UCLouvain

École polytechnique de Louvain

Modernisation et extension de TCPSnitch

Analyse comportementale multi-couches des applications

Auteur: **Matias Nieto Navarrete**
Promoteur: **Olivier Bonaventure**
Lecteurs: **Paul-Edouard Bertrand Van Ouytsel, Rémi Van Boxem**
Année académique 2025–2026
Master [60] en sciences informatiques

Résumé

Les applications interagissent avec la pile réseau via l'API Socket. Au fil des années, le contexte dans lequel cette API évolue a profondément changé. Conçu en 2017 pour analyser ces interactions, le logiciel *TCPSnitch* s'appuyait sur une interception en espace utilisateur. Aujourd'hui, cette méthode est limitée par l'émergence d'interfaces asynchrones, rendant le logiciel aveugle aux décisions internes du noyau. Pour restaurer une visibilité totale, ce mémoire propose une architecture hybride : une interception au niveau de l'application conserve le contexte ; des sondes intra-noyau extraient la dynamique TCP/IP et un moniteur système surveille la topologie physique. L'évaluation de cet outil sur des traces récentes (Android et Linux) révèle une évolution majeure des pratiques. Sur mobile, le trafic est massivement dominé par UDP sous l'impulsion de QUIC, et les applications réagissent de manière très inégale aux coupures réseau. À l'inverse, l'écosystème de bureau sous Linux s'appuie très efficacement sur le noyau pour absorber ces pannes. Enfin, l'étude souligne que cette observation « de l'intérieur » est désormais fortement entravée par les politiques de sécurité modernes. Le code source de la version modernisée est disponible en *open source* et l'intégralité du jeu de données en libre accès.

Remerciements

Je tiens tout d'abord à remercier le Professeur Olivier Bonaventure de m'avoir proposé ce sujet de mémoire et pour ses conseils tout au long de sa réalisation.

Je souhaite également remercier Paul-Edouard Bertrand Van Ouytsel pour son accompagnement, sa grande disponibilité et son aide tout au long de ce travail.

Enfin, je tiens à exprimer toute ma gratitude envers mes parents pour leur soutien et leurs encouragements lors de ma reprise d'études dans le cadre de ce Master en sciences informatiques. Leur confiance au quotidien a été essentielle pour me permettre de mener à bien ce projet.

Table des matières

Abréviations et symboles

ADB	Android Debug Bridge
API	Application Programming Interface
BPF	Berkeley Packet Filter
BSD	Berkeley Software Distribution
BTF	BPF Type Format
CDF	Cumulative Distribution Function
CLI	Command Line Interface
CO-RE	Compile Once - Run Everywhere
CQ	Completion Queue
CWND	Congestion Window
eBPF	Extended Berkeley Packet Filter
FD	File Descriptor
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
IPv4 / IPv6	Internet Protocol version 4 / version 6
JIT	Just-In-Time
JSON	JavaScript Object Notation
JSONL	JSON Lines
NDK	Native Development Kit (Android)
P2P	Peer-to-Peer
QUIC	Quick UDP Internet Connections
RFC	Request for Comments
RSS	Resident Set Size
RTO	Retransmission Timeout
RTT	Round-Trip Time
SELinux	Security-Enhanced Linux
SQ	Submission Queue
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UID	User Identifier

Chapitre 1

Introduction

Depuis des années, l'API Socket constitue l'interface standard par laquelle les applications interagissent avec la pile TCP/IP. Bien que cette interface ait été conçue à l'origine pour des environnements relativement statiques, elle est aujourd'hui au cœur d'architectures logicielles modernes et de terminaux mobiles qui basculent fréquemment entre différents réseaux. Face à cette évolution, comprendre comment les applications modernes exploitent réellement le réseau est devenu un défi majeur.

En 2017, Grégory Vander Schueren a posé une première étape importante en développant TCPSnitch [1], un outil capable d'intercepter les appels à l'API Socket en espace utilisateur via la variable d'environnement `LD_PRELOAD`. L'analyse d'un volume massif d'appels avait alors révélé un écart significatif entre la théorie enseignée dans la littérature et l'usage empirique des sockets (connexions inactives, appels redondants, micro-tampons).

Cependant, près d'une décennie plus tard, l'écosystème logiciel et le noyau Linux ont radicalement changé. Les applications modernes cherchent à minimiser les copies mémoire et les transitions entre l'espace utilisateur et le noyau, notamment grâce à de nouvelles interfaces d'E/S asynchrones comme `io_uring` [2]. L'approche purement confinée à l'espace utilisateur de 2017 est désormais insuffisante, la rendant souvent aveugle à la réalité physique du trafic.

C'est dans ce contexte que s'inscrit ce mémoire. L'objectif principal de ce travail est de mettre à jour et d'évaluer TCPSnitch [3]. Pour y parvenir, notre mission s'articule autour de quatre axes majeurs :

1. **Moderniser l'interception utilisateur :** Mettre à jour l'outil pour supporter les appels système les plus récents de Linux.
2. **Capturer la topologie réseau (`Netlink`) :** Ajouter un moniteur asynchrone capable d'intercepter les messages de routage du noyau via le protocole `RT_NETLINK` [4]. L'objectif est de lier le comportement de l'application aux changements d'adresses IP et d'interfaces.
3. **Infiltrer l'espace noyau (`eBPF`) :** Intégrer la technologie *Extended Berkeley Packet Filter* [5] pour exécuter des sondes de manière sécurisée directement dans le noyau. Cette innovation offre une visibilité sur la dynamique interne de la pile TCP/IP (retransmissions, congestion, complétions asynchrones).
4. **Évaluer et comparer les données :** Valider la pertinence de cette nouvelle architecture et comparer les modèles d'utilisation de l'API Socket d'aujourd'hui avec les données récoltées en 2017 [1].

Le présent document est structuré de la manière suivante :

- Le **Chapitre 2** dresse l'état de l'art en analysant l'évolution de l'API Socket, les limites actuelles de l'approche `LD_PRELOAD`, et les technologies fondatrices de notre solution.
- Le **Chapitre 3** détaille l'architecture et l'implémentation de la mise à jour de TCPSnitch. Il explique comment ces diverses technologies ont été combinées pour construire une plateforme d'observabilité hybride.
- Le **Chapitre 4** est consacré à la validation expérimentale et à l'évaluation des performances de TCPSnitch, tout en délimitant son périmètre d'efficacité.
- Le **Chapitre 5** décrit la méthodologie rigoureuse employée pour constituer notre jeu de données.
- Le **Chapitre 6** plonge au cœur de l'analyse des traces récoltées. En confrontant nos nouvelles données à l'étude de 2017, il met en lumière l'évolution des protocoles et démontre l'apport indispensable de notre nouvelle architecture noyau pour comprendre le comportement des applications sous de fortes contraintes réseau.
- Le **Chapitre 7** clôture ce mémoire par une discussion globale, en dressant le bilan de notre plateforme multicouche et en ouvrant de nouvelles perspectives pour l'avenir de l'observabilité réseau.

Chapitre 2

État de l'art et Contexte

L'observation et l'analyse du trafic réseau au niveau des applications sont essentielles pour comprendre le comportement des logiciels modernes, diagnostiquer les problèmes de performance et identifier les anomalies de sécurité. Dans son mémoire, Vander Schueren [1] a mis en évidence la valeur d'une approche d'analyse dynamique pour capturer les interactions entre les applications et la pile TCP/IP. L'outil résultant de ce travail, TCPSnitch [3], reposait sur l'interception des appels à la bibliothèque C standard via la variable d'environnement `LD_PRELOAD`.

Bien que cette méthode ait démontré son efficacité sur une grande variété d'applications, l'évolution des pratiques de développement et de l'architecture du noyau Linux met en lumière certaines de ses limites. Dès 2017, la technologie eBPF (*Extended Berkeley Packet Filter*) était identifiée par Vander Schueren comme une piste prometteuse, opérant directement dans le noyau. Les progrès considérables réalisés depuis l'introduction du *BPF Type Format* (BTF) [6], de la portabilité CO-RE [7] et du *ring buffer* [8] en font aujourd'hui un mécanisme mature et complémentaire à l'approche `LD_PRELOAD`.

Ce chapitre pose les bases théoriques qui motivent et justifient la modernisation et l'extension de TCPSnitch. Après avoir passé en revue l'évolution de l'API Socket sous Linux et l'émergence de nouveaux protocoles, nous comparerons les méthodes d'observation traditionnelles en espace utilisateur avec les solutions modernes opérant directement dans le noyau, en particulier eBPF [5]. Pour terminer, nous présenterons le protocole Netlink [4] et son rôle dans le suivi de la topologie réseau.

2.1 L'API Socket sous Linux : Des appels classiques aux évolutions récentes

L'API Socket constitue l'interface standard par laquelle les applications interagissent avec la pile TCP/IP. Introduite historiquement dans les systèmes BSD (*Berkeley Software Distribution*), elle s'appuie sur la philosophie Unix selon laquelle « tout est fichier », représentant ainsi chaque point de communication réseau par un descripteur de fichier.

2.1.1 Historique et principes de base

Le cycle de vie et l'exploitation d'un socket s'articulent traditionnellement autour d'appels systèmes bien définis. Historiquement, l'observation du réseau se concentrait sur une surface d'API restreinte. Cependant, pour refléter fidèlement les interactions modernes, l'analyse théorique se doit d'englober les trois piliers fondamentaux de la communication réseau.

Le premier pilier concerne la gestion du cycle de vie des connexions, allant de leur initialisation via des fonctions incontournables comme `socket()` et `connect()` jusqu'à leur clôture avec `close()`. Le deuxième pilier couvre le transfert effectif des octets. Au-delà des appels classiques tels que `read()` ou `send()`, l'API s'est enrichie de fonctions capables de traiter plusieurs messages en une seule fois comme `sendmmsg()`, permettant ainsi de réduire significativement le nombre d'appels systèmes. Enfin, le dernier pilier englobe la configuration matérielle et logicielle des descripteurs, rendue possible par des outils de manipulation standards comme `fcntl()` ou `ioctl()`. L'intégrité du suivi doit par ailleurs être assurée lors de la création de processus enfants (`fork()`) afin de maintenir le contexte de la connexion.

Toutefois, pour gérer de multiples connexions simultanément sans bloquer le processus principal, l'API s'est dotée de mécanismes de multiplexage. Si des interfaces historiques comme `select()` ont posé les bases, la famille `epoll()` est aujourd'hui privilégiée sous Linux pour garantir le passage à l'échelle. Néanmoins, quelle que soit l'interface de multiplexage utilisée, toutes s'exécutent exclusivement en espace utilisateur : elles ne capturent que les interactions explicitement initiées par l'application, laissant dans l'ombre les événements internes au noyau tels que les retransmissions TCP ou les abandons de paquets.

2.1.2 L'intégration d'appels systèmes récents et d'interfaces asynchrones

Face aux goulots d'étranglement propres à la pile réseau traditionnelle, le noyau Linux s'est enrichi de nouveaux appels systèmes et de techniques d'optimisation. Pour rester pertinents, les outils d'observabilité modernes se doivent de supporter ces évolutions qui contournent souvent les méthodes d'interception classiques :

La fermeture optimisée avec `close_range()` : Introduit dans la version 5.9 du noyau Linux [9], cet appel système permet de fermer efficacement toute une plage de descrip-

teurs de fichiers en un seul appel système. Historiquement, un processus cloné devait effectuer une boucle et appeler `close()` de manière individuelle. L'introduction de cette fermeture groupée complique considérablement le maintien de l'état des connexions pour les outils d'observabilité opérant en espace utilisateur.

Le transfert « Zero-Copy » : D'autres mécanismes plus anciens, tels que `splice()` [10] ou `sendfile()` [11], permettent également de contourner l'espace utilisateur en transférant des données directement entre deux descripteurs au sein de l'espace noyau, limitant ainsi la consommation mémoire.

L'interface asynchrone `io_uring` : Introduite par Jens Axboe [2], cette API propose une approche plus efficace de la gestion des entrées/sorties sous Linux. Contrairement au modèle classique nécessitant un appel système pour chaque opération, `io_uring` s'appuie sur des files d'attente circulaires (*ring buffers*) partagées directement entre l'espace utilisateur et le noyau via `mmap()`, réduisant ainsi le coût des transitions entre ces deux espaces. L'application dépose ses requêtes d'entrée/sortie dans une file de soumission (*Submission Queue*, SQ), tandis que le noyau publie les résultats dans une file de complétion (*Completion Queue*, CQ).

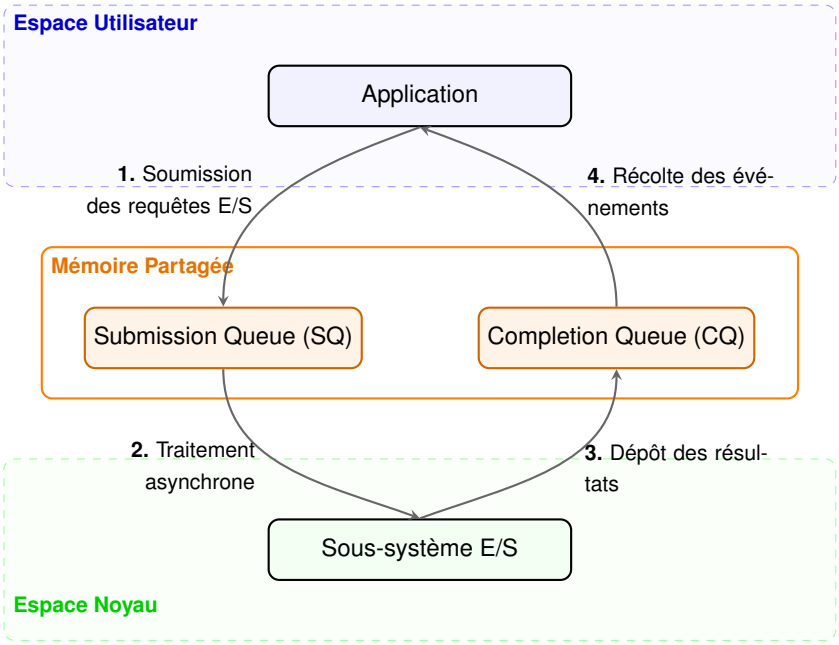

FIGURE 2.1 – Architecture de l'interface `io_uring`. L'application et le noyau s'échangent des requêtes via une zone de mémoire partagée contenant deux files d'attente (SQ et CQ). Ce modèle supprime le besoin d'effectuer un appel système coûteux pour chaque opération d'entrée/sortie réseau.

L'utilisation croissante de ces mécanismes avancés, et tout particulièrement l'adoption de l'architecture `io_uring`, rend obsolètes les outils d'interception purement statiques basés sur la bibliothèque C. Puisque les transferts de données ne déclenchent plus systématiquement d'appels `read()` ou `write()` standards, l'interception en espace utilisateur seule est désormais aveugle face à une part grandissante du trafic des applications modernes.

2.1.3 L'émergence de QUIC : un défi pour l'observation noyau

Le passage au protocole QUIC (HTTP/3) [12] déplace la gestion de la fiabilité et de la congestion du noyau vers l'espace utilisateur. En s'appuyant sur UDP, il rend les outils traditionnels centrés sur la pile TCP structurellement « aveugles ». Ce cloisonnement entre la logique applicative et le transport réseau impose désormais une approche capable d'observer simultanément différentes couches du système.

2.2 L'observation en espace utilisateur : fonctionnement et limites de LD_PRELOAD

La première version de TCPSnitch [3] reposait sur un mécanisme fondamental des systèmes Unix : la variable d'environnement `LD_PRELOAD`. Cette section explique son fonctionnement, puis en identifie les limites face aux architectures logicielles modernes.

2.2.1 Le mécanisme d'interception

Lorsqu'une application est lancée, l'éditeur de liens dynamiques (`ld.so`) charge en mémoire les bibliothèques partagées dont elle dépend, en particulier la bibliothèque C standard (`libc`), qui implémente les fonctions réseau comme `connect()` ou `send()`.

La variable `LD_PRELOAD` permet d'insérer une bibliothèque personnalisée *avant* la `libc` dans cet ordre de chargement. Toute fonction définie dans cette bibliothèque prend alors la priorité sur son équivalent dans la `libc`. C'est ce mécanisme, appelé *hooking*, que TCPSnitch exploite.

Concrètement, lorsqu'une application appelle `connect()`, c'est la version de TCPSnitch qui s'exécute en premier. Elle extrait les arguments, enregistre un horodatage et collecte les métadonnées utiles (identifiant du thread, valeur de retour, errno). Elle appelle ensuite la vraie fonction `libc` via `dlsym(RTLD_NEXT)`, qui transmet l'appel au noyau. À son retour, le résultat est capturé et formaté en JSON, comme l'illustre la Figure 2.2.

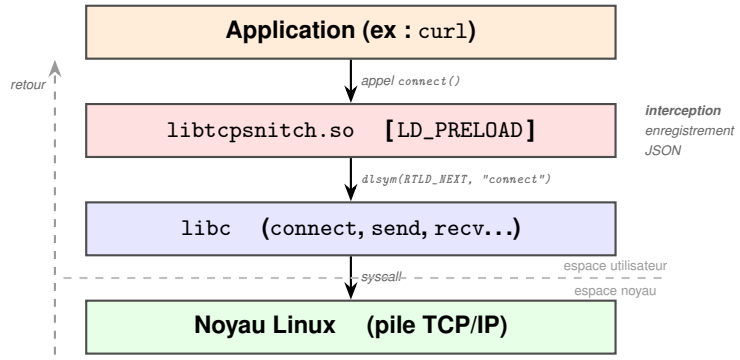

FIGURE 2.2 – Mécanisme d'interception par LD_PRELOAD.

Ce mécanisme présente plusieurs avantages : il est transparent pour l'application, ne nécessite pas de privilèges particuliers et offre un accès complet aux métadonnées de l'espace utilisateur (arguments, threads).

2.2.2 Les limites face aux architectures modernes

Malgré son efficacité, le traçage par LD_PRELOAD présente plusieurs limites importantes.

Visibilité partielle des événements noyau

L'API Socket en espace utilisateur n'offre qu'une vue de haut niveau de la communication. Lorsqu'une application appelle `send()`, la `libc` copie les données dans les tampons du noyau et s'arrête là.

TCPSnitch contourne partiellement cette limite en interrogeant périodiquement l'option socket `TCP_INFO` via `getsockopt()`. Cette option expose des compteurs internes de la pile TCP : le RTT lissé, la fenêtre de congestion (`snd_cwnd`), le nombre de retransmissions ou encore les octets acquittés. Ces *snapshots* sont capturés à intervalles réguliers et apparaissent dans les traces JSON.

