19.)
- [**Salim et coll., 2001**] Jamal Hadi Salim, Robert Olson et Alexey Kuznetsov. *Beyond softnet*. In Proceedings of the 5th annual Linux Showcase & Conference, volume 5, pages 18–18, 2001. (Cité en page 12.)
- [**Serodi 2013**] Leonardo Serodi. *Traffic Diversion Techniques for DDoS Mitigation using BGP Flowspec*. In Alcatel Lucent, 2013. (Cité en page 8.)
- [**Snijder 2014**] Job Snijder. *DDoS Damage Control Cheap & effective*. In RIPE 68, 2014. (Cité en page 7.)
- [**Srinivasan et coll., 1999**] Venkatachary Srinivasan, Subhash Suri et George Varghese. *Packet classification using tuple space search*. In ACM SIGCOMM Computer Communication Review, volume 29, pages 135–146. ACM, 1999. (Cité en page 42.)
- [**Suricata 2012**] Suricata. *Suricata, to 10Gbps and beyond*. Article de blog : <https://home.regit.org/2012/07/suricata-to-10gbps-and-beyond/>, 2012. (Cité en page 41.)

Statistiques

```
=====
(pid : 62508) conf 0 ----- Packets ----- Bytes
Sent: 318772 411058733
Received: 319208 411557925
Filtered: 436 499192
Dropped: 0 0
(pid : 62509) conf 1 ----- Packets ----- Bytes
Sent: 2416 3382032
Received: 2416 3382032
Filtered: 0 0
Dropped: 0 0
(pid : 62510) conf 2 ----- Packets ----- Bytes
Sent: 0 0
Received: 0 0
Filtered: 0 0
Dropped: 0 0
Aggregate statistics =====
Total sent: 321188 414440765
Total received: 321624 414939957
Total filtered: 436 499192
Total dropped: 0 0
=====

NIC statistics
=====
Port | Rx-packets | Rx-missed | Rx-bytes | Tx-packets | Tx-errors | Tx-bytes |
0: 321624 | 0 | 414939957 | 0 | 0 | 0 |
1: 0 | 0 | 0 | 321188 | 0 | 414440765 |
=====

Port | Rx-badcrc | Rx-badlen | Rx-errors | Rx-nombuf |
0: 0 | 0 | 0 | 0 |
1: 0 | 0 | 0 | 0 |
=====

Port | Fdirmatch | Fdirmis |
0: 0 | 0 |
1: 0 | 0 |
=====
```

Code iii.1 – Exemple de statistiques affichées

Graphique de l'application

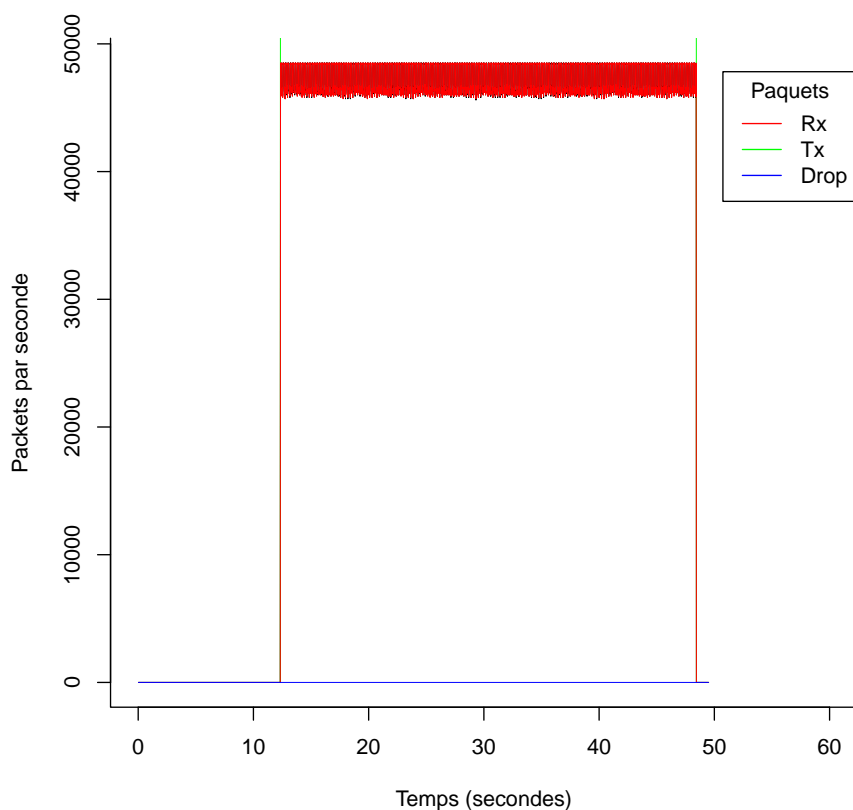


FIGURE 1. – *Exemple d'un graphique en R.*

Spécifications matérielles

Les SFP utilisés sont les mêmes pour toutes les machines : « Ethernet controller : Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network Connection (rev 01) ».

	Configuration 1	Configuration 2	Machine 1
CPU	E5-2620 2 GHz	X5650 2,67 GHz	i7-4470 3,4 GHz
Cache L1d	32 Ko	32 Ko	32 Ko
Cache L1i	32 Ko	32 Ko	32 Ko
Cache L2	256 Ko	256 Ko	256 Ko
Cache L3	15 360 Ko	12 288 Ko	8192 Ko
Memory channels	2	2	2
Carte Réseau	X520-DA2	X520-DA2	HSTNS-BN96

TABLE 1. – *Spécifications matérielles. Les configurations 1 et 2 correspondent aux machines exécutant le bridge.*

Résultats avec la configuration 2

<i>bridge</i> (Mpps)	testpmd (Mpps)	Transmis (Mpps)	Taille (octets)
11.5	14.030	14.88	64
11.2	5.75	14.705	65
9.6	5.801	13.02	80
6.184	5.857	8.680	128
4.595	4.595	4.595	256
2.352	2.352	2.352	512
1.202	1.202	1.202	1024
0.812	0.812	0.812	1518

TABLE 2. – *Débit de réception mesuré sur la machine 1. Une redirection de port est utilisée par la machine 2 avec la configuration matérielle 2.*

