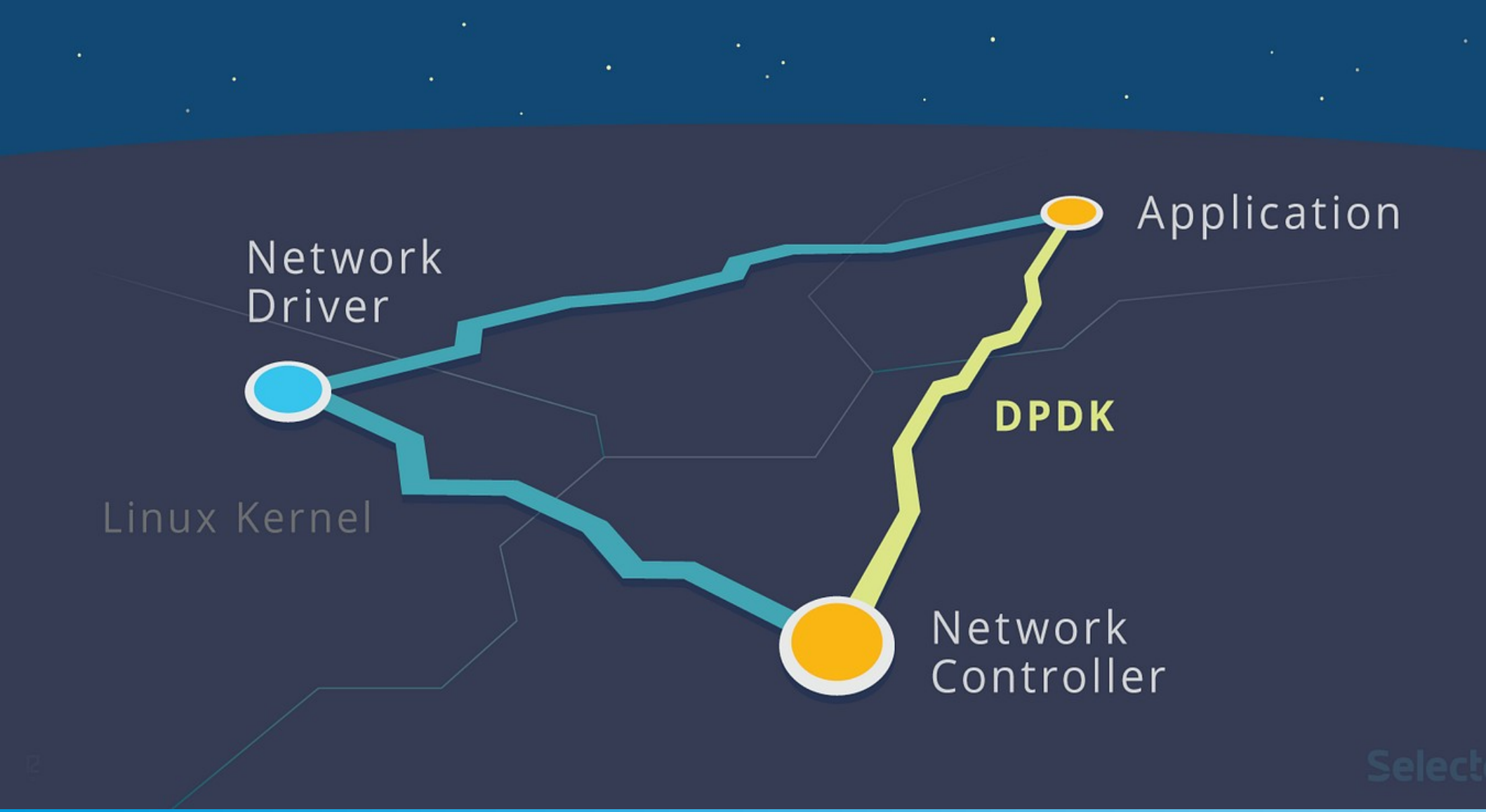

Application
Network
Driver
DPDK
Linux Kernel
Network
Controller

DPDK

DATA PLANE DEVELOPMENT KIT

Le Mans / 13/12/2025

Thierry GAYET

https://linuxmaine.org

PLAN

- Introduction à DPDK
- Usages
- Liste des cartes réseaux compatibles
- Compilation
- Comment isoler une NIC du kernel
- DPDK EAL : le cœur du framework
- Gestion mémoire
- PMD (Poll Mode Drivers) & NIC Offloads
- RX/TX Queues & Pipelines
- La Flow API : SDN et offload matériel
- Optimisations CPU, NUMA, cache & préfetch
- Mesure des performances
- Étude d’un programme DPDK complet