Cette méthode offre une vision indirecte et fragmentée de la pile réseau. Elle ne permet pas de saisir l'instant précis d'une retransmission, ni de détecter des événements critiques comme la perte de paquets (*drops*).

L'opacité des applications et des environnements isolés

Le mécanisme de LD_PRELOAD montre ses limites avec les navigateurs web actuels sous Linux et les applications haute performance. Ce problème est particulièrement visible dans deux situations :

1. **Piles réseau personnalisées :** Des projets comme Chromium (Chrome, Electron) n'utilisent plus exclusivement les mécanismes standards du noyau pour certains protocoles. En implémentant leur propre gestion des flux (comme évoqué à la section précédente avec l'adoption de QUIC/HTTP3), ils contournent les fonctions TCP classiques de la `libc`. TCPSnitch perd alors toute visibilité sur la logique de congestion ou de retransmission, gérée en interne par l'application.
2. **Mécanismes de *sandboxing* :** Des navigateurs tels que Firefox isolent leurs processus de rendu dans des environnements restreints via des *namespaces* et des politiques `seccomp` [13]. Cette isolation compromet TCPSnitch de deux manières : elle bloque souvent l'héritage de la variable `LD_PRELOAD` vers les processus enfants et même en cas de succès, elle restreint les capacités de TCPSnitch à effectuer ses propres opérations (écriture des traces, accès aux métadonnées).

Dans ces scénarios, l'outil ne se contente plus d'observer l'application, il subit les mêmes restrictions de sécurité qu'elle, rendant l'observation partielle, voire impossible.

Restrictions de sécurité sur Android

Sur Android, `LD_PRELOAD` est bloqué par défaut par les politiques SELinux. Pour contourner ces restrictions, l'observation nécessite d'utiliser les propriétés d'encapsulation (*wrap properties*) d'Android, ce qui requiert un accès *root* au dispositif, une contrainte technique non négligeable en pratique.

2.3 L'observation en espace noyau : eBPF

La section précédente a montré que l'approche `LD_PRELOAD` ne peut pas capturer les événements internes à la pile TCP/IP. Pour franchir cette frontière et obtenir une visibilité complète, il faut exécuter du code directement dans le noyau. C'est précisément ce que permet la technologie eBPF [14].

2.3.1 Des origines de BPF à eBPF

BPF a été introduit historiquement par McCanne et Jacobson [15] comme un mécanisme de filtrage de paquets réseau au niveau du noyau. Dans sa forme originale, il s'agissait d'une machine virtuelle simple, conçue pour évaluer des filtres de paquets de manière sécurisée, sans risquer de déstabiliser le système d'exploitation.

L'eBPF est une évolution majeure de cette technologie, qui s'étend désormais bien au-delà du simple filtrage réseau. Elle permet d'attacher des programmes à divers points d'ancrage du noyau (*hooks*) tels que les sockets, les appels systèmes ou les sondes `kprobe`. L'eBPF devient ainsi un mécanisme générique d'extension du noyau et dépasse largement le cadre de l'observabilité, étant aujourd'hui massivement utilisé pour optimiser les performances réseau ou la sécurité [16].

2.3.2 Principe de fonctionnement

Un programme eBPF suit un cycle de vie logique en quatre grandes étapes (compilation, vérification, compilation à la volée et attachement), faisant intervenir les différents composants illustrés à la Figure 2.3.

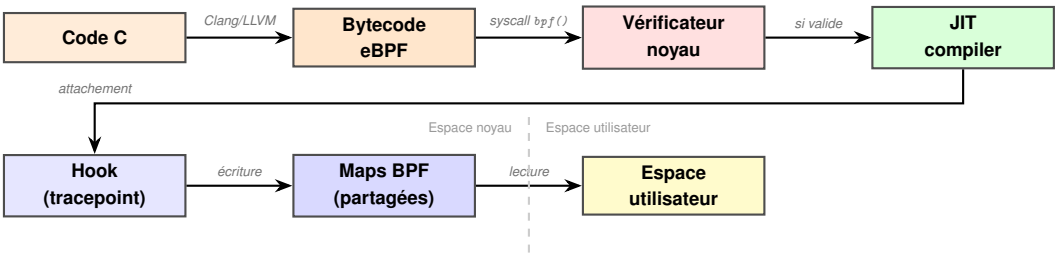

FIGURE 2.3 – Cycle de vie d'un programme eBPF.

Le programme est tout d'abord compilé en *bytecode* portable par un compilateur tel que Clang. Ce *bytecode* est ensuite soumis au *vérificateur*, un analyseur statique rigoureux qui garantit que le programme n'accédera à aucune zone mémoire hors limites et valide sa terminaison historique ou l'usage de boucles sécurisées [17].

Si le programme est valide, il est compilé en code machine natif via un compilateur JIT (*Just-In-Time*) et attaché à son *hook*, c'est-à-dire l'événement précis du système qu'il doit surveiller. Lors de son exécution, les résultats récoltés sont écrits dans des *Maps BPF*, des structures de données (comme des tables de hachage ou des tableaux) partagées et accessibles en lecture depuis l'espace utilisateur [18].

Cette architecture offre deux propriétés fondamentales : la **sécurité** (le vérificateur empêche tout comportement instable) et la **performance** (le code s'exécute directement dans le noyau, sans changement de contexte).

2.3.3 Complémentarité avec LD_PRELOAD

L'eBPF et `LD_PRELOAD` sont complémentaires plutôt que concurrents. `LD_PRELOAD` intercepte les appels à la `libc` et donne accès au contexte applicatif, mais reste aveugle face aux sous-systèmes modernes comme `io_uring`. L'eBPF comble cet angle mort en capturant les événements internes au noyau (retransmissions TCP, évolution de la fenêtre de congestion, abandons de paquets), invisibles depuis l'espace utilisateur.

2.4 La surveillance de la topologie réseau : Netlink

L'API Socket (en espace utilisateur) et eBPF (en espace noyau) permettent d'observer les échanges de données entre une application et la pile TCP/IP. Mais le comportement réseau

d'une application dépend aussi de son environnement : un changement d'interface, une nouvelle adresse IP ou une modification de routage peuvent directement influencer les connexions en cours. Pour capturer cette dimension, l'observabilité réseau moderne s'appuie sur la surveillance du protocole Netlink.

2.4.1 Qu'est-ce que Netlink ?

Netlink est un mécanisme de communication entre l'espace utilisateur et le noyau Linux, formalisé par la RFC 3549 [4]. Contrairement à l'API Socket classique, qui gère des flux de données applicatives, Netlink est dédié exclusivement à la configuration et à la surveillance de la pile réseau elle-même : gestion des interfaces, des adresses IP et des règles de routage.

Une application peut ouvrir un socket de la famille `AF_NETLINK` et s'abonner à des groupes de notifications multicast. Le noyau envoie alors des messages structurés à chaque changement d'état du réseau : `RTM_NEWADDR` et `RTM_DELADDR` signalent l'ajout ou la suppression d'une adresse IP, tandis que `RTM_NEWROUTE` et `RTM_DELROUTE` informent des modifications de la table de routage.

2.4.2 L'utilité pour l'observation réseau

L'intégration de Netlink permet de contextualiser les données de trafic. En croisant l'activité de l'application avec les évolutions de l'environnement physique, il devient possible d'expliquer des comportements réseau qui, autrement, sembleraient totalement arbitraires.

Corréler topologie réseau et observabilité système

Sur mobiles, une transition d'un réseau Wi-Fi vers la 4G (Handover) ou une perte soudaine de signal peuvent expliquer une série de retransmissions TCP. Sur Linux, l'apparition d'une nouvelle route ou la désactivation d'une interface peuvent justifier une interruption de connexion.

Par conséquent, l'observabilité moderne ne peut plus se limiter aux données de l'application ou du noyau. Elle doit y associer les évolutions de la topologie réseau pour donner un véritable contexte aux traces récoltées.

2.5 Conclusion

Ce chapitre a démontré qu'une approche d'observation purement confinée à l'espace utilisateur via `LD_PRELOAD` est désormais insuffisante. Aveugle aux événements profonds du noyau et bloquée par les architectures logicielles modernes (piles réseau personnalisées, environ-

nements isolés), elle nécessite d'être complétée par des technologies d'observation plus profondes.

Si des outils comme *bpftrace* [19] surveillent le noyau, leur approche reste trop globale. À l'inverse, le *TCPSnitch* d'origine (2017) se limitait uniquement à l'espace utilisateur. Nous avons donc entièrement repensé cet outil. Par la suite, le nom *TCPSnitch* désignera cette nouvelle version, capable de relier directement les appels de l'application aux événements internes du noyau.

Pour pallier les manques des outils actuels, TCPSnitch repose sur une architecture hybride combinant `LD_PRELOAD` pour le contexte applicatif, `eBPF` pour l'observation du noyau et `Netlink` pour la topologie réseau. Cette alliance constitue le cœur de notre projet, dont l'architecture et l'implémentation sont détaillées dans le chapitre suivant.

Chapitre 3

Architecture et Implémentation de TCPSnitch

Le chapitre précédent a mis en évidence la nécessité d'une approche d'observabilité réseau hybride, capable de lier le contexte applicatif, les événements profonds du noyau et les changements de topologie physique. Ce chapitre détaille la conception technique et l'implémentation de TCPSnitch, la nouvelle version de l'outil initialement développé par Vander Schueren en 2017 [3].

Nous commencerons par une brève revue de l'architecture héritée de 2017 et de son format de sortie. Nous détaillerons ensuite la modernisation de la couche d'interception en espace utilisateur via `LD_PRELOAD`, avant de présenter l'intégration de la surveillance environnementale via Netlink. La suite sera consacrée au cœur de cette refonte : l'implémentation de la couche eBPF et le défi technique de sa corrélation avec l'espace utilisateur. Enfin, nous présenterons le nouvel outil d'analyse visuelle accompagnant cette architecture.

3.1 Revue de l'architecture héritée (2017) et de son intégration JSON

Pour comprendre le nouveau TCPSnitch, il faut d'abord se pencher sur sa version originale de 2017. Conçu pour cibler Linux et Android, l'outil se basait uniquement sur une bibliothèque dynamique en C (`libtcpsnitch.so`).

En utilisant la variable `LD_PRELOAD` (ou les *wrap properties* d'Android), cette bibliothèque s'insérait dans l'application cible dès son démarrage. Elle agissait alors comme un pont trans-

parent, capable d'intercepter environ quarante fonctions réseau de la `libc`.

Le traitement de chaque appel système intercepté (tels que `connect()` ou `close()`) suivait une séquence stricte, illustrée par la Figure 3.1 : (1) l'extraction du contexte applicatif, incluant l'horodatage, le thread, les arguments et le descripteur de fichier ; (2) l'exécution transparente de la fonction originale de la `libc` grâce à `dlsym(RTLD_NEXT)` et (3) la capture du code de retour et de la variable `errno`, suivie de leur sérialisation immédiate au format JSON.

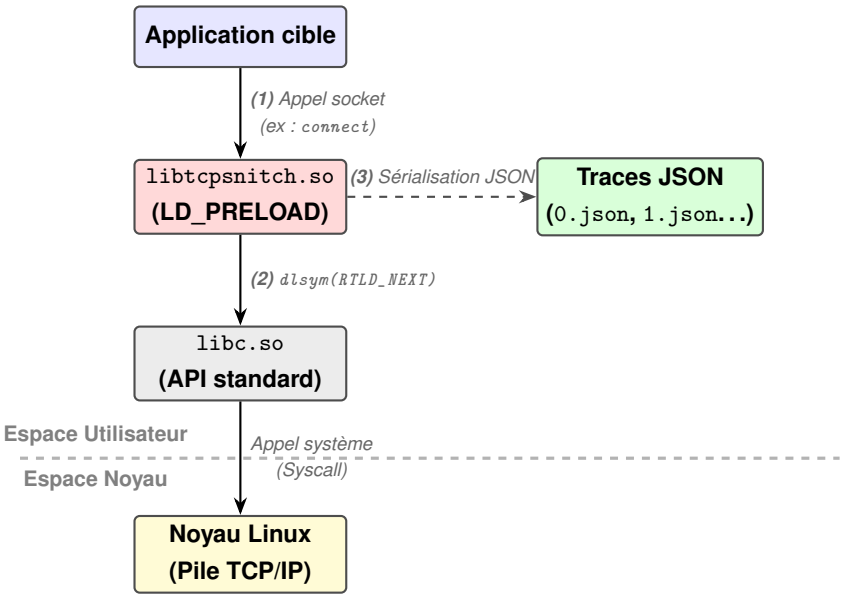

FIGURE 3.1 – Architecture de TCPSnitch (2017).

Les événements interceptés étaient ainsi consignés ligne par ligne dans un fichier de trace structuré. À titre d'exemple, le listing 3.1 illustre la sortie générée par l'outil de 2017 lors de l'interception d'un appel à `connect()`.

Listing 3.1 – Représentation JSON d'un appel connect() par TCPSnitch en 2017.

```
{
  "type": "connect",
  "timestamp_usec": 1491043720731853,
  "return_value": 0,
  "success": true,
  "thread_id": 17313,
  "details": {
    "addr": {
      "sa_family": "AF_INET",
      "ip": "127.0.1.1",
      "port": "53"
    }
  }
}
```

Ce format structuré remplit parfaitement son rôle pour extraire le contexte applicatif, tel que

le `thread_id`. Plutôt que d'alourdir cette structure éprouvée, notre refonte maintient cette base pour l'espace utilisateur, et déploie de nouveaux flux de données parallèles pour observer les couches profondes du système.

3.2 Modernisation de la couche LD_PRELOAD : L'implémentation des nouveaux appels

Afin de prendre en charge les évolutions de l'API Socket détaillées au chapitre 2.1.2, nous avons enrichi la bibliothèque d'interception (`libc_overrides.c`). L'objectif de cette modernisation est d'intégrer ces appels récents tout en préservant la stabilité et les performances de l'application surveillée. Cette section détaille l'implémentation spécifique de `splice()` et `close_range()`.

3.2.1 Filtrage dynamique du *Zero-Copy*

L'appel `splice()` permet de transférer des octets directement entre des descripteurs au sein du noyau (*Zero-Copy*). Cet appel étant générique (fichiers locaux, *pipes*, sockets), notre redéfinition doit impérativement filtrer les événements pour ne tracer que le trafic réseau. La logique d'interception se résume en trois étapes séquentielles :

1. **Exécution normale :** L'outil laisse d'abord le noyau réaliser le transfert des données et sauvegarde simplement le résultat.
2. **Vérification :** Il contrôle ensuite si la source (`fd_in`) ou la destination (`fd_out`) correspond à un socket réseau (via la fonction `is_inet_socket()`).
3. **Journalisation :** Si le transfert concerne bien le réseau, l'événement est enregistré. L'outil rend alors immédiatement la main à l'application.

Cette approche garantit que les opérations locales (comme la copie d'un fichier sur le disque) ne polluent pas notre jeu de données réseau.

3.2.2 Suivi d'état et gestion des fermetures de masse

Bien que l'appel `close_range()` [20] optimise le nettoyage des descripteurs par le noyau, son interception pose deux défis majeurs à TCPSnitch : la préservation des performances et la persistance de l'historique des connexions.

D'abord, pour empêcher qu'une application ne fige l'outil en demandant la fermeture d'une plage allant jusqu'à la valeur maximale (`UINT_MAX`), nous appliquons un garde-fou. L'analyse est systématiquement plafonnée aux 4096 premiers descripteurs, ce qui correspond à la limite matérielle standard (`RLIMIT_NOFILE`) par processus sous Linux [21].

Le second défi réside dans la perte irrémédiable d'informations induite par cet appel. Une fois la fonction exécutée, le noyau détruit instantanément les structures associées aux descripteurs, interdisant toute vérification pour identifier s'il s'agissait de sockets réseau. Pour contourner cette contrainte, TCPSnitch implémente une stratégie classique de capture spéculative d'état (*State Caching*), dont le fonctionnement est illustré par la Figure 3.2.

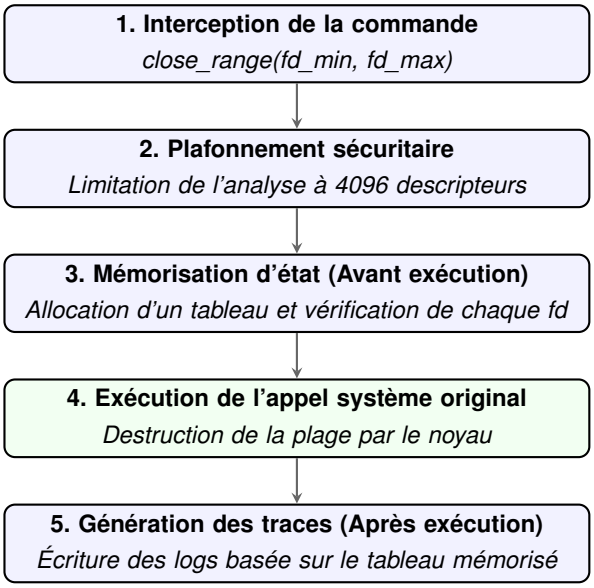

FIGURE 3.2 – Stratégie d'interception spéculative (*State Caching*) pour l'appel `close_range`. L'état des descripteurs doit impérativement être photographié avant l'action destructrice du noyau.

Bien que cette approche soit la solution idéale en théorie, elle se heurte au fonctionnement du noyau Linux en conditions réelles. Puisque le système distribue les descripteurs un par un, les fichiers de l'application risquent fortement de s'entremêler avec les fichiers journaux générés par TCPSnitch. L'impact critique de cet entrelacement sur la viabilité de l'outil sous forte charge sera évalué et démontré dans le chapitre 4.5 lors de la validation.

3.3 Implémentation du moniteur Netlink : Capture de la topologie

L'analyse purement applicative des sockets atteint rapidement ses limites face aux environnements réseaux dynamiques, particulièrement sur les terminaux mobiles (transitions Wi-Fi/4G, pertes de signal). Afin d'enrichir les événements interceptés par `LD_PRELOAD` avec un véritable contexte environnemental, nous avons développé un moniteur dédié à l'écoute du protocole Netlink (`netlink_spy.c`).

3.3.1 Fonctionnement asynchrone et filtrage Multicast

L'objectif principal de ce composant est de ne jamais introduire de latence dans l'application tracée. Pour cela, le moniteur est lancé dans un thread POSIX séparé au démarrage de l'outil, fonctionnant en parallèle du fil d'exécution principal, comme l'illustre la Figure 3.3.

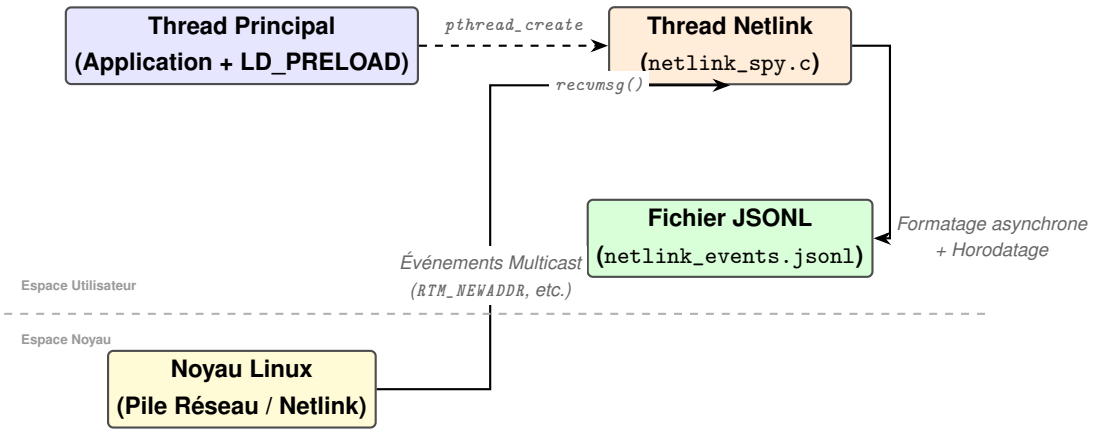

FIGURE 3.3 – Architecture asynchrone du moniteur Netlink.

Ce thread Netlink ouvre un socket brut dédié à la famille de routage (`AF_NETLINK`). Le noyau Linux étant extrêmement bavard, écouter l'intégralité du trafic Netlink surchargerait inutilement notre moniteur. Afin d'isoler uniquement les changements de topologie et d'ignorer le reste du bruit noyau, notre implémentation s'abonne explicitement à un masque de groupes de diffusion (*multicast*) très précis. Le moniteur ne s'intéresse qu'à quatre catégories d'événements : l'ajout ou la suppression d'adresses IP (sur IPv4 et IPv6) ainsi que les modifications de la table de routage. Une fois cette configuration appliquée, le thread entre dans une boucle infinie d'écoute passive.

3.3.2 Construction de l'état initial

L'écoute passive présente un biais majeur : elle ne capture que les *modifications* survenant après le lancement de l'application. Or, pour que l'analyse post-capture soit cohérente, l'outil doit impérativement connaître la topologie réseau existante dès l'instant initial ($t = 0$).

Pour combler ce vide, le moniteur exécute une fonction de balayage (*dump_initial_interfaces*) juste avant d'entrer dans sa boucle d'écoute. En interrogeant le système via les API standards (`getifaddrs`), l'outil liste l'intégralité des interfaces actives et de leurs adresses. À partir de ces données, le moniteur synthétise manuellement des événements de création d'adresses (`RTM_NEWADDR`) et les injecte dans le journal. Ce mécanisme garantit qu'une photographie complète de la connectivité est enregistrée dans les traces avant même que l'application ne crée son premier socket.

3.3.3 Extraction, horodatage et journalisation

Lorsqu'un véritable message Netlink est intercepté par la boucle d'écoute, sa charge utile brute doit être décodée (parsée). Le noyau compacte ces données sous forme d'attributs de routage séquentiels.

La logique de traitement s'articule autour de deux vérifications :

- **Les événements d'interfaces (**`RTM_NEWADDR` / `RTM_DELADDR`**) :** L'outil extrait le nom de l'interface concernée (ex. `wlan0`) et l'adresse IP qui vient de lui être assignée ou retirée.
- **Les événements de routage (**`RTM_NEWROUTE` / `RTM_DELROUTE`**) :** L'outil isole la modification de la passerelle par défaut ou les changements de routes spécifiques affectant la connectivité globale.

Pour corréler parfaitement ces changements environnementaux avec les appels de la bibliothèque C (interceptés par `LD_PRELOAD`), il est crucial de garantir une synchronisation temporelle stricte. Juste avant son formatage, chaque événement Netlink est horodaté à la microseconde près en utilisant exactement la même horloge système (`CLOCK_REALTIME`) que le reste de l'application.

Ces événements sont enregistrés dans un fichier dédié (`netlink_events.jsonl`). Cela permet d'isoler l'activité de l'application des changements de la topologie réseau, facilitant ainsi l'analyse post-capture.

3.4 Conception et implémentation de la couche eBPF

Après avoir établi les fondements théoriques de l'eBPF (voir chapitre 2.3), cette section détaille son intégration concrète au sein de TCPSnitch. L'enjeu n'est plus ici de justifier l'usage du noyau, mais de décrire les choix d'implémentation qui permettent de transformer cette technologie en un collecteur de données fiable.

3.4.1 Interaction entre l'espace utilisateur et le noyau

Pour comprendre l'architecture de TCPSnitch, il est nécessaire de distinguer deux phases clés : la **mise en place** (initialisation) et la **surveillance** (exécution en temps réel).

Lors du démarrage (fonction `init_tcpsnitch()` dans le fichier `init.c`), la bibliothèque `LD_PRELOAD` agit comme le processus pilote. Elle est responsable de l'initialisation du système : elle demande au noyau de charger les programmes eBPF et les attache à ses points d'ancrage (*tracepoints*).

Une fois cette étape terminée, les sondes eBPF deviennent autonomes. Elles surveillent le trafic de manière globale au sein du noyau sans que la bibliothèque `LD_PRELOAD` n'ait besoin

d'intervenir. Les deux composants de l'outil s'exécutent alors en parallèle et communiquent exclusivement via des structures de mémoire partagée appelées *BPF Maps* [18].

Cette séparation des rôles en phase de capture, illustrée par la Figure 3.4, s'articule autour de trois acteurs fondamentaux :

1. **L'application (via LD_PRELOAD) :** Elle fournit le contexte sémantique. Lorsqu'une connexion s'ouvre (par exemple via l'appel `connect()`), la bibliothèque intercepte l'action. Elle récupère les informations de liaison (descripteur de fichier et port source alloué) et les enregistre immédiatement dans la *BPF Map* [18].
2. **Le noyau (via eBPF) :** Il observe le trafic physique. Les sondes agissent de manière événementielle : lorsqu'un incident réseau survient (comme une perte de paquet déclenchant le *tracepoint* `tcp_retransmit_skb`), le noyau exécute la sonde correspondante. Celle-ci extrait le port source du paquet fautif et consulte la *BPF Map* pour vérifier s'il appartient à l'application surveillée.
3. **La collecte (via le *Ring Buffer*) :** Il s'agit du canal d'exfiltration. Si l'événement concerne l'application, la sonde noyau extrait les métriques et les insère dans un tampon circulaire haute performance (*Ring Buffer*). En parallèle, dans l'espace utilisateur, un *thread* asynchrone lit ces données en continu pour les sérialiser au format JSONL.

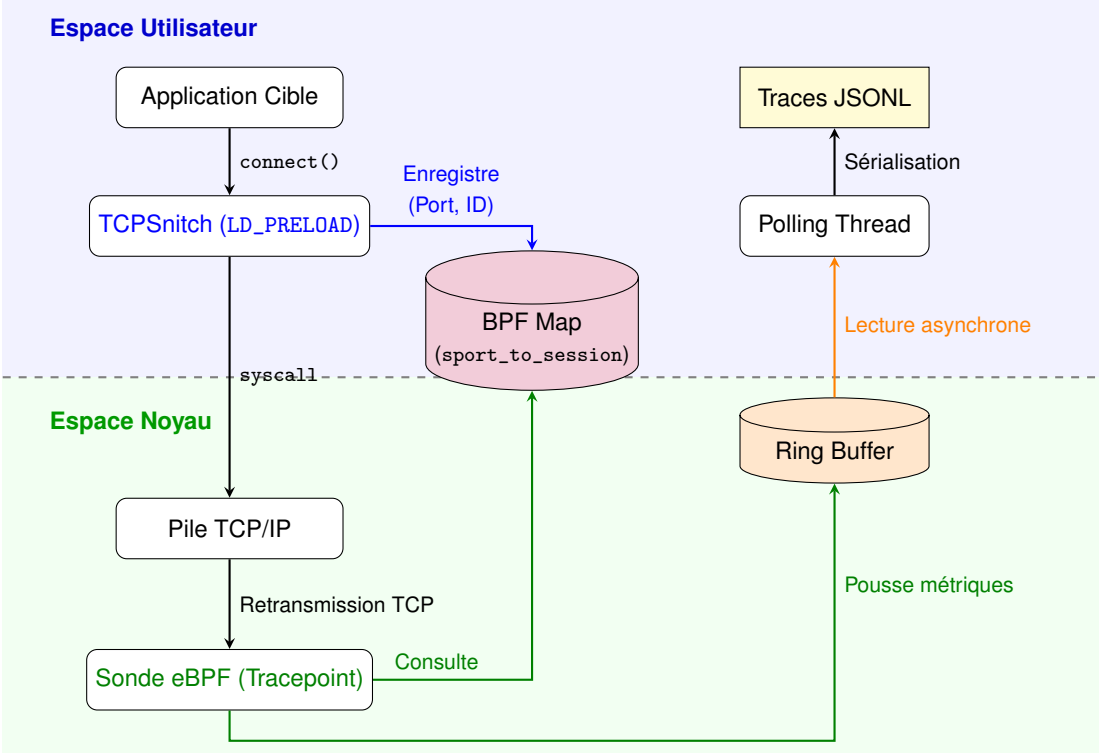

FIGURE 3.4 – Phase de capture : Collaboration asynchrone entre l'espace utilisateur (`LD_PRELOAD`) et le noyau (eBPF) via les structures partagées.

Cette architecture asynchrone est cruciale pour séparer l'interception des données de leur écriture sur le disque. Grâce à cela, le noyau et l'espace utilisateur travaillent en totale indépendance, évitant ainsi de bloquer l'exécution de l'application hôte.

3.4.2 Compilation et défis de portabilité

Le déploiement de sondes eBPF se heurte historiquement à un problème majeur : la disposition de la mémoire du noyau Linux change d'une version à l'autre. Une sonde compilée pour un appareil spécifique plantera sur un autre si elle ne connaît pas les nouveaux emplacements mémoire.

Pour que TCPSnitch fonctionne de manière portable sans exiger de lourde recompilation sur l'appareil cible, nous utilisons la technologie CO-RE (*Compile Once - Run Everywhere*) [7]. Cette approche s'appuie sur le format BTF (*BPF Type Format*) [6], une empreinte système qui permet de générer un fichier d'en-tête (`vmlinux.h`) décrivant l'agencement exact de la mémoire du noyau.

Le fichier `Makefile` automatise ce processus pour masquer cette complexité, avec des stratégies adaptées à chaque plateforme :

- **L'approche native Linux (Squelettes BPF) :** Le processus est standardisé. L'outil `bpftool` [22] analyse le code noyau et génère un « squelette » C. Ce squelette encapsule directement le programme eBPF compilé au sein de l'application utilisateur, offrant une interface prête à l'emploi et limitant les erreurs lors de l'échange de données.
- **L'extraction dynamique sur Android :** L'écosystème mobile étant très fermé, notre `Makefile` interroge directement le smartphone connecté (via la passerelle ADB). Il aspire l'empreinte mémoire du téléphone (`/sys/kernel/btf/vmlinux`) pour générer le `vmlinux.h` sur mesure, et récupère la bibliothèque `libbpf.so` native de l'appareil. Le code est ensuite compilé via l'Android NDK.

Grâce à cette automatisation, la gestion des offsets mémoire est résolue en amont. Au lancement de l'application, la bibliothèque `libbpf` se charge d'ajuster dynamiquement les dernières adresses pour garantir une exécution stable et sécurisée.

Gestion de la compatibilité et solution de secours (Android)

Sur les anciens terminaux mobiles (disposant d'un noyau antérieur à la version 5.10), les fonctions avancées d'eBPF ne sont pas toujours disponibles. Pour éviter que l'outil ne s'arrête brutalement ou ne provoque des erreurs, TCPSnitch choisit automatiquement la meilleure méthode de capture. Selon les capacités du smartphone détecté, le système utilise l'une des deux versions suivantes :

- **La version eBPF complète (`ebpf_collector_android_bpf.c`) :** Elle s'active sur les modèles récents compatibles. Cette version utilise toute la puissance du noyau pour

surveiller les connexions. Elle se charge d'ouvrir dynamiquement le fichier objet (`.o`) poussé sur le téléphone, de le charger en mémoire et d'attacher manuellement les sondes aux points de trace (par exemple, `tcp_retransmit_skb`).

- **La version de secours (`ebpf_collector_android.c`) :** Pour les anciens modèles, nous remplaçons le code eBPF par des fonctions simplifiées (dites « vides »). Cela garantit que l'outil ne plante jamais en cas d'incompatibilité matérielle ou d'erreurs. Sur ces appareils, TCPSnitch se comporte alors exactement comme sa version originelle de 2017 : il continue de tracer l'application de manière fluide via `LD_PRELOAD`, mais perd logiquement sa visibilité sur les événements internes du noyau (comme les retransmissions TCP ou les mesures de congestion).

Grâce à cette organisation, TCPSnitch devient un outil universel. Toute la complexité étant gérée en amont par le système de compilation, l'utilisateur n'a aucune configuration manuelle à effectuer : l'outil s'adapte automatiquement à l'appareil cible.

3.4.3 Corrélation des événements : la stratégie des Maps BPF

La difficulté principale de l'analyse réseau dans le noyau est la perte du contexte applicatif. Lorsqu'un événement de bas niveau survient, le noyau l'intercepte souvent en tâche de fond. À ce moment précis, il ne sait plus quelle application (ni quel descripteur de fichier) est à l'origine du trafic.

Pour relier les observations du noyau à la session surveillée par `LD_PRELOAD`, TCPSnitch s'appuie sur deux tables de hachage (*BPF Maps*) distinctes : l'une basée sur le port source (`sport_to_session`) pour le trafic TCP classique, et l'autre sur le descripteur de fichier (`fd_to_session_map`) pour les opérations asynchrones via `io_uring`.

Anticipation du format des octets

Dans le cadre d'un trafic TCP standard, lorsque le noyau détecte la perte d'un paquet, il déclenche le point de trace `tcp_retransmit_skb`. À cet instant, le seul identifiant réseau qui permet de retrouver l'application est le port source. L'outil doit donc enregistrer ce port dans la table `sport_to_session` dès l'ouverture de la connexion.

Cependant, le réseau et le processeur local ne lisent pas les séquences d'octets dans le même sens (*endianness*). Les standards d'Internet exigent le format *Network Byte Order* (octet de poids fort en premier), tandis que les processeurs utilisent généralement le format inverse (*Host Byte Order*) [23]. Lors de l'interception, la sonde eBPF lit la mémoire brute du noyau et peut y trouver le port sous l'un ou l'autre de ces formats selon le contexte.

Au lieu d'alourdir la sonde noyau avec des vérifications et des conversions mathématiques en temps réel, TCPSnitch délègue ce travail à l'espace utilisateur (`libc_overrides.c`). Lors de la connexion, l'outil calcule directement les deux formats possibles du port et les enregistre simultanément dans la table de hachage (voir l'étape 1 de la Figure 3.5). Ce qui permet à la

sonde noyau d'identifier immédiatement la connexion dans la table, assurant ainsi un filtrage à la fois robuste et très performant.

Filtrage strict et rejet du trafic système

Les sondes eBPF étant globales par nature, elles interceptent le trafic de l'intégralité du système d'exploitation. Pour garantir l'isolation des données, l'outil interroge la table de hachage pour chaque événement capturé. Si le port source n'y figure pas, l'événement ne provient pas de l'application tracée.

Pour isoler ce trafic parasite, la sonde noyau (`tcpsnitch.bpf.c`) lui attribue par défaut l'identifiant de rejet `9999` (comme illustré par l'étape 2 de la Figure 3.5). Une fois ces événements transmis via le *Ring Buffer*, le collecteur en espace utilisateur (`ebpf_collector.c`) les identifie et les ignore silencieusement.

Cette architecture s'est d'ailleurs révélée précieuse lors de l'évaluation de l'outil : en désactivant temporairement ce filtre, il devient possible de capturer et de quantifier avec précision le « bruit de fond » généré par le reste du système (cf. Chapitre 4.3).

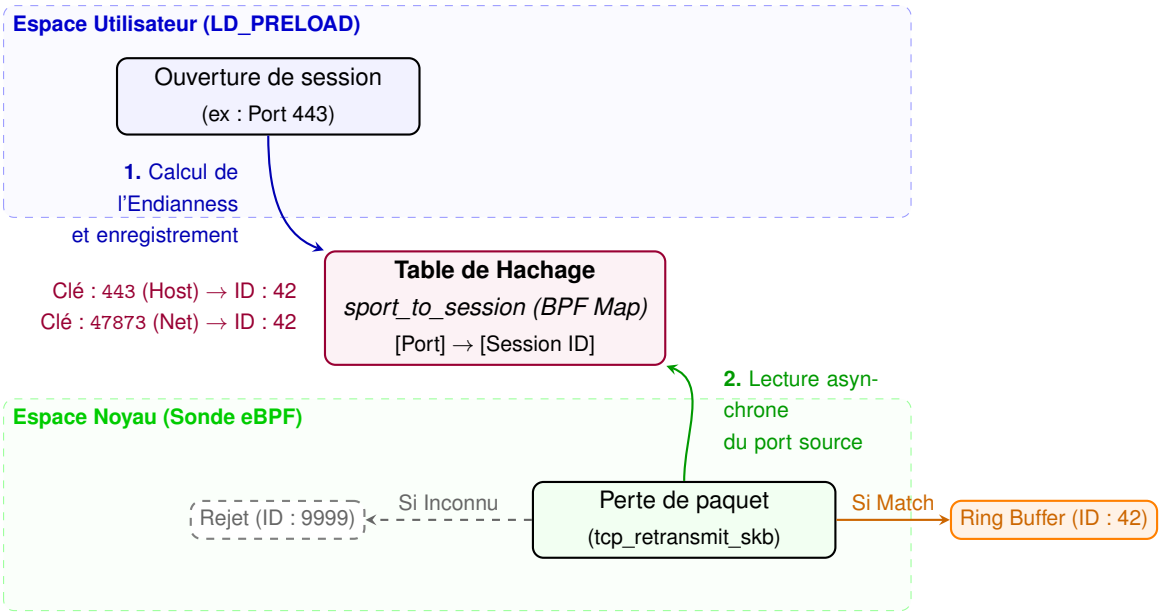

FIGURE 3.5 – Corrélation asynchrone via la BPF Map. L'espace utilisateur pré-enregistre les ports (en gérant l'*Endianness*), permettant à la sonde noyau d'associer instantanément le trafic à la session correspondante, ou de rejeter le bruit de fond (ID : 9999).