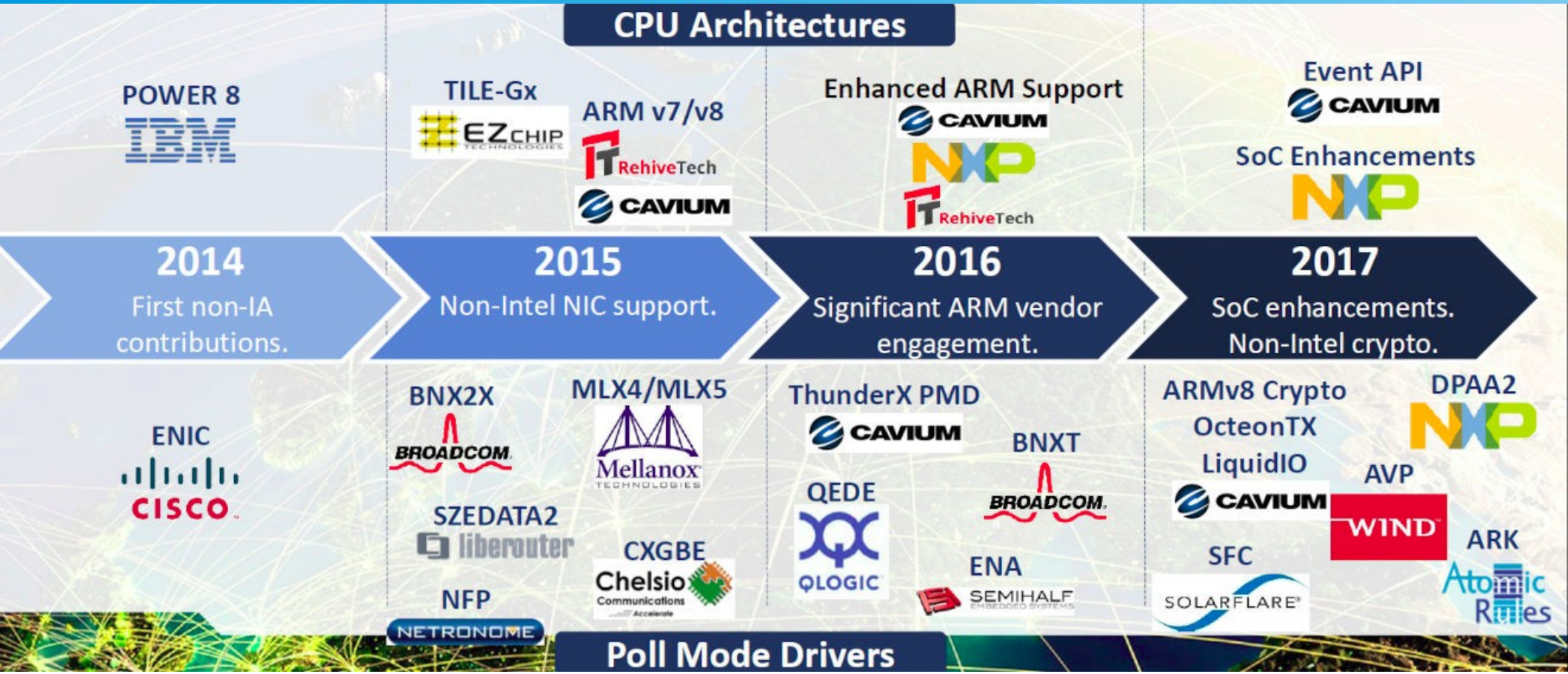
- NUMA et multi-socket
- Architecture de référence haute performance (100Mpps)
- Annexes
- Exemples de codes
- Details
- Travaux pratique (TP/LABS)
- Outils
- Glossaire
- Aide complementaire

Introduction à DPDK

DPDK, un framework simple et complet pour le traitement rapide des paquets.