Nettoyage autonome et prévention des collisions

Les structures de données eBPF (*BPF Maps*) sont persistantes par nature [18]. Si un port réseau n'est pas explicitement supprimé à la fin d'une communication, sa réattribution

ultérieure par le système à un autre processus entraînera des collisions de données et la saturation de la table (`max_entries`).

Pour prévenir ce risque, TCPSnitch utilise une sonde autonome attachée à la machine à états TCP (`inet_sock_set_state`). Dès qu'une connexion se ferme (`TCP_CLOSE`), la sonde nettoie la table de hachage. Une simple opération binaire (`» 8 | « 8`) permet d'inverser les octets et de supprimer instantanément les deux formats d'*endianness* du port, garantissant une purge complète.

Le suivi asynchrone par descripteur

Contrairement au trafic TCP classique, les opérations d'entrées/sorties modernes (via `io_uring`) sont totalement asynchrones. L'application confie la tâche au système et continue son exécution. Par conséquent, l'interception classique par `LD_PRELOAD` devient aveugle au moment où l'opération s'achève réellement dans le noyau.

Pour capturer la fin de ce flux, TCPSnitch écoute le point de trace `io_uring_complete`. À cet instant précis, le port réseau n'est plus toujours disponible en mémoire. L'outil s'appuie donc sur sa seconde table de hachage (`fd_to_session_map`), qui relie l'événement à la session en utilisant le descripteur de fichier (`FD`). Pour faire le lien, la sonde exploite le champ `user_data` de l'API `io_uring`, dans lequel l'application a joint l'identifiant du descripteur lors de sa requête initiale.

Hypothèse de conception : Notre suivi asynchrone repose sur une condition stricte : l'application doit stocker son descripteur de fichier sous forme d'entier dans le champ `user_data`. La norme de l'API `io_uring` autorise en effet à y placer soit un entier, soit un pointeur [2]. Si l'utilisation d'un entier est la norme pour les programmes en C/C++, les langages de plus haut niveau (comme Go ou NodeJS) y injectent systématiquement des pointeurs. Puisque notre sonde noyau ne peut pas déchiffrer la mémoire de ces langages à la volée, elle ne peut pas y retrouver le descripteur. L'architecture de TCPSnitch est donc volontairement optimisée et limitée aux applications natives.

3.4.4 Exfiltration des données et sécurité de la mémoire

Le dernier défi de l'architecture eBPF concerne l'extraction des métriques. Les sondes s'exécutant dans le noyau traitent potentiellement des milliers d'événements par seconde. Utiliser des appels système classiques pour remonter ces informations vers l'espace utilisateur créerait un goulot d'étranglement qui ralentirait le système.

Pour garantir une transparence totale, TCPSnitch s'appuie sur un *Ring Buffer* [8]. Il s'agit d'une zone de mémoire circulaire ultra-rapide pré-allouée, permettant des transferts asynchrones et sans verrou (*lockless*) entre le noyau et l'application.

Optimisation de la charge utile

La mémoire étant une ressource critique dans le noyau, la taille des événements envoyés dans ce tampon doit être strictement minimisée. Pour y parvenir, tous les événements (retransmissions TCP, complétions `io_uring`) sont condensés dans une structure unifiée exploitant le concept d'`union` en langage C [24].

Par définition, tous les membres d'une union partagent le même espace d'adressage. Ainsi, la structure globale ne consomme en mémoire que la taille de son plus grand sous-événement. Les métadonnées communes (comme l'horodatage, l'identifiant de session et le `PID`) sont partagées en en-tête, tandis que la charge utile spécifique s'adapte dynamiquement : le reste du bloc mémoire contiendra soit les métriques de congestion TCP (`snd_cwnd`, `srtt`), soit le code de retour asynchrone (`res`) d'`io_uring`.

Dans le noyau, le processus d'envoi est ainsi extrêmement court et peu coûteux : la sonde réserve un bloc mémoire de taille fixe (`bpf_ringbuf_reserve`), le remplit avec le contexte adéquat, puis valide son expédition (`bpf_ringbuf_submit`) sans aucun gaspillage d'espace.

Sécurité des accès concurrents

En espace utilisateur, un fil d'exécution dédié (`polling_thread`) lit ce tampon en continu pour formater et écrire les données au format JSONL sur le disque.

Cependant, ce découpage soulève un problème de concurrence critique (*Race Condition*). L'outil fonctionne en simultané : l'application surveillée (via `LD_PRELOAD`) peut tenter d'écrire une trace au même instant précis où le *thread* eBPF tente d'y écrire un événement réseau. Sans protection, les textes s'entremêleraient, corrompant irrémédiablement le fichier de logs.

Pour garantir l'intégrité absolue des données, TCPSnitch sécurise l'accès au fichier via un verrou d'exclusion mutuelle (*Mutex* POSIX). Avant chaque écriture sur le disque, que ce soit depuis l'intercepteur utilisateur ou depuis le moniteur eBPF, le programme exige un accès exclusif. Ce verrouillage ne dure que le temps d'écrire une ligne JSONL complète, avant d'être immédiatement relâché.

Cette asynchronie de bout en bout, allant de la collecte sans verrou (*lockless*) dans le noyau jusqu'à l'écriture sécurisée par Mutex sur le disque, permet à TCPSnitch de surveiller le trafic de manière fiable, tout en conservant un impact processeur virtuellement indétectable.

3.5 Outil d'analyse visuelle

L'architecture de TCPSnitch génère un flux massif de données issues de trois sources distinctes (`LD_PRELOAD`, `eBPF`, et `Netlink`). Pour exploiter ces traces facilement, nous avons développé un tableau de bord local et interactif en Python (`Streamlit`).

Au-delà de la simple visualisation, le rôle critique de cet outil est de résoudre le problème de la réconciliation temporelle. En effet, les différentes couches du système n'utilisent pas la même référence chronologique :

- L'espace utilisateur et Netlink s'appuient sur l'horloge système standard (Epoch, exprimée en microsecondes).
- Les sondes eBPF utilisent l'horloge monotone du noyau (temps écoulé depuis le démarrage de la machine, en nanosecondes).

Lors du chargement des fichiers (`app.py`), l'analyseur calcule un point d'origine commun (t_0) pour ces deux horloges. Tous les événements sont alors normalisés sur un axe chronologique unique.

Grâce à cette synchronisation, le tableau de bord permet enfin de corréler visuellement les actions de l'application avec le comportement du système. Il devient possible d'inspecter le cycle de vie d'un *socket*, d'observer les variations de la fenêtre de congestion (CWND), ou encore de lier une erreur d'entrée/sortie à une perte de signal détectée par Netlink.

3.6 Conclusion

Ce chapitre a présenté la transformation de TCPSnitch en une plateforme d'observabilité hybride. En combinant la modernisation de l'interception `LD_PRELOAD`, l'intégration de `Netlink` pour la topologie réseau et la puissance d'eBPF pour l'analyse noyau, l'outil offre désormais une visibilité complète sur la pile TCP/IP. L'intégralité du code source de cette nouvelle architecture est d'ailleurs consultable en libre accès (voir Annexe 7).

L'élaboration de cette architecture a permis de résoudre plusieurs défis d'ingénierie critiques : la réconciliation des événements asynchrones via les *BPF Maps*, la portabilité vers Android grâce au standard CO-RE, et l'exfiltration asynchrone des métriques réseau via des tampons circulaires (*Ring Buffer*). Enfin, le développement d'un analyseur dédié permet d'unifier ces sources de données sur une chronologie commune.

Grâce à cette base technique, TCPSnitch est prêt pour une évaluation en conditions réelles. Le chapitre suivant sera consacré à l'expérimentation afin de mesurer l'impact de l'outil sur les performances du système et de valider sa capacité de diagnostic.

Chapitre 4

Validation de l'outil et Limites

Après avoir détaillé l'architecture de TCPSnitch, ce chapitre se consacre à sa validation expérimentale. L'objectif est de mesurer l'impact réel de l'outil sur le système et de prouver sa précision technique face au trafic réseau.

Nous identifierons également les limites de notre approche face aux protocoles modernes et aux contraintes de sécurité. Nous présenterons d'abord notre méthodologie et nos benchmarks, puis nous détaillerons les restrictions de déploiement rencontrées sous Android et Linux.

4.1 Méthodologie et environnement de test

L'évaluation de TCPSnitch repose sur un protocole automatisé visant à isoler l'impact de notre architecture du bruit de fond du système. Tous les tests ont été réalisés via une connexion réseau filaire (Ethernet) sur une configuration matérielle fixe [1] afin d'assurer la stabilité et la comparabilité des mesures.

Validité statistique

Afin d'obtenir des résultats fiables et de lisser les variations naturelles du système, chaque scénario est exécuté 30 fois ($N = 30$). Les métriques récoltées (temps de transfert, usage CPU) sont agrégées en utilisant la médiane, une valeur moins sensible aux écarts extrêmes que la moyenne arithmétique.

1. Plateforme de test : AMD Ryzen 5 PRO 2500U, 6,7 Go de RAM, interface réseau filaire, sous Ubuntu 24.04.4 LTS (Noyau Linux 6.17.0-22-generic).

Environnement réseau contrôlé

Puisque l'utilisation d'Internet introduit des instabilités imprévisibles, nous utilisons l'outil Linux `tc qdisc netem` pour simuler de manière rigoureuse des conditions réseau dégradées. Ce mécanisme nous permet d'injecter artificiellement une latence fixe de 20 ms et des pertes de paquets allant de 2% à 20%.

Il est important de préciser que ces règles de contrôle de trafic ont été appliquées directement sur l'interface réseau de notre machine de test, et uniquement dans le sens sortant (*egress*). Les règles `tc` opérant de manière unidirectionnelle, cette configuration bride exclusivement le trafic émis par notre système. Ce choix nous permet d'observer avec précision le comportement de la pile réseau locale face à une congestion sévère, tout en fixant un cadre d'expérimentation maîtrisé.

4.2 Évaluation des performances et du surcoût

L'ajout de sondes dans le système d'exploitation consomme inévitablement des ressources. Pour vérifier que TCPSnitch reste léger, nous avons mesuré son surcoût (*overhead*) en termes de temps d'exécution, d'utilisation du processeur (CPU) et de mémoire (RAM).

Le test consiste à télécharger un fichier de 100 Mo via la commande `curl` sur le réseau local (`localhost`). Ce choix permet d'éliminer la latence d'Internet pour pousser l'outil dans ses limites à très haut débit (*stress test*). Le test a été répété 30 fois sous quatre conditions :

1. **Baseline :** Téléchargement normal, sans aucun outil de capture.
2. **TCPSnitch Standard :** Interception classique (`LD_PRELOAD` et Netlink).
3. **TCPSnitch Complet :** Ajout de l'interception au sein du noyau (`eBPF`).
4. **TCPSnitch Complet sous perte :** Avec eBPF et 10 % de perte de paquets artificielle (via `netem`) pour simuler un réseau fortement dégradé.

Les résultats médians de ces scénarios sont synthétisés dans le Tableau 4.1.

Condition de test	Temps (s)	Surcoût (s)	CPU (%)	RAM (Mo)
1. Baseline (Sans outil)	0.23 s	-	-	-
2. TCPSnitch (Standard)	0.38 s	+ 0.15 s	$\sim$ 3.5 %	7.4
3. TCPSnitch (+ eBPF)	0.60 s	+ 0.37 s	$\sim$ 2.1 %	7.4
4. TCPSnitch (+ Perte 10%)	10.40 s	N/A	$<$ 1.0 %	7.4

TABLE 4.1 – Impact de TCPSnitch 2026 sur un transfert local de 100 Mo (médianes sur 30 itérations).

Interprétation des résultats et comportement sous contrainte

L'analyse de ce tableau met en évidence trois points essentiels concernant l'impact matériel de l'outil :

Le temps d'exécution et l'écriture des données
On observe que le passage de la version de base (0,23 s) à la version eBPF complète (0,60 s) ajoute un délai d'un peu plus d'un tiers de seconde (+ 0,37 s). Ce surcoût s'explique logiquement par le travail supplémentaire que l'outil doit accomplir : formater les données et les écrire physiquement sur le disque sous forme de fichiers journaux (JSONL). Cependant, il faut relativiser ce ralentissement : transférer 100 Mo en 0,60 seconde correspond à un débit très élevé (environ 1,3 Gbps). L'outil reste donc performant face à un volume de données important.

La stabilité de la mémoire (RAM)
Un constat rassurant est que la consommation de mémoire vive reste strictement figée à 7,4 Mo, que l'on utilise la version standard, la version eBPF, ou que le réseau soit fortement dégradé. Cette observation nous permet de confirmer que l'outil alloue correctement sa mémoire au démarrage (pour ses tables de hachage et son tampon) et qu'il n'y a pas de fuite de mémoire (*memory leak*) au fil du temps.

L'impact de la congestion réseau
La dernière ligne du tableau présente une situation intéressante : lorsqu'on dégrade artificiellement le réseau, le temps de transfert explose (10,40 s), mais paradoxalement, l'utilisation du processeur (CPU) chute en dessous de 1 %. Notre hypothèse, basée sur le fonctionnement standard de TCP, est que face aux paquets perdus, le système freine l'envoi des données (mécanismes de contrôle de congestion et délais de retransmission). Pendant ces longues pauses de la carte réseau, le processeur ne travaille presque plus : il est simplement en attente. Dans cette situation, le temps de calcul nécessaire à TCPSnitch devient invisible face à la lenteur du réseau. C'est pourquoi le surcoût temporel de l'outil est noté comme non applicable (N/A) : le goulot d'étranglement exclusif est le réseau, et non l'outil d'interception.

En conclusion, ces mesures montrent que TCPSnitch est un outil léger. Dans des conditions réelles d'utilisation (sur un réseau Wi-Fi ou mobile standard), la vitesse de téléchargement sera presque toujours limitée par la connexion physique. L'impact de notre outil sera donc très discret, voire imperceptible, pour la plupart des utilisateurs.

4.3 Précision et fiabilité du filtrage eBPF

Comme détaillé lors de la conception (voir chapitre 3.4.3), les sondes eBPF s'exécutent de manière globale et interceptent les événements réseau de l'intégralité du système d'exploitation. L'enjeu de TCPSnitch est d'isoler les paquets de l'application ciblée grâce à une table de hachage (*BPF Map*), et de marquer le trafic parasite appartenant aux autres processus avec un identifiant de rejet arbitraire (la valeur `9999`).

Afin d'évaluer l'efficacité de ce mécanisme d'isolation, nous avons temporairement désactivé le rejet silencieux en espace utilisateur. Cela nous permet de journaliser et de quantifier le trafic classé comme « bruit de fond » (9999).

Protocole expérimental

Pour générer un trafic contrôlé, nous avons développé un script Python exécutant des requêtes HTTP POST vers le serveur public `httpbin.org`. Chaque requête transmet une charge utile aléatoire de 500 Ko. Ce script exploite le *multithreading* pour ouvrir simultanément un nombre défini de connexions (sockets). L'outil `netem` a été appliqué sur l'interface sortante pour forcer la perte de paquets et déclencher des retransmissions TCP. Nous avons testé quatre scénarios de charge :

1. **Scénario A (Nominal) :** 1 connexion, 5 % de perte de paquets.
2. **Scénario B (Stress) :** 1 connexion, 15 % de perte (forte congestion).
3. **Scénario C (Parallèle) :** 3 connexions simultanées, 10 % de perte.
4. **Scénario D (Massif) :** 10 connexions simultanées, 5 % de perte.

Le taux de précision présenté dans le Tableau 4.2 correspond à la proportion d'événements interceptés par le noyau qui ont été correctement associés (via leur port) aux connexions de notre script Python, par rapport au volume total d'événements capturés sur le système.

Scénario	Retransmissions (Total)	Bruit (ID : 9999)	Corrélés	Précision
A. Nominal (1 socket)	625	37	588	**94,1 %**
B. Stress (1 socket)	1815	124	1691	**93,2 %**
C. Parallèle (3 sockets)	3163	81	3082	**97,4 %**
D. Massif (10 sockets)	7587	21	7566	**99,7 %**

TABLE 4.2 – Évaluation de la précision du filtrage eBPF. La précision illustre la capacité de l'outil à extraire le trafic de la session applicative tout en isolant le bruit du système (9999).

Analyse des résultats et origine du bruit

L'architecture démontre une excellente fiabilité : elle relie correctement l'écrasante majorité des événements à la session ciblée (jusqu'à 99,7 % sous forte charge concurrente au Scénario D). Le trafic système parasite a été systématiquement identifié et isolé sous l'identifiant 9999.

Il est important de souligner que les événements non corrélés (classés 9999 dans les scénarios A et B) ne constituent pas un défaut ou une « perte » de l'outil. Au contraire, ils valident le bon fonctionnement du filtrage. Ce volume de rejet s'explique par deux phénomènes observables :

- **Le trafic de fond du système d'exploitation :** L'outil `netem` dégradant l'interface réseau de manière globale, les autres processus en arrière-plan (démons système, synchronisations) subissent également des pertes de paquets. Leurs retransmissions TCP sont logiquement interceptées par la sonde globale eBPF, puis correctement rejetées avec le code `9999` car leurs ports sources n'appartiennent pas à notre script Python.
- **Les retransmissions asynchrones de fermeture :** À la fin d'un transfert, le script Python ferme son descripteur de fichier. Cette action déclenche la purge de la référence dans notre table eBPF (nettoyage autonome). Cependant, sur un réseau dégradé, le noyau peut continuer d'essayer de transmettre des segments de clôture de connexion (`FIN` ou `ACK`) en arrière-plan. L'outil intercepte ces ultimes paquets, mais ne les trouvant plus dans sa table de corrélation, il les classe logiquement comme du bruit.

En conclusion, l'outil parvient avec succès à accomplir son objectif : extraire fidèlement le signal de l'application cible au milieu du bruit généré par l'activité globale du noyau Linux.

4.4 Réactivité du système et limites protocolaires

Après avoir validé le filtrage eBPF, nous devons tester deux autres aspects de l'outil : la capacité de Netlink à suivre l'état du réseau en temps réel et la compatibilité des sondes avec les nouveaux protocoles web.

Performance du monitoring Netlink

Le rôle de Netlink est de détecter les changements physiques (perte de WiFi, changement d'IP) pour expliquer les erreurs de l'application. Pour être efficace, cette surveillance doit être complète et instantanée.

Pour tester ses limites, nous avons simulé 3000 changements d'adresses IP en rafale sur l'interface locale, sans aucun délai entre les opérations. Les résultats montrent une fiabilité parfaite : 100 % des événements ont été capturés, même lors de pics de trafic extrêmes.

En termes de rapidité, l'analyse des horodatages (*timestamps*) indique une latence médiane de seulement 9,60 millisecondes. Ce délai très court garantit que l'outil peut corréler précisément une coupure réseau avec une erreur de transfert signalée par l'application.

L'angle mort de l'analyse noyau : Le cas HTTP/3 (QUIC)

Bien que TCPSnitch soit extrêmement performant pour disséquer le trafic TCP, l'adoption massive du protocole HTTP/3 pose un défi architectural majeur. Pour illustrer cette limite, nous avons mis en place une expérience ciblée.

Protocole expérimental

L'expérience repose sur l'envoi d'une charge utile de 1 Mo (générée aléatoirement) vers un serveur public de test (`https://httpbin.org/post`). Le transfert est effectué côté client via la commande `curl`, qui permet de forcer consécutivement l'utilisation des protocoles HTTP/1.1, HTTP/2 et HTTP/3. L'outil `netem` est appliqué sur l'interface sortante du client pour injecter des pertes de paquets allant de 2 % à 20 %. Chaque condition a été répétée 30 fois.

Les résultats, présentés dans le Tableau 4.3, divisent ces protocoles en deux catégories distinctes.

Protocole utilisé	Perte 2 %	Perte 5 %	Perte 10 %	Perte 20 %
HTTP/1.1	Capturé	Capturé	Capturé	Capturé
HTTP/2	Capturé	Capturé	Capturé	Capturé
HTTP/3 (QUIC)	**0**	**0**	**0**	**0**

TABLE 4.3 – Détection des retransmissions eBPF côté client selon le protocole de la couche application.

D'une part, les protocoles reposant sur TCP (HTTP/1.1 et HTTP/2) sont parfaitement pris en charge. Pour ces flux, les sondes eBPF interceptent l'intégralité des retransmissions et des métriques de congestion. La Figure 4.1 illustre d'ailleurs le niveau de détail obtenu sur ces protocoles, où l'on observe nos sondes capturer avec une grande précision les chutes de la fenêtre de congestion lors des pertes de paquets.

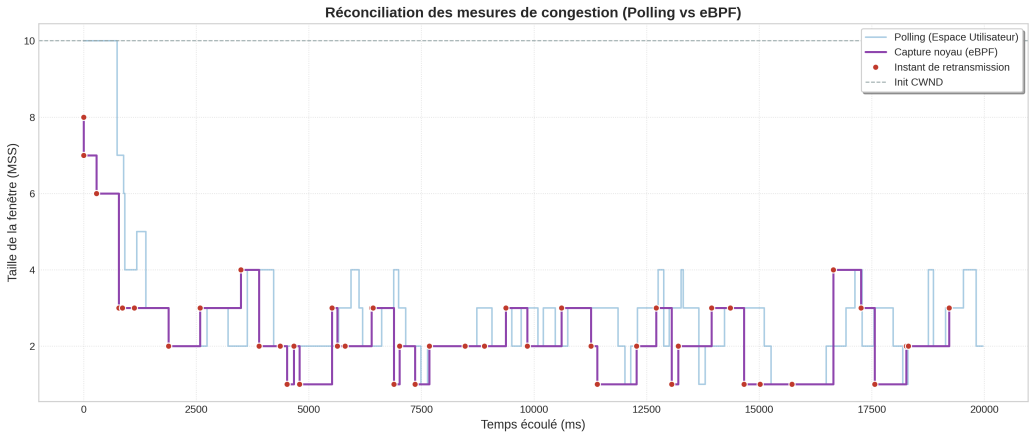

FIGURE 4.1 – Superposition des mesures de la fenêtre de congestion (trafic TCP). La courbe bleue représente le sondage applicatif (*polling*), tandis que les points rouges correspondent aux captures précises effectuées par l'eBPF lors des retransmissions du noyau.

D'autre part, sous HTTP/3, le tableau montre un silence absolu (0 événement détecté

par eBPF). Ce résultat n'est pas un dysfonctionnement de notre outil, mais s'explique par la conception même de HTTP/3, qui repose sur le protocole QUIC.

Contrairement à TCP, la gestion des pertes, de la fiabilité et de la congestion de QUIC n'est plus déléguée au noyau Linux, mais gérée directement de manière chiffrée en espace utilisateur par l'application. Cette différence de conception provoque un aveuglement partiel de notre chaîne de capture :

1. **Côté noyau (`eBPF`) : L'aveuglement structurel.** Le noyau ne gérant pas le transport fiable de QUIC, nos sondes eBPF attachées aux fonctions de retransmission TCP (`tcp_retransmit_skb`) ne sont naturellement jamais déclenchées. L'extraction de la dynamique interne de QUIC est donc structurellement hors de porté de TCPSnitch. L'analyse de ces flux nécessite d'autres approches standardisées au niveau applicatif, comme l'utilisation du format de journalisation `qlog` intégré directement dans les bibliothèques QUIC [25].
2. **Côté espace utilisateur (`LD_PRELOAD`) : La visibilité applicative.** Bien que l'outil perde la vision de la congestion, il n'est pas totalement aveugle à QUIC. L'intercepteur `LD_PRELOAD` continue de capturer l'ouverture des sockets UDP ainsi que les appels de transfert (`sendmsg()` et `recvmsg()`). TCPSnitch reste donc parfaitement capable de comptabiliser l'allocation des descripteurs et le volume d'octets échangés sur le réseau.

Cette distinction est fondamentale : elle délimite avec précision le périmètre expérimental de notre outil. Si TCPSnitch se révèle structurellement inopérant pour diagnostiquer la congestion interne de HTTP/3 (QUIC), il demeure en revanche parfaitement capable d'intercepter la création des sockets UDP et de mesurer les volumes d'octets échangés en surface par l'application.

4.5 Robustesse sous charge et stabilité architecturale

Pour garantir la viabilité de TCPSnitch sur de longues sessions de capture, nous avons évalué sa stabilité sous charge, puis testé sa résilience face à des comportements extrêmes afin d'en identifier les failles potentielles.

Conformément à la méthodologie établie précédemment, nous avons conçu un protocole de test automatisé générant un trafic HTTP ciblé (requêtes vers `httpbin.org`). Trois scénarios de *stress-test* ont ainsi été définis pour pousser l'architecture dans ses retranchements :

1. **Scénario A (Intégrité concurrente) :** L'outil surveille un script générant 100 connexions simultanées via des *threads* parallèles. L'objectif est de vérifier que les écritures concurrentes ne corrompent pas les fichiers de journaux JSON.
2. **Scénario B (Intégrité séquentielle) :** L'outil surveille l'enchaînement rapide de 50 requêtes séquentielles. Ce test valide la précision du suivi, garantissant qu'aucun événement n'est perdu entre deux ouvertures de sockets très rapprochées.

3. **Scénario C (Stabilité mémoire) :** L'application génère des transferts continus en boucle pendant 10 minutes ininterrompues. En parallèle, l'empreinte mémoire physique (RSS - *Resident Set Size*) du processus d'interception est relevée dynamiquement afin de traquer d'éventuelles fuites de mémoire.

Les résultats de ces expériences sont illustrés par la Figure 4.2.

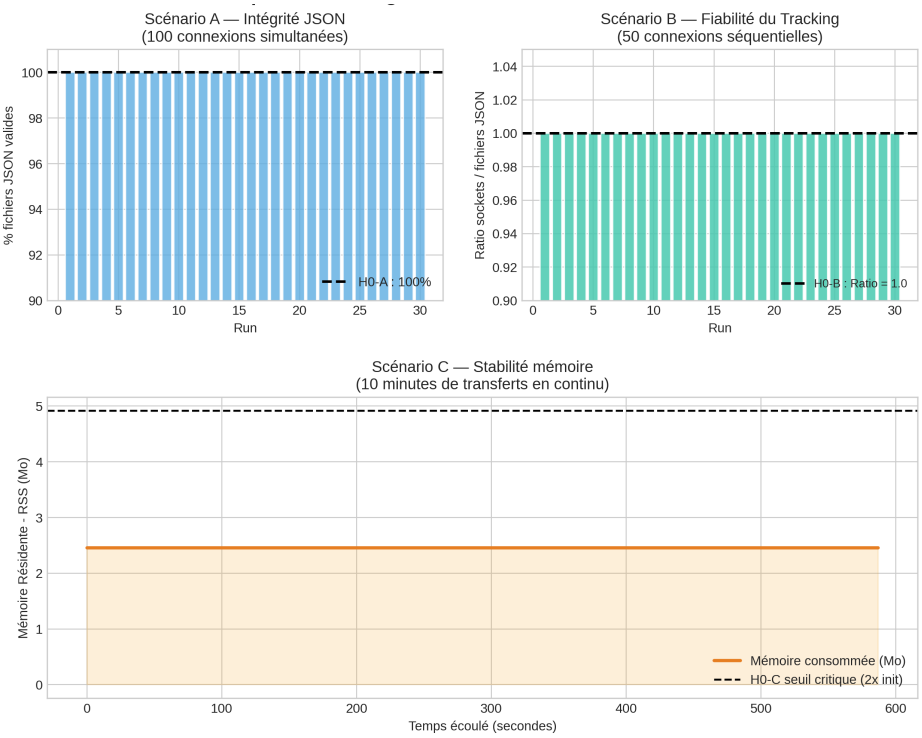

FIGURE 4.2 – Résultats des tests de stabilité nominale. Les scénarios A et B affichent un taux de capture et d'intégrité parfait sous charge, tandis que le scénario C démontre une consommation mémoire strictement constante sur la durée.

L'analyse de ces métriques valide la robustesse de l'architecture. Tout d'abord, l'empreinte mémoire (RSS) reste figée à $\sim$2,5 Mo sur l'intégralité des 10 minutes du Scénario C, écartant définitivement tout risque de fuite de mémoire (*memory leak*) au sein des structures eBPF ou de l'espace utilisateur. De plus, lors des scénarios A et B, l'outil a généré 100 % de fichiers JSON syntaxiquement valides. Ce taux de réussite parfait prouve que nos verrous d'exclusion mutuelle (*Mutex*) protègent efficacement les journaux contre la corruption (*Race Condition*), même lorsque des dizaines de *threads* tentent d'y inscrire leurs événements réseau à la même milliseconde.

Limites de l'interception : la faille des fermetures massives

Comme anticipé lors de l'implémentation de notre mécanisme de capture spéculative (*State Caching*, cf. Chapitre 3.2.2), l'interception de l'appel système `close_range()` expose l'outil à un risque majeur d'entrelacement des descripteurs de fichiers.

Pour vérifier cette hypothèse et mesurer l'impact de cette contrainte architecturale sous forte charge, nous avons développé deux programmes de test en langage C. Ces derniers invoquent directement cet appel système de nettoyage massif de manière très agressive :

1. **Scénario D (Fermeture partielle) :** Le programme alloue 100 sockets TCP, puis invoque `close_range()` pour forcer la destruction stricte de la première moitié de ces descripteurs.
2. **Scénario E (Fermeture totale) :** Le programme alloue 300 sockets TCP, évalue la plage complète des descripteurs qu'il utilise (du plus petit au plus grand), puis invoque `close_range()` sur l'intégralité de cette plage pour un nettoyage absolu.

Les résultats valident empiriquement notre mise en garde. Lors d'une fermeture partielle (Scénario D), près de 28 % des fichiers générés sont corrompus, bien que la moitié des événements de fermeture (1 500) aient pu être interceptés. La fermeture totale (Scénario E) provoque quant à elle l'effondrement de l'outil : plus de 62 % de corruption, perte totale des traces de fermeture (aucun événement capturé).

L'explication confirme le phénomène d'entrelacement. Via `LD_PRELOAD`, TCPSnitch partage la table des descripteurs (FDs) de l'application. En appelant `close_range()` [20], l'hôte nettoie aveuglément une large plage de FDs, détruisant involontairement les canaux d'écriture de notre outil (journaux et *Ring Buffer*). Bien que l'état soit correct en RAM, l'outil est physiquement déconnecté du système.

TCPSnitch connaissant ses propres descripteurs, il pourrait théoriquement contourner cette faille en filtrant la demande (ce qui obligerait à scinder l'appel global en de multiples appels `close()` individuels). Toutefois, cela annulerait l'avantage de performance natif de `close_range()`.

En définitive, cette défaillance illustre la limite principale de l'interception via `LD_PRELOAD` : l'outil étant greffé à l'application, il en subit directement les purges de mémoire.

4.6 Sandboxing et restrictions de sécurité : les limites de l'approche hybride

Comme nous l'avions anticipé dans l'état de l'art, l'injection en espace utilisateur fait face aux mécanismes de sécurité modernes (*sandboxing*). Notre objectif en intégrant eBPF était de contourner l'aveuglement de `LD_PRELOAD`. Cependant, nos tests de déploiement en conditions réelles ont mis en évidence une faille architecturale majeure de notre conception : en utilisant

`LD_PRELOAD` comme vecteur d'initialisation pour eBPF, notre outil hybride hérite inévitablement des restrictions de sécurité imposées au processus cible.

Le verrouillage des privilèges eBPF sous Android (SELinux)

Lors de nos tests sur un smartphone Google Pixel 7a (équipé du noyau Linux 6.1), l'activation de la couche eBPF a échoué, forçant l'outil à basculer en mode `LD_PRELOAD` exclusif. Les journaux système (*logcat*) révèlent des alertes SELinux (`avc: denied`) et un rejet explicite du noyau (`Operation not permitted`).

Ces erreurs découlent directement du modèle de sécurité d'Android. Lors de son installation, chaque application se voit attribuer un identifiant utilisateur unique (*UID*) destiné à l'isoler au sein d'une *application sandboxée* strict [26]. S'exécutant sous le domaine SELinux restreint `untrusted_app`, le processus est totalement privé des privilèges Linux de bas niveau (notamment `CAP_BPF` ou `CAP_SYS_ADMIN`) [27], indispensables pour interagir avec le sous-système eBPF du noyau.

Notre architecture présente ici une limite conceptuelle : c'est la bibliothèque `LD_PRELOAD`, confinée dans cet espace utilisateur isolé et non privilégié, qui tente de charger le bytecode eBPF.

À titre de comparaison, les concepteurs du framework *ePacket* [28] parviennent à utiliser eBPF sur Android en adoptant une approche totalement différente. Au lieu de laisser l'application cible charger les sondes, ils s'appuient sur un outil externe nommé `eadb` exécuté avec les privilèges administrateur (*root*). Notre tentative démontre donc qu'une approche où l'application non privilégiée essaie elle-même d'instrumenter le noyau est structurellement vouée à l'échec face aux règles de SELinux.

Le sandboxing agressif des navigateurs (le cas Firefox)

Le second mur de sécurité concerne les applications Linux manipulant des données sensibles. Lors de nos tentatives d'instrumentation, le navigateur Mozilla Firefox s'est systématiquement effondré au démarrage.

Ce crash n'est pas lié à l'interception de l'API Socket, mais aux nouvelles fonctionnalités hybrides que nous avons introduites. Firefox isole ses processus réseau via des filtres *Seccomp-BPF* [13] agissant comme une liste blanche très stricte : tout appel système inattendu provoque la mort immédiate du processus (signal `SIGSYS`).

Même en tentant d'assouplir ces *sandboxing* via des variables d'environnement, les appels systèmes générés en arrière-plan par la nouvelle architecture de TCPSnitch (création de threads asynchrones, ouverture de sockets bruts Netlink, manipulation d'eBPF) violent radicalement les règles de sécurité du navigateur. Les applications modernes hautement sécurisées assimilent les comportements de notre outil hybride à ceux d'un logiciel malveillant et s'auto-

terminent préventivement.

4.7 Conclusion

L'évaluation de TCPSnitch aboutit à un constat clair : notre outil est performant, mais son architecture lui impose des limites strictes.

D'un côté, l'outil excelle dans son domaine de prédilection : les applications natives standards. Dans ce contexte, il remplit parfaitement son rôle. Il consomme très peu de ressources, réagit instantanément aux coupures réseau, et filtre le trafic avec une précision de presque 100 %.

De l'autre côté, l'outil se heurte à trois limites majeures face aux environnements modernes : il est aveugle au protocole HTTP/3, il s'effondre si l'application détruit massivement ses descripteurs (`close_range`), et il est totalement bloqué par les sécurités récentes de SELinux.

La cause de ces limites est purement architecturale. En utilisant `LD_PRELOAD` pour s'injecter dans l'application, notre outil devient prisonnier de celle-ci. S'il s'exécute à l'intérieur du processus, il subit logiquement les mêmes restrictions de sécurité et partage les mêmes vulnérabilités matérielles.

Cette analyse honnête est essentielle, car elle nous a permis de définir exactement où TCPSnitch fonctionne de manière optimale. C'est donc en l'utilisant dans ce périmètre de confiance que nous avons pu récolter notre jeu de données réseau. Le chapitre suivant sera consacré à la discussion de ces données et à leur comparaison avec les résultats de l'étude originelle de 2017.

Chapitre 5

Méthodologie et Constitution du Jeu de Données

Ce chapitre détaille la méthodologie employée pour constituer notre nouveau jeu de données. Contrairement à l'étude de 2017 qui reposait sur des interactions manuelles non calibrées, nous présentons une approche automatisée et reproductible. Nous décrivons ensuite la composition de nos deux jeux de données, capturés respectivement sous Android et Linux.

5.1 L'évolution méthodologique

Pour dépasser le manque de reproductibilité des tests environnementaux de l'étude originelle, notre approche s'inspire de la méthodologie de création de vérité terrain (*ground-truth*) issue du projet MIRAGE [29]. Nous avons conçu des scripts d'automatisation (*Bash*) structurant chaque cycle de capture en deux phases chronométrées :

1. **Phase d'activité (3 minutes) :** L'application subit une interaction humaine réaliste (défilement, appel, messagerie).
2. **Phase de repos (1 minute) :** Toute interaction cesse (*idle time*) afin de capturer le trafic d'arrière-plan et les fermetures asynchrones de sockets.