Historique

- **Origine chez Intel (2010)** : DPDK a été créé en 2010 chez Intel par des ingénieurs de cette société, notamment sous la direction de Venky Venkatesan, souvent considéré comme le “père de DPDK”.
- **Ouverture du projet (2013)** : En 2013, l’équipe a établi une communauté open source autour de DPDK avec la création du site DPDK.org, portée initialement avec l’aide de 6WIND pour fédérer plus largement les contributeurs.
- **Transfert à la Linux Foundation (2017)** : En 2017, le projet DPDK est devenu un projet officiel de The Linux Foundation, ce qui a donné à la communauté un cadre neutre pour la gouvernance et la collaboration entre nombreux contributeurs industriels (Intel, Red Hat, Arm, etc.).
- Depuis, la communauté n'a cessé de croître, tant en nombre de contributeurs et de correctifs qu'en nombre d'organisations participantes.
- Cinq versions majeures ont été publiées, avec la contribution de plus de **160 personnes** issues de **25 organisations différentes**.
- DPDK prend désormais en charge toutes les principales architectures de processeurs et cartes réseau de nombreux fabricants, ce qui le rend idéal pour les applications nécessitant une portabilité multiplateforme.


CPU Architectures
POWER 8
IBM
TILE-Gx
EZCHIP
ARM v7/v8
RehiveTech
CAVIUM
Enhanced ARM Support
CAVIUM
NXP
RehiveTech
Event API
CAVIUM
SoC Enhancements
NXP
2014
First non-IA contributions.
2015
Non-Intel NIC support.
2016
Significant ARM vendor engagement.
2017
SoC enhancements. Non-Intel crypto.
ENIC
CISCO
BNX2X
BROADCOM
SZEDATA2
liberouter
NFP
NETRONOME
MLX4/MLX5
Mellanox
CXGBE
Chelsio
Communications
ThunderX PMD
CAVIUM
QEDE
QLOGIC
BNXT
BROADCOM
ENA
SEMIHALF
ARMv8 Crypto
OcteonTX
LiquidIO
CAVIUM
DPAA2
NXP
AVP
WIND
ARK
Atomic Rules
SFC
SOLARFLARE
Poll Mode Drivers

Open source projects based on DPDK

- **ANS**– Accelerated Network Stack
- **BESS**– Berkeley Extensible Software Switch
- **Butterfly**– Connects Virtual Machines
- **Catnip**– TCP Stack in Rust
- **DPVS**– Layer-4 load balancer
- **dperf** – Network load tester
- **FastClick**– Highspeed dataplane
- **F-Stack**– TCP Stack
- **Gatekeeper** – DDos Protection System
- **IMTL** – Real time and low latency media transport library
- **Lagopus**– software OpenFlow 1.3 switch
- **Metronome** – adaptive packet retrieval in DPDK
- **MoonGen**– Packet generator
- **mTCP**– User-level TCP Stack
- **OPNFV**– Open Platform for NFV
- **OpenDataPlane** – Open DataPlane DPDK platform implementation
- **Open vSwitch**– Multilayer Open Virtual Switch
- **Packet-journey**– Userland router
- **Pktgen-dpdk**– Packet generator
- **PcapPlusPlus**– C++ packet parsing framework
- **Ruru**– Real-time TCP latency monitoring
- **Seastar**– open-source C++ framework
- **SPDK**– Storage Performance Development Kit
- **TLDK**– TCP Stack
- **TRex**– Stateful Traffic Generator
- **VPP**– Fast Data Project
- **WARP17**– Stateful Traffic Generator
- **YANFF**– NFF-Go -Network Function Framework for GO (former YANFF)
- **YAStack**– DPDK with L7 Envoy Proxy

Qu’est-ce que DPDK ?