Ce cycle de capture de 4 minutes est répété indépendamment pour chacun des trois scénarios suivants : la **Baseline** (une connexion stable servant de référence expérimentale), le **Handover** (une désactivation scriptée de l'interface principale à $t = 90\,\text{s}$, forçant le système à basculer vers une connexion secondaire, comme le réseau cellulaire sur mobile ou le Wi-Fi sur ordinateur), et le **Blackout** (une coupure réseau totale et scriptée de 15 secondes en cours de session, permettant d'observer la stratégie de rattrapage de l'application et les re-

transmissions TCP du noyau). Ainsi, chaque fonctionnalité ciblée est exécutée à trois reprises, générant au total trois traces distinctes par application.

5.2 Le jeu de données Android

Le jeu de données Android a été capturé sur un smartphone Google Pixel 4a sous Android 14 (noyau Linux 4.1.278) disposant d'un accès *root*. Afin de privilégier une analyse comportementale en profondeur, nous avons sélectionné cinq applications ultra-dominantes, représentatives du trafic mobile moderne. Le Tableau 5.1 synthétise les 21 traces générées par le croisement de ces fonctionnalités et des scénarios de stress.

Catégorie	**Application**	**Fonctionnalité tracée**	**Scénarios exécutés**
Réseau Social	Facebook v562.0.0.51.73	Défilement de flux (*scroll*)	Baseline, Handover, Blackout
Messagerie	Messenger v562.0.0.53.83	Discussion textuelle (*chat*)	
		Appel vidéo	
	Discord v329.13	Discussion textuelle (*chat*)	
		Appel audio	
Streaming Vidéo	Twitch v29.6.2	Diffusion en direct (*Live HD*)	
Divertissement	TikTok v45.3.3	Défilement de vidéos courtes	

TABLE 5.1 – Composition du dataset Android 2026 (21 traces sur Google Pixel 4a, Android 14).

5.3 Le jeu de données Linux

Pour le jeu de données Linux, capturé sous la distribution Ubuntu 24.04 LTS (noyau Linux 6.17.0), nous suivons scrupuleusement la décision méthodologique de Vander Schueren [1] consistant à écarter les outils en ligne de commande (*CLI*) et les utilitaires réseaux spécialisés (`curl`, `git`, `nmap`). Comme le soulignait l'étude de 2017 : « *Ces applications présentent souvent des interactions singulières avec l'API Socket. Étant donné qu'elles ont tendance à fausser les résultats, nous les excluons lors du calcul des statistiques sur le jeu de données Linux* ».

Nous nous concentrons donc exclusivement sur la catégorie des applications lourdes dotées d'une interface graphique. Nous avons sélectionné trois applications natives complexes pour éprouver la couche eBPF. Le tableau 5.2 résume les 12 traces Linux obtenues.

Catégorie	Application	Fonctionnalité tracée	Scénarios exécutés
Messagerie	Telegram v6.8.2	Discussion textuelle (*chat*)	Baseline, Handover, Blackout
		Appel audio	
Lecteur Média	VLC v3.0.20	Lecture d'un flux audio en continu [2]	
Partage P2P	Transmission v4.0.5	Téléchargement d'un fichier ISO [3]	

TABLE 5.2 – Composition du dataset Linux 2026 (12 traces sur Ubuntu 24.04 LTS).

5.4 Conclusion

Ce chapitre a permis de définir rigoureusement la méthodologie d'acquisition des données ainsi que la composition du nouveau *dataset*. En s'inspirant du projet *MIRAGE* et en automatisant les scénarios de stress réseau (Baseline, Handover, Blackout), un protocole expérimental reproductible a été constitué, calibré sur un cycle précis de 3 minutes d'activité et 1 minute de repos.

Avec un total de 21 traces structurées sous Android et 12 traces sous Linux ciblant des applications graphiques, le projet dispose désormais d'une base de données robuste. Dans une démarche de science ouverte et de reproductibilité, l'intégralité de ces traces brutes a été rendue publique et consultable en libre accès (voir Annexe 7).

Le chapitre suivant exploitera ce jeu de données pour analyser l'évolution du trafic, l'utilisation moderne de l'API Socket, et la résilience des protocoles face aux perturbations réseau, tout en confrontant ces observations à l'étude historique de 2017.

2. Le test a été réalisé sur la réception du flux audio distant suivant : `http://icecast.radiofrance.fr/fip-midfi.mp3`
3. Le fichier téléchargé est l'image officielle `ubuntu-24.04-desktop-amd64.iso` via le réseau BitTorrent.

Chapitre 6

Évolution des Pratiques Réseau (2017-2026)

En étudiant un total de 33 traces issues d'applications très populaires (dont le détail des versions et de l'environnement de test est répertorié au chapitre 5), l'objectif est de comparer le comportement réseau actuel avec celui observé par Vander Schueren [1] en 2017.

Nous avons structuré nos résultats autour de nos deux systèmes d'exploitation. La première grande partie de ce chapitre se concentre sur l'environnement mobile (Android). Nous y regarderons d'abord le volume de trafic pour montrer la montée en puissance du protocole QUIC. Ensuite, nous étudierons comment les applications utilisent l'API Socket aujourd'hui, en particulier pour configurer le réseau localement et gérer la taille des transferts de données. Enfin, nous testerons la résistance de ces applications mobiles face aux coupures réseau fréquentes, comme les changements d'antenne (*Handover*) ou les pertes de signal (*Blackout*).

Dans un second temps, nous passerons à l'écosystème de bureau (Linux). Grâce à nos sondes eBPF, nous verrons comment les applications intègrent les nouvelles interfaces asynchrones (`io_uring`). Surtout, nous pourrons observer directement depuis le noyau comment le système TCP arrive à survivre et à gérer de lui-même une coupure totale du réseau.

6.1 Le trafic Android : Analyse globale

Pour comparer fidèlement ces résultats avec l'étude de 2017 [1], la même méthodologie est appliquée. L'analyse des volumes de données se limite aux scénarios stables (*Baseline*), afin d'éviter les biais liés aux pannes. Comme établi précédemment (cf. Tableau 5.1), cette

analyse s'appuie sur des captures réalisées sous Android (Google Pixel 4a).

6.1.1 Protocoles de transport : Une répartition contrastée du trafic

En 2017, UDP était très peu utilisé pour le transfert de données lourdes. Neuf ans plus tard, nos traces *Baseline* révèlent une mutation majeure, mais loin d'être uniforme. La Figure 6.1 illustre la part du volume de données transitant par TCP et UDP pour nos différentes applications.

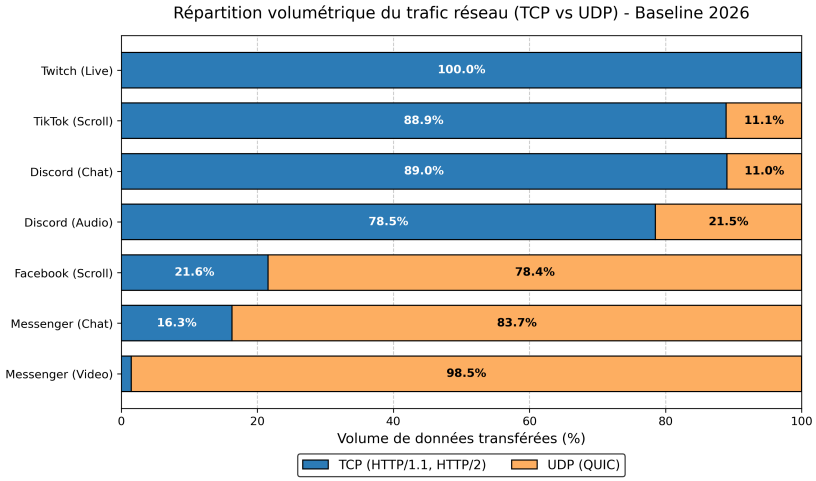

FIGURE 6.1 – Répartition volumétrique du trafic réseau en conditions stables. Le volume d'octets UDP est massif chez Meta, tandis que TCP reste dominant sur les autres plateformes.

Le graphique met en évidence deux profils d'utilisation bien distincts au sein de l'écosystème mobile : d'une part, le maintien des connexions TCP classiques, et d'autre part, un volume massif de données échangées via des sockets UDP. Cette prédominance d'UDP reflète l'adoption généralisée du protocole HTTP/3 (QUIC) par les applications modernes. En effet, bien que la dynamique interne de ce protocole soit invisible pour le noyau, l'intercepteur en espace utilisateur comptabilise parfaitement les volumes de ces transferts UDP.

D'un côté, le groupe Meta a opéré une migration radicale : Messenger Vidéo (98,5 %) et Facebook (78,4 %) réalisent désormais l'écrasante majorité de leurs transferts en UDP. D'un autre côté, des applications très populaires comme TikTok (88,9 % de TCP), Discord et Twitch (100 % de TCP) s'appuient encore massivement sur l'architecture classique. Pour une plateforme de diffusion vidéo en direct comme Twitch, une hypothèse probable est qu'elle privilégie la fiabilité basée sur les standards traditionnels du streaming adaptatif sur HTTP (HTTP/1.1 ou HTTP/2) [30], plutôt que d'entamer une transition complexe vers des architectures entièrement basées sur UDP.

En résumé, bien qu'eBPF soit aveugle à la dynamique interne de HTTP/3, l'approche hybride résout ce problème. La simple comptabilisation des transferts UDP permet d'évaluer

précisément l'adoption globale de QUIC.

6.1.2 Le paradoxe UDP : Sockets muets

En 2017, les sockets UDP représentaient environ 27 % des descripteurs réseau alloués [1]. Dans nos relevés de 2026, cette proportion globale reste très similaire, atteignant 30,8 % (soit 5 823 sockets UDP sur un total de près de 19 000). Toutefois, l'usage réel de ces sockets a évolué de manière extrême.

Notre analyse révèle en effet que 97,5 % de ces descripteurs UDP n'échangent absolument aucune donnée applicative. Si l'étude originelle s'étonnait déjà d'observer ce comportement sur 85 % de ses sockets UDP [1], cette pratique est aujourd'hui devenue la norme absolue.

Cette abondance de descripteurs « muets » est due à une simple astuce d'optimisation système. La documentation du noyau Linux (`netdevice(7)`) impose d'utiliser un socket pour pouvoir interroger l'état matériel du réseau via la commande `ioctl` [31]. Les développeurs créent donc systématiquement un « socket factice » UDP (`SOCK_DGRAM`). Étant sans connexion, c'est le descripteur le plus rapide à allouer pour le noyau, avant d'être immédiatement refermé.

Par conséquent, l'énorme volume de données UDP visible sur notre graphique (chez Meta notamment) transite en réalité sur les 2,5 % de sockets restants (soit à peine 144 descripteurs). Contrairement à TCP où les applications multiplient les connexions parallèles, l'architecture moderne de QUIC centralise tout son trafic via une poignée de sockets UDP [32].

6.2 Évolution de l'API Socket

L'analyse des appels système permet d'observer directement l'évolution de la programmation réseau. Afin de mesurer cette dynamique, les mêmes indicateurs statistiques qu'en 2017 [1] ont été appliqués à l'intégralité du jeu de données Android (incluant à la fois les scénarios *Baseline* et les scénarios de pannes).

Il existe cependant une différence dans notre approche. L'étude d'origine analysait près de 90 applications pour obtenir une vue très large. Nous avons plutôt choisi d'étudier en profondeur un petit groupe de géants du numérique (Meta, TikTok, Twitch, Discord). Bien que moins varié que celui de 2017, ce choix permet d'analyser des applications modernes très populaires et reflétant les pratiques réseau actuelles.

6.2.1 Répartition globale : Le poids de la configuration

En 2017, les appels dédiés à la réception de données (`read()` et ses variantes) dominaient largement l'usage de l'API (49 %) [1]. Comme l'illustre la Figure 6.2, la situation de 2026 montre

un rééquilibrage important au sein des applications modernes.

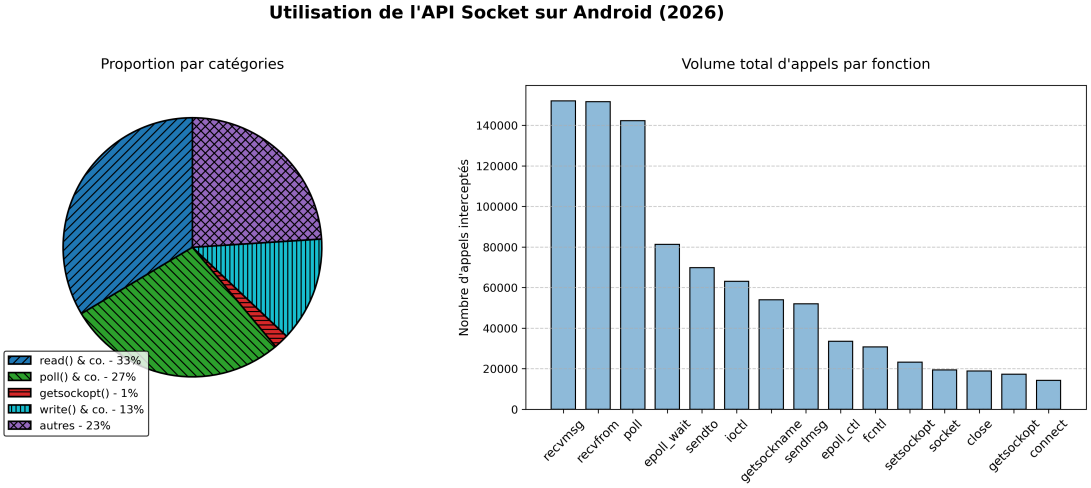

FIGURE 6.2 – Analyse des appels système sur Android (2026). À gauche, la répartition par catégorie montre la diminution de la réception classique au profit des fonctions de configuration (« autres »). À droite, le volume brut par fonction confirme l'usage massif de `ioctl` et `getsockname`.

La proportion des appels de lecture est descendue à 33 %. Cette baisse ne signifie pas que les applications reçoivent moins de données, mais plutôt qu'elles multiplient les appels liés au suivi du réseau.

L'évolution la plus marquante concerne d'ailleurs la catégorie « autres » . Alors qu'elle ne représentait que 6 % en 2017 [1], elle atteint aujourd'hui 23 % du total des appels. Nos données montrent que cette hausse est directement liée aux requêtes de configuration locale. On y retrouve principalement `ioctl` (63 000 appels) pour vérifier l'état des connexions Wi-Fi ou cellulaires, `getsockname` (54 000 appels) pour identifier l'adresse locale de l'appareil, et `fcntl` (30 000 appels) pour activer le mode non-bloquant.

À l'inverse, l'usage de la famille `getsockopt()` s'est effondré, passant de 9 % en 2017 à seulement 1 % en 2026. Cette chute s'explique logiquement par la transition vers QUIC. Puisque ce protocole gère ses connexions et sa congestion au sein de bibliothèques directement intégrées en espace utilisateur, l'état du transport est désormais interrogé via des appels de fonctions internes à ces bibliothèques. Les applications n'ont donc plus besoin de faire appel à la fonction standard `getsockopt()` de la `libc` (interceptée par notre outil) qui servait historiquement à interroger la pile TCP du noyau.

6.2.2 Fréquence d'utilisation par les applications

Au-delà du volume total d'appels, il est intéressant d'observer l'empreinte de chaque fonction, c'est-à-dire le nombre d'applications qui l'utilisent. La Figure 6.3 illustre cette répartition

et permet une comparaison directe avec les habitudes de 2017.

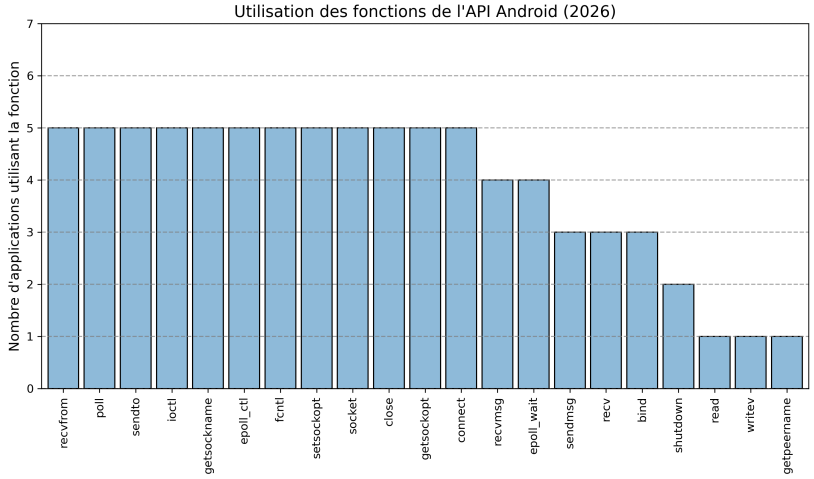

FIGURE 6.3 – Nombre d'applications utilisant chaque fonction de l'API (Android 2026). La réception classique via `read` est remplacée par les fonctions orientées paquets (`recvfrom` et `recvmsg`).

Le changement le plus net concerne les méthodes de lecture. En 2017, la fonction `read()` (utilisée pour les connexions TCP) était présente dans presque toutes les applications. Sur notre figure, on remarque que l'usage de `read()` a presque disparu. À l'inverse, ce sont les fonctions `recvfrom` et `recvmsg` qui dominent le classement. Ce changement s'explique par le passage vers UDP : les applications ne lisent plus un flux continu de données, mais reçoivent des paquets isolés. De plus, nos traces montrent que des options de lecture comme `MSG_ERRQUEUE`, qui n'étaient pas utilisées en 2017, sont maintenant activées pour détecter rapidement les erreurs du réseau et adapter la taille des paquets envoyés.

Pour gérer plusieurs connexions en même temps, la fonction classique `poll()` reste très utilisée. Toutefois, notre figure montre une forte utilisation de la fonction `epoll`. Sous Linux et Android, `epoll` a été spécifiquement conçue pour être une alternative plus rapide et plus moderne à `poll` lorsqu'il faut gérer de nombreux sockets. Cette évolution montre que les applications utilisent aujourd'hui des bibliothèques réseau mieux optimisées pour le système.

6.2.3 Taille des données reçues

En 2017, l'analyse des réceptions révélait des comportements inattendus. Sur TCP, la moitié des transferts se limitait à 5 octets ou moins (dont beaucoup à un seul octet). Sur UDP, on observait déjà une forte concentration à 1350 octets, annonçant l'arrivée de QUIC [1].

Pour observer la situation en 2026, nous avons mesuré le volume exact de chaque transfert réussi lors de nos scénarios stables. La Figure 6.4 présente cette distribution pour TCP et UDP.

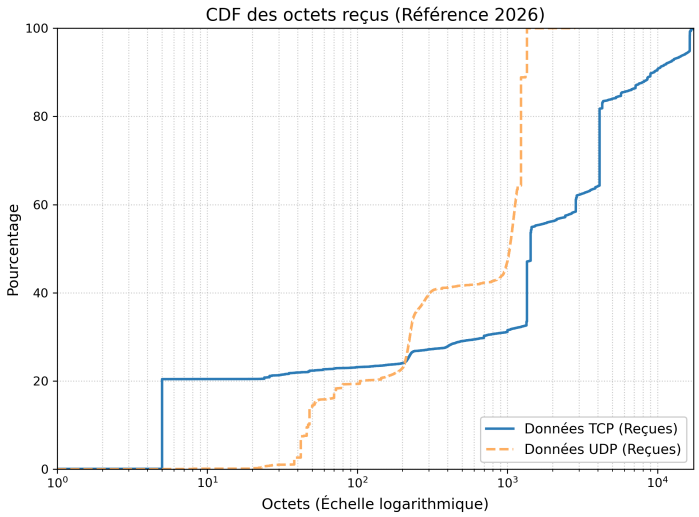

FIGURE 6.4 – Répartition cumulée (CDF) de la taille des données lues par appel système (Android 2026). L'axe horizontal logarithmique souligne des paliers francs à 5 octets pour TCP, et à 1232/1350 octets pour UDP.

Le premier constat est que les lectures inefficaces d'un seul octet ont quasiment disparu. Cependant, le graphique révèle de nouveaux paliers très marqués :

Le mystère des 5 octets sur TCP :
Sur la courbe bleue (TCP), 20 % des appels réceptionnent très exactement 5 octets. En 2017, l'étude originale avait déjà remarqué cette taille spécifique, sans toutefois l'expliquer. Aujourd'hui, nous pouvons formuler une hypothèse technique forte : ces 5 octets correspondent à la taille exacte de l'en-tête de chiffrement défini par le standard TLS (*TLS Record Header*) [33]. Il est très probable que les bibliothèques réseau lisent d'abord ces 5 octets pour connaître la taille exacte du message chiffré, avant de lire le reste des données dans un second temps.

Les limites strictes sur UDP :
Sur la courbe orange (UDP), deux sauts massifs apparaissent à la fin. Contrairement à TCP, UDP gère des paquets indépendants. Ces paliers de 1232 et 1350 octets ne sont pas dus au hasard. Ce sont des limites maximales imposées par le standard QUIC [12] pour éviter que les routeurs d'Internet ne découpent les paquets en cours de route. La taille de 1232 octets correspond très exactement à la limite mathématique du réseau IPv6, tandis que 1350 octets est la valeur de sécurité standard pour les réseaux IPv4.

En résumé, les développeurs d'applications ne s'occupent plus manuellement de la taille de leurs transferts. Ce sont désormais les bibliothèques réseau qui font tout le travail en arrière-plan : elles lisent les données par étapes pour gérer le chiffrement (sur TCP), et elles calibrent la taille des paquets au millimètre près pour respecter les limites physiques d'Internet (sur UDP).

6.3 Résilience réseau : Handover et Blackout

Au-delà de l'utilisation générale de l'API Socket, l'environnement mobile est par nature sujet à de fréquentes perturbations physiques, qu'il s'agisse d'un basculement entre antennes (*Handover*) ou d'une perte totale de signal (*Blackout*). Cette section se concentre spécifiquement sur la résilience des applications face à ces coupures de connectivité.

En 2017, l'outil d'origine se limitait à l'espace utilisateur. Lors de tests simulant des perturbations réseau (comme des *handovers*), cette approche s'était révélée insuffisante pour détecter le moindre changement de comportement applicatif [1]. L'outil restait aveugle à l'état physique de la connexion. Aujourd'hui, en écoutant directement le noyau via `Netlink`, notre version modernisée détecte la milliseconde exacte de la coupure. Cela nous permet d'analyser précisément comment les applications réagissent à ces pannes.

6.3.1 Activité et stress lors de la coupure

En synchronisant nos traces avec l'instant de la panne (T_0), nous pouvons mesurer la réaction immédiate des applications. La Figure 6.5 compare la création de nouveaux sockets et le nombre d'erreurs générées dans les 5 premières secondes suivant un *Handover* ou un *Blackout*. Bien que le *Blackout* dure 15 secondes au total, se concentrer sur cette première fenêtre garantit une comparaison parfaitement équitable entre les deux scénarios.

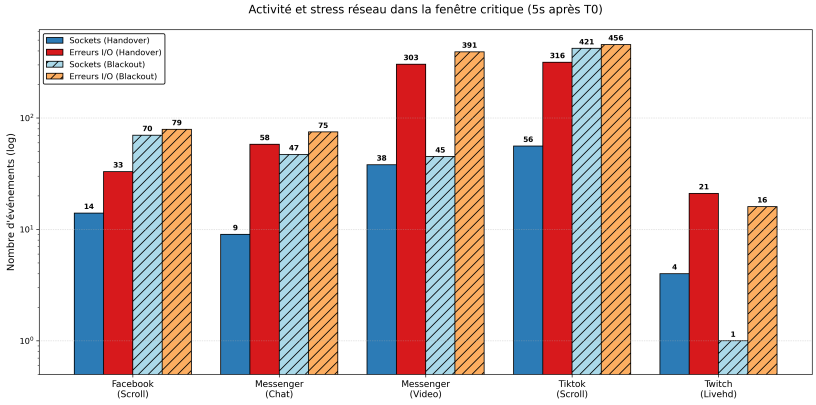

FIGURE 6.5 – Activité et stress réseau durant les 5 premières secondes d'une coupure (T_0). L'axe vertical (logarithmique) souligne l'hyperactivité de certaines applications.

L'analyse de ces graphiques et de nos journaux d'erreurs montre que la patience silencieuse est devenue rare. Seules quelques applications (comme Twitch) se reposent passivement sur TCP, subissant la rupture sans générer de nouveaux appels système. La majorité de l'écosystème réagit immédiatement, mais avec des stratégies radicalement différentes.

D'un côté, on observe une tendance à la force brute. Sur UDP, Messenger Vidéo continue

d'envoyer ses paquets à l'aveugle, générant instantanément des centaines d'erreurs (`ENETUNREACH`). Sur TCP, TikTok entre dans une boucle agressive : l'application crée massivement des sockets (plus de 400 en 5 secondes) pour tenter de se reconnecter, échoue, et recommence frénétiquement.

De l'autre côté, l'écosystème Meta (Facebook, Messenger Chat) adopte une approche beaucoup plus contrôlée. Ces applications utilisent des sockets non-bloquants pour interroger le système sans le surcharger. Lors d'une panne, elles reçoivent de simples erreurs d'attente (`EAGAIN`) mais ne génèrent aucune rafale de requêtes. Elles acceptent l'erreur localement et patientent intelligemment avant de réessayer.

6.3.2 Anatomie d'un Blackout complet

Pour mieux comprendre cette gestion de crise sur la durée, nous avons isolé l'activité de Facebook et TikTok pendant toute la durée d'un *Blackout*. La Figure 6.6 synchronise leurs appels système avec l'état du réseau : la ligne pointillée noire marque la coupure ($T = 0$) et la ligne verte le retour du signal.

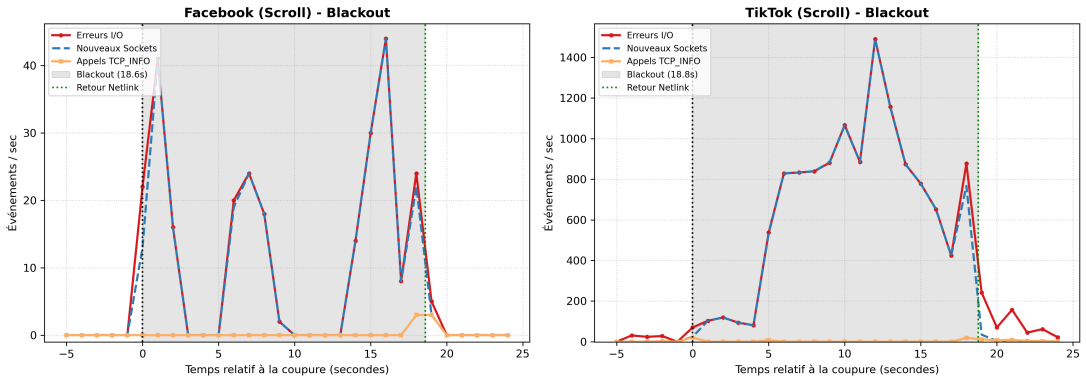

FIGURE 6.6 – Évolution des appels système lors d'un *Blackout* de 18 secondes. À gauche, Facebook patiente. À droite, l'échelle révèle la boucle de panique de TikTok (plus de 12 000 sockets).

Il faut d'abord noter l'immense différence d'échelle entre les deux graphiques. À gauche, Facebook démontre une maîtrise parfaite. Dès la détection de la panne, l'application retire proprement son socket défaillant puis le ferme. Ensuite, elle tente de se reconnecter par petites salves très espacées. En acceptant l'échec et en patientant, Facebook limite sa consommation à seulement 254 sockets sur toute la durée de la panne. On y remarque également quelques requêtes `TCP_INFO` ciblées, confirmant l'utilité de cette option pour surveiller l'état de la connexion.

À droite, TikTok illustre le pire scénario possible. Face à la coupure, l'application s'emballe totalement. Elle crée un socket, exige un envoi immédiat des données en désactivant

l'algorithme de Nagle (`TCP_NODELAY`), tente de se connecter, échoue (`ENETUNREACH`), ferme le socket et recommence l'opération à la microseconde suivante. Sans aucun mécanisme d'attente, cette boucle de panique génère plus de 12 400 sockets et 13 200 erreurs en seulement 15 secondes. L'application sature inutilement le processeur du smartphone pour tenter de précharger la vidéo suivante, illustrant un défaut majeur d'optimisation face à la mobilité.

6.4 L'écosystème Linux : Optimisation et résilience TCP

Après l'étude de l'environnement Android, l'analyse se porte désormais sur l'écosystème de bureau sous Linux. L'objectif est d'observer de manière factuelle comment les applications sélectionnées (détaillées dans le Tableau 5.2) gèrent leurs transferts de données. Grâce aux sondes eBPF, il est possible de mettre en évidence l'utilisation de méthodes d'entrées/sorties très modernes pour optimiser les performances, ainsi qu'une confiance totale accordée au système d'exploitation pour survivre aux pannes réseau.

6.4.1 Vue globale en conditions stables

Pour établir un point de comparaison (*Baseline*), l'analyse porte sur trois applications aux usages très différents : VLC (lecteur multimédia), Telegram (messagerie) et Transmission (téléchargement). La Figure 6.7 illustre leurs différences en comparant les protocoles utilisés et le nombre d'appels système nécessaires pour transférer un mégaoctet (Mo) de données.

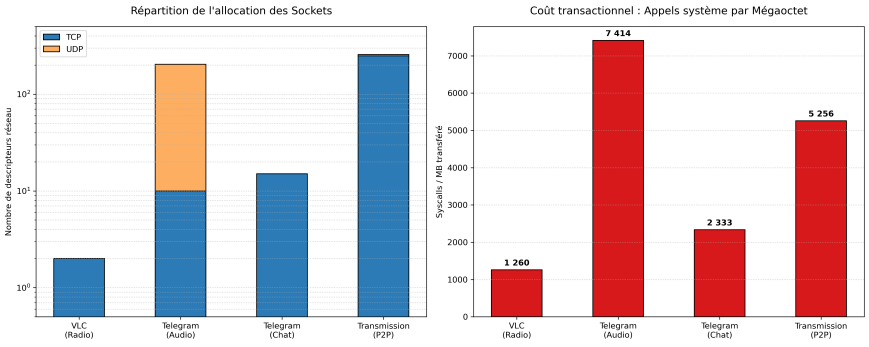

FIGURE 6.7 – Profil réseau en conditions stables. À gauche, les protocoles utilisés. À droite, l'efficacité des applications (nombre d'appels système par Mo transféré).

Ces graphiques révèlent des comportements très variés. VLC est très économe : l'application n'utilise que 2 sockets TCP pour maintenir la réception d'un flux audio distant (*streaming*), gardant le nombre d'appels système très bas (1 260 par Mo). À l'inverse, un appel vocal sur Telegram utilise massivement UDP (194 sockets). Ce choix, typique du temps réel, fragmente les données et demande logiquement beaucoup plus d'efforts au système (7 414 appels par Mo). Cependant, lors de l'envoi de simples messages texte, Telegram redevient classique et bascule sur TCP pour garantir la fiabilité de l'envoi.

Enfin, Transmission illustre parfaitement la dynamique des réseaux *Peer-to-Peer* (P2P) [34]. Pour télécharger 400 Mo de l'image ISO d'Ubuntu, l'application ouvre 249 sockets TCP simultanés. Pourtant, l'analyse des traces montre qu'un unique socket concentre à lui seul 98 % de l'activité, générant plus de 2 millions d'appels système. Cette forte asymétrie s'explique directement par le standard BitTorrent [35], qui privilégie automatiquement la source la plus rapide. Les autres connexions, maintenues pour la découverte du réseau, génèrent un trafic négligeable. Ainsi, malgré un volume absolu impressionnant de 2,11 millions d'appels, l'efficacité globale s'établit à 5 256 appels par Mo. Ce résultat s'explique naturellement par le fonctionnement du P2P : le fichier est découpé en milliers de petits blocs, ce qui multiplie mécaniquement les requêtes de lecture.

L'examen des appels système révèle également un comportement inattendu : l'utilisation de l'API `io_uring` par VLC et Telegram. L'intégration de cette interface d'entrées/sorties à haute performance soulève de nouvelles questions architecturales. Il s'agit à présent de comprendre comment ces applications intègrent ce mécanisme asynchrone moderne à leurs transferts réseau classiques.

6.4.2 io_uring : Comment les applications gèrent la latence ?

Pour comprendre comment cette nouvelle interface `io_uring` fonctionne avec la pile réseau classique, nous avons croisé les actions de l'espace utilisateur avec nos données eBPF. L'utilisation des identifiants de session (`session_id`) permet de relier ces deux niveaux d'exécution de manière très précise.

Nos résultats mettent en évidence deux stratégies d'intégration logicielles distinctes. Telegram adopte une architecture strictement découplée, typique des applications multithreadées modernes. L'intégralité de son trafic réseau est gérée par un processus dédié (Session 1) utilisant des appels synchrones classiques (`read`, `write`, `poll`). En parallèle, ses événements `io_uring` sont isolés sur le processus principal (Session 0). L'application sépare donc hermétiquement son moteur réseau de ses opérations locales (comme la gestion du cache ou de l'interface). À l'inverse, VLC illustre une architecture hybride fortement couplée : le lecteur concentre simultanément les appels réseau et les complétions `io_uring` au sein d'une seule et même session (Session 0).

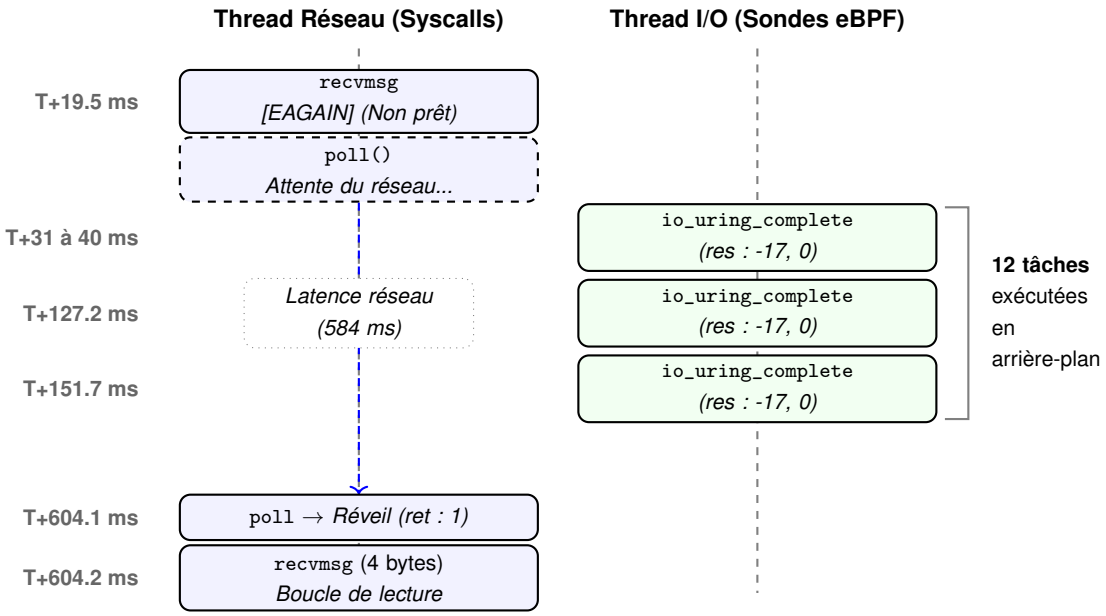

FIGURE 6.8 – Chronologie eBPF de VLC en conditions stables. Face à une latence de 584 ms, le thread réseau se suspend via `poll` (qui renverra `1` dès qu'un socket sera prêt). L'application exploite ce temps mort pour valider 12 opérations locales via `io_uring`. Les codes de retour asynchrones indiquent l'état de l'opération, tel que `-17` pour le code d'erreur standard `EEXIST` (ressource déjà existante).

Cette approche centralisée chez VLC permet d'observer directement l'optimisation des temps morts. Comme l'illustre la Figure 6.8, à la 19e milliseconde, le lecteur tente de réceptionner son flux multimédia, mais les paquets ne sont pas encore arrivés (le noyau renvoyant logiquement l'erreur d'attente `EAGAIN`). Plutôt que d'entrer dans une boucle active, le thread réseau se suspend proprement via la fonction `poll`. Pour un outil de traçage classique, cette période s'étalant jusqu'à la réception effective des données (à 604 ms) apparaîtrait comme une inactivité totale. Pourtant, nos sondes eBPF interceptent l'exécution de plusieurs tâches `io_uring` durant cet intervalle précis. L'application ne monopolise donc pas les ressources du processeur face à la latence du réseau, mais délègue intelligemment le traitement de ses entrées/sorties locales à la file d'attente asynchrone du noyau.