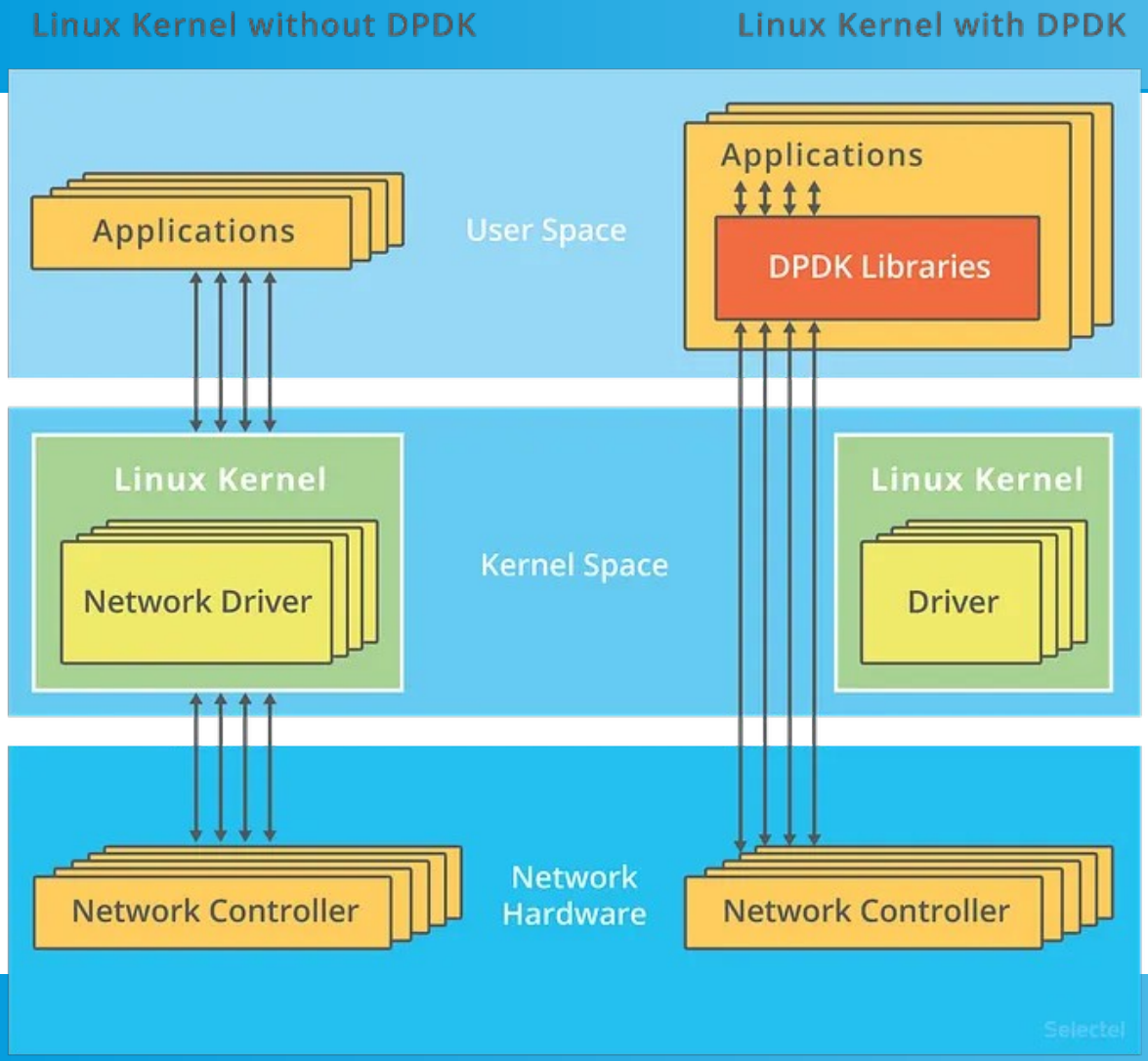
DPDK (Data Plane Development Kit) est un framework user-space conçu pour réaliser du packet processing intensif avec :

- Accélérer le data plane, c'est-à-dire la partie du réseau qui traite réellement les paquets (forwarding, switching, firewalling, load balancing, etc.), par opposition au control plane (routes, configuration…).
- latence ultra-faible (<10 µs),
- throughput très élevé (40–400 Gbps selon NIC),
- utilisation massive du CPU,
- contournement complet du kernel networking stack.

DPDK remplace l’infrastructure suivante du kernel Linux :

Composant Kernel	Équivalent DPDK
Netfilter/iptables	Flow API
Kernel NIC driver	Poll Mode Drivers (PMD)
Sock API	Zero-copy mbuf buffers
Interrupt ISR	Polling busy-loop

DPDK n'est pas une API réseau classique : c’est une bibliothèque bas niveau de pipeline brut.