Cette efficacité montre la robustesse de Linux en temps normal. Il reste maintenant à voir comment le système réagit face à une crise majeure. Pour cela, nous allons soumettre notre application la plus lourde, Transmission, à une coupure totale du réseau.

6.4.3 Test de coupure : La survie automatique de TCP

Nous avons soumis le client Transmission à un « trou noir » réseau à l'aide d'`iptables`. Comme les paquets sont discrètement supprimés en chemin (via la règle `DROP`), l'application ne reçoit aucun message d'erreur explicite. C'est donc la pile TCP du noyau Linux qui doit

gérer la crise toute seule.

Les données extraites sont impressionnantes. Pendant la coupure, l'application reste totalement calme et silencieuse. Mais en dessous, nos sondes eBPF capturent une véritable tempête : le noyau tente de sauver les connexions en effectuant 1 825 retransmissions TCP réparties sur plus de 300 sockets en même temps. Ce calme de l'application s'oppose totalement à la panique que nous avions observée sur Android avec TikTok. Sur Linux, l'application fait une confiance absolue au système d'exploitation.

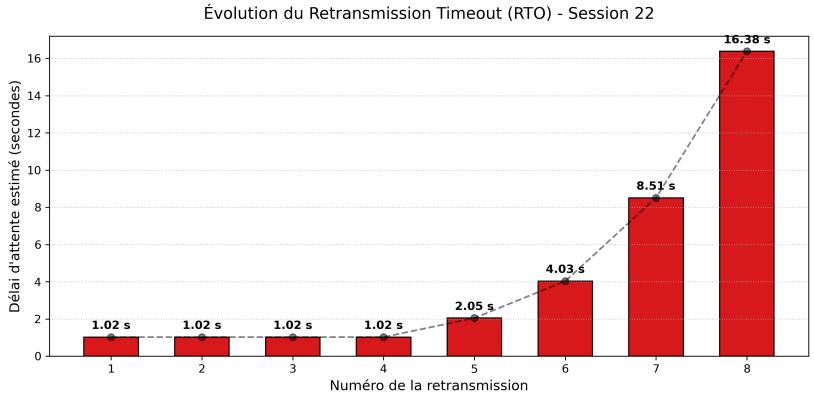

FIGURE 6.9 – Évolution du délai d'attente avant retransmission (RTO) pour un socket lors de la coupure. Le noyau double mathématiquement son temps d'attente entre chaque essai.

Pour vérifier la précision de cette réaction, nous avons isolé l'un de ces sockets (la Session 22) pendant sa longue coupure. La Figure 6.9 montre le temps que le noyau attend entre chaque tentative de renvoi. L'outil eBPF confirme que Linux respecte à la lettre les standards officiels de contrôle de congestion TCP [36]. Dès qu'il détecte la perte, il réduit sa fenêtre d'envoi (`snd_cwnd`) au minimum pour ne pas saturer le réseau.

Surtout, la chronologie met en évidence l'application stricte de l'algorithme d'attente exponentielle (*Exponential Backoff*), tel qu'exigé par le standard officiel RFC 6298 [37]. Après de premiers essais espacés d'une seconde, le temps de temporisation double mathématiquement à chaque échec : 2 secondes, puis 4, puis 8, pour finalement atteindre 16 secondes. Cette progression valide la conformité absolue du noyau face aux pannes. Elle confirme surtout l'intérêt de TCPSnitch : sans les sondes eBPF pour observer directement le noyau, ce mécanisme interne serait resté totalement invisible.

6.5 Conclusion

L'analyse des traces révèle une évolution radicale des pratiques réseau. Sur l'écosystème mobile, l'adoption massive de QUIC déplace la gestion de la complexité vers l'espace utilisateur. Si cette flexibilité profite à des acteurs comme Meta, qui gèrent les déconnexions de

manière très contrôlée, elle expose d'autres applications (telles que TikTok) à des boucles de requêtes excessives, particulièrement coûteuses pour les ressources du smartphone lors d'une perte de signal.

À l'inverse, l'écosystème Linux démontre une fiabilité remarquable. Les applications de bureau délèguent entièrement la gestion du réseau au système d'exploitation : elles adoptent des interfaces asynchrones modernes (`io_uring`) pour masquer la latence, et s'en remettent à la robustesse de la pile TCP du noyau pour gérer les pannes prolongées. Surtout, ces découvertes valident l'approche technique développée : sans le couplage de l'interception classique avec les sondes eBPF, les mécanismes internes de résilience du système seraient restés totalement invisibles. Le chapitre suivant s'attachera à prendre du recul sur ces résultats et à discuter des perspectives d'évolution de l'outil.

Chapitre 7

Discussion et Perspectives

L'objectif de ce mémoire était d'actualiser TCPSnitch pour pallier l'aveuglement progressif des outils de traçage limités à l'espace utilisateur. Les résultats obtenus confirment qu'une approche hybride est aujourd'hui indispensable pour appréhender la complexité des communications modernes.

Évolution des pratiques : La montée en puissance d'UDP

La comparaison de nos nouvelles traces avec celles de 2017 [1] confirme certaines similitudes, mais révèle surtout une transformation majeure des communications.

Le changement le plus frappant est l'explosion de l'utilisation du protocole UDP sur mobile. Alors qu'il était minoritaire en 2017, son usage a massivement progressé sous l'impulsion du protocole QUIC (HTTP/3) [12]. On observe également que la grande majorité des sockets UDP créés n'échangent aucune donnée : ils servent uniquement de "sonde" rapide pour lire la configuration matérielle du réseau (via l'appel `ioctl`) avant d'être refermés [31].

Les tests de coupure révèlent aussi un fossé de plus en plus grand entre les plateformes :

- **Sur Android :** La gestion des pannes repose désormais sur les applications elles-mêmes, avec des succès très inégaux. D'un côté, Meta maîtrise parfaitement le protocole QUIC et gère les coupures calmement. De l'autre, des applications restées sur TCP (comme TikTok) réagissent très mal : au lieu de laisser le noyau gérer l'interruption, elles saturent inutilement le processeur en tentant de se reconnecter en boucle (*busy polling*).
- **Sur Linux :** Les applications de bureau restent fidèles au protocole classique TCP et

font totalement confiance au système. Grâce aux sondes eBPF, il a été possible de prouver que le noyau Linux gère parfaitement ces pannes de manière autonome, tout en permettant aux applications d'utiliser des outils modernes comme `io_uring` pour masquer la latence.

Les limites d'une observation de l'intérieur

Malgré ces réussites, la mise à l'épreuve de cette nouvelle architecture a également mis en évidence ses limites. Le choix technique d'injecter l'outil directement à l'intérieur de l'application (via `LD_PRELOAD`) le rend particulièrement vulnérable.

Par exemple, les règles de sécurité strictes des smartphones récents (SELinux) bloquent l'accès au noyau, empêchant le chargement des sondes eBPF. De plus, si l'application décide de fermer brutalement et massivement ses fichiers réseau (via l'appel `close_range`), elle détruit involontairement les journaux d'enregistrement de l'outil, provoquant des erreurs sous forte charge. Ces limites prouvent qu'il devient de plus en plus difficile d'observer un système uniquement depuis l'espace utilisateur.

Perspectives d'avenir

Si cette modernisation remplit ses objectifs fondamentaux, elle ouvre la voie à de nombreuses évolutions. Dans l'esprit open-source du projet, le code a été rendu public pour encourager la communauté à poursuivre ces travaux :

- **Contourner les sécurités :** Pour fonctionner sur les smartphones Android récents, il sera nécessaire d'initialiser la couche eBPF depuis un processus externe disposant des privilèges administrateur (root) et des capacités Linux requises, telles que `CAP_BPF` [27]. Cette approche permettra d'injecter les sondes sans subir le blocage imposé par le *sandboxing* de l'application surveillée.
- **Restaurer l'observatoire web :** L'outil de 2017 s'appuyait sur le site `tcpsnitch.org`, aujourd'hui inactif. Un beau défi consisterait à moderniser ce serveur central pour qu'il puisse traiter les nouvelles traces hybrides (eBPF et Netlink) et recréer une grande base de données publique.
- **Suivre les nouveaux protocoles :** Le réseau évolue très vite. Face à l'émergence de protocoles à très faible latence comme WebTransport [38], il sera nécessaire d'adapter continuellement l'outil pour éviter qu'il ne devienne aveugle.
- **Explorer d'autres systèmes :** Porter l'outil sur macOS permettrait de comparer comment d'autres environnements gèrent les transferts et la congestion réseau par rapport à Linux.

En définitive, ce mémoire démontre que pour comprendre et optimiser les réseaux de demain, l'analyse ne peut plus s'arrêter à la surface de l'application. Naviguer au plus près des mécanismes internes du noyau est désormais une nécessité absolue.

Bibliographie

[1] Grégory VANDER SCHUEREN. « TCPSnitch : Dissecting the Socket API Usage ». Mém. de mast. Université catholique de Louvain, 2017. URL : https://thesis.dial.uclouvain.be/entities/masterthesis/d16be92e-3c80-468a-9cdb-9c007cc5c813.

[2] Jens AXBOE. *Efficient IO with io_uring*. Rapp. tech. Kernel.dk, 2019. URL : https://kernel.dk/io_uring.pdf.

[3] Grégory VANDER SCHUEREN. *TCPSnitch*. 2017. URL : https://github.com/GregoryVds/tcpsnitch.

[4] Andi KLEEN, Hormuzd M. KHOSRAVI, Alexey KUZNETSOV et Jamal Hadi SALIM. *Linux Netlink as an IP Services Protocol*. RFC 3549. Juill. 2003. DOI : 10.17487/RFC3549. URL : https://www.rfc-editor.org/info/rfc3549.

[5] EBPF PROJECT. *What is eBPF ?* URL : https://ebpf.io/fr-fr/what-is-ebpf/.

[6] LINUX KERNEL ORGANIZATION. *BPF Type Format (BTF)*. URL : https://docs.kernel.org/bpf/btf.html.

[7] Andrii NAKRYIKO. *BPF Portability and CO-RE*. URL : https://nakryiko.com/posts/bpf-portability-and-co-re/.

[8] Andrii NAKRYIKO. *BPF ring buffer*. URL : https://nakryiko.com/posts/bpf-ringbuf/.

[9] LINUX KERNEL NEWBIES. *Linux 5.9*. URL : https://kernelnewbies.org/Linux_5.9.

[10] *splice(2) — Linux manual page*. URL : https://man7.org/linux/man-pages/man2/splice.2.html.

[11] *sendfile(2) — Linux manual page*. URL : https://man7.org/linux/man-pages/man2/sendfile.2.html.

[12] Jana IYENGAR et Martin THOMSON. *QUIC : A UDP-Based Multiplexed and Secure Transport*. RFC 9000. Mai 2021. DOI : 10.17487/RFC9000. URL : https://www.rfc-editor.org/info/rfc9000.

[13] MOZILLA FOUNDATION. *Security/Sandbox/Seccomp*. URL : https://wiki.mozilla.org/Security/Sandbox/Seccomp.

[14] Paul CHAIGNON. *Comprendre eBPF, ses applications et ses limites*. URL : https://pchaigno.github.io/assets/BreizhCamp%202023%20eBPF.pdf.

[15] Steven MCCANNE, Van JACOBSON et al. « The BSD packet filter : A new architecture for user-level packet capture. » In : *USENIX winter*. T. 46. 1993, p. 259-270.

[16] Yoann GHIGOFF, Julien SOPENA, Kahina LAZRI, Antoine BLIN et Gilles MULLER. « BMC : Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing ». In : *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, avr. 2021, p. 487-501. ISBN : 978-1-939133-21-2. URL : https://www.usenix.org/conference/nsdi21/presentation/ghigoff.

[17] Elazar GERSHUNI et al. « Simple and precise static analysis of untrusted linux kernel extensions ». In : *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, p. 1069-1084.

[18] CILIUM PROJECT AUTHORS. *BPF Architecture*. URL : https://docs.cilium.io/en/stable/reference-guides/bpf/architecture/#maps.

[19] BPFTRACE CONTRIBUTORS. *bpftrace*. URL : https://github.com/bpftrace/bpftrace.

[20] LINUX MAN-PAGES PROJECT. *close_range(2) — Linux manual page*. URL : https://man7.org/linux/man-pages/man2/close_range.2.html.

[21] Lennart POETTERING. *File Descriptor Limits*. URL : https://0pointer.net/blog/file-descriptor-limits.html.

[22] LIBBPF COMMUNITY. *bpftool : Tool for inspection and simple manipulation of eBPF programs*. URL : https://github.com/libbpf/bpftool.

[23] LINUX MAN-PAGES PROJECT. *Linux Programmer's Manual : BYTEORDER(3)*. URL : https://man7.org/linux/man-pages/man3/byteorder.3.html.

[24] CPPREFERENCE.COM. *C reference : union declaration*. URL : https://en.cppreference.com/w/c/language/union.

[25] Robin MARX, Wim LAMOTTE, Jonas REYNDERS, Kevin PITTEVILS et Peter QUAX. « Towards QUIC debuggability ». In : *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. 2018, p. 1-7.

[26] ANDROID OPEN SOURCE PROJECT. *The Android Application Sandbox*. URL : https://source.android.com/docs/security/app-sandbox.

[27] LINUX MAN-PAGES PROJECT. *capabilities(7) — Linux manual page*. URL : https://man7.org/linux/man-pages/man7/capabilities.7.html.

[28] Mingyang LI et al. « Enabling Robust Android Malicious Packet Capturing and Detection via Android Kernel ». In : *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2024, p. 1022-1028. DOI : 10.1109/TrustCom63139.2024.00147.

[29] Idio GUARINO, Domenico CIUONZO, Antonio MONTIERI et Antonio PESCAPÈ. « MirageApp×Act-2024 : A Novel Dataset for Mobile App and Activity Traffic Analysis ». In : *2024 20th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 2024, p. 663-666. DOI : 10.1109/WiMob61911.2024.10770459.

[30] Abdelhak BENTALEB, Bayan TAANI, Ali C. BEGEN, Christian TIMMERER et Roger ZIMMERMANN. « A Survey on Bitrate Adaptation Schemes for Streaming Media Over HTTP ». In : *IEEE Communications Surveys & Tutorials* 21.1 (2019), p. 562-585. DOI : 10.1109/COMST.2018.2862938.

[31] *netdevice(7) - Linux manual page*. Linux man-pages project. URL : https://man7.org/linux/man-pages/man7/netdevice.7.html.

[32] Adam LANGLEY et al. « The QUIC Transport Protocol : Design and Internet-Scale Deployment ». In : *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA : Association for Computing Machinery, 2017, p. 183-196. ISBN : 9781450346535. DOI : 10.1145/3098822.3098842. URL : https://doi.org/10.1145/3098822.3098842.

[33] Eric RESCORLA. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Août 2018. DOI : 10.17487/RFC8446. URL : https://www.rfc-editor.org/info/rfc8446.

[34] IAB et Gonzalo CAMARILLO. *Peer-to-Peer (P2P) Architecture : Definition, Taxonomies, Examples, and Applicability*. Rapp. tech. 5694. Nov. 2009. 26 p. DOI : 10.17487/RFC5694. URL : https://www.rfc-editor.org/info/rfc5694.

[35] Bram COHEN. *The BitTorrent Protocol Specification*. BitTorrent Enhancement Proposal 0003. BitTorrent.org, 2008. URL : http://bittorrent.org/beps/bep_0003.html.

[36] Ethan BLANTON, Dr. Vern PAXSON et Mark ALLMAN. *TCP Congestion Control*. RFC 5681. Sept. 2009. DOI : 10.17487/RFC5681. URL : https://www.rfc-editor.org/info/rfc5681.

[37] Matt SARGENT, Jerry CHU, Dr. Vern PAXSON et Mark ALLMAN. *Computing TCP's Retransmission Timer*. Rapp. tech. 6298. Juin 2011. 11 p. DOI : 10.17487/RFC6298. URL : https://www.rfc-editor.org/info/rfc6298.

[38] Alan FRINDELL, Eric KINNEAR et Victor VASILIEV. *WebTransport over HTTP/3*. Internet-Draft draft-ietf-webtrans-http3-15. Work in Progress. Internet Engineering Task Force, mars 2026. 40 p. URL : https://datatracker.ietf.org/doc/draft-ietf-webtrans-http3/15/.

Annexes

Artefacts de reproductibilité

Dans une démarche de science ouverte et afin de garantir la reproductibilité totale des expériences présentées dans ce mémoire, l'ensemble des données brutes a été archivé et rendu public.

Code source de TCPSnitch

L'intégralité du code source de la nouvelle architecture hybride (couches eBPF, `LD_PRELOAD`, Netlink), ainsi que le tableau de bord d'analyse visuelle, est hébergée sur la plateforme GitHub :

- **Dépôt de développement :** `https://github.com/mnietona/tcpsnitch/tree/fix/compilation-modern-linux`

Jeu de données (*Dataset*) 2026

Les 33 traces brutes (Android et Linux), totalisant les expériences en conditions nominales (*Baseline*) et dégradées (*Handover*, *Blackout*), sont regroupées sous la forme d'une archive (*Release*) :

- **Archive des traces :** `https://github.com/mnietona/tcpsnitch/releases/tag/v1.0-dataset`

Déclaration relative à l'utilisation de l'intelligence artificielle

Des outils d'intelligence artificielle générative (tels que ChatGPT, Gemini et DeepSeek) ont été employés durant la réalisation de ce mémoire. Le contenu final, les analyses et les conclusions restent le fruit d'un travail strictement personnel dont j'assume l'entière responsabilité.

Ces outils m'ont accompagné ponctuellement pour les tâches suivantes :

- **Aide à la rédaction :** Les IA m'ont servi d'assistants linguistiques afin de corriger la grammaire, d'améliorer le style et de reformuler certains passages pour en optimiser la clarté.
- **Aide à la recherche :** J'ai interagi avec ces modèles pour m'aider à décortiquer, synthétiser et mieux vulgariser certains articles scientifiques complexes lors de mes recherches.
- **Aide technique et développement :** L'IA a joué un rôle de support lors de la phase pratique. Elle m'a d'abord aidé à m'approprier le code source originel de TCPSnitch (datant de 2017), puis m'a accompagné pour structurer et débugger certaines portions de la nouvelle implémentation.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**