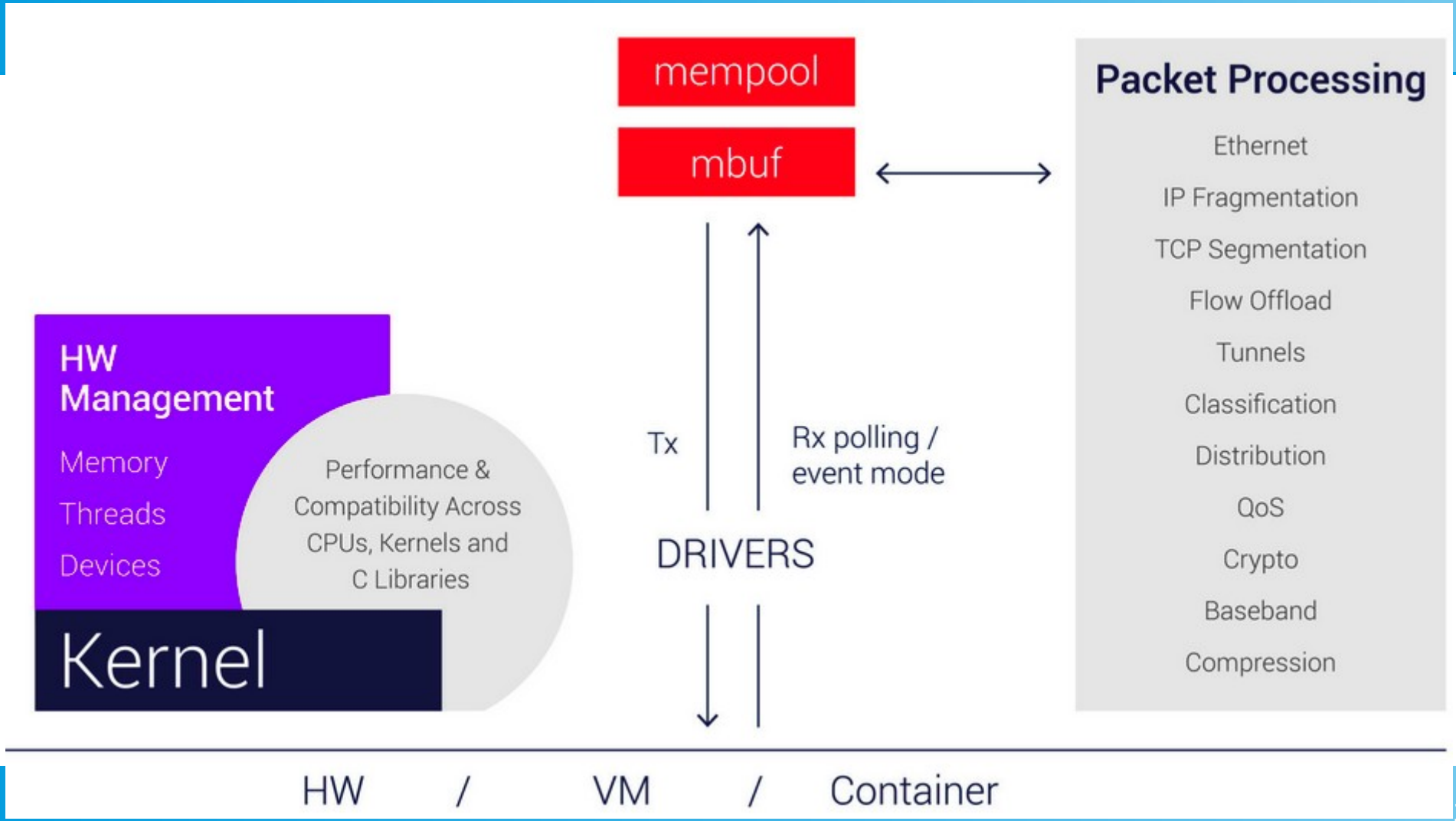



```
+---------------------------+
|   Your Application        |
+-------------+-------------+
              |
              v
+-------------+-------------+
|   DPDK (RTE Runtime)      |
|  - EAL                     |
|  - Mempools                |
|  - Mbufs                   |
|  - Rings/RWLocks           |
|  - Timers/Logs             |
|  - PMD Drivers             |
+-------------+-------------+
              |
              v
+-------------+-------------+
|    NIC (Intel, Mellanox)  |
|    + Poll Mode Driver     |
+---------------------------+
```

Architecture

DPDK repose essentiellement sur un ensemble de bibliothèques et de pilotes assurant un traitement rapide des paquets dans divers environnements matériels et logiciels. Ce framework inclut une couche d'abstraction d'environnement (EAL) qui masque les spécificités matérielles des applications, permettant ainsi aux développeurs d'écrire du code portable et performant sur différentes plateformes, notamment les processeurs x86, ARM et PowerPC :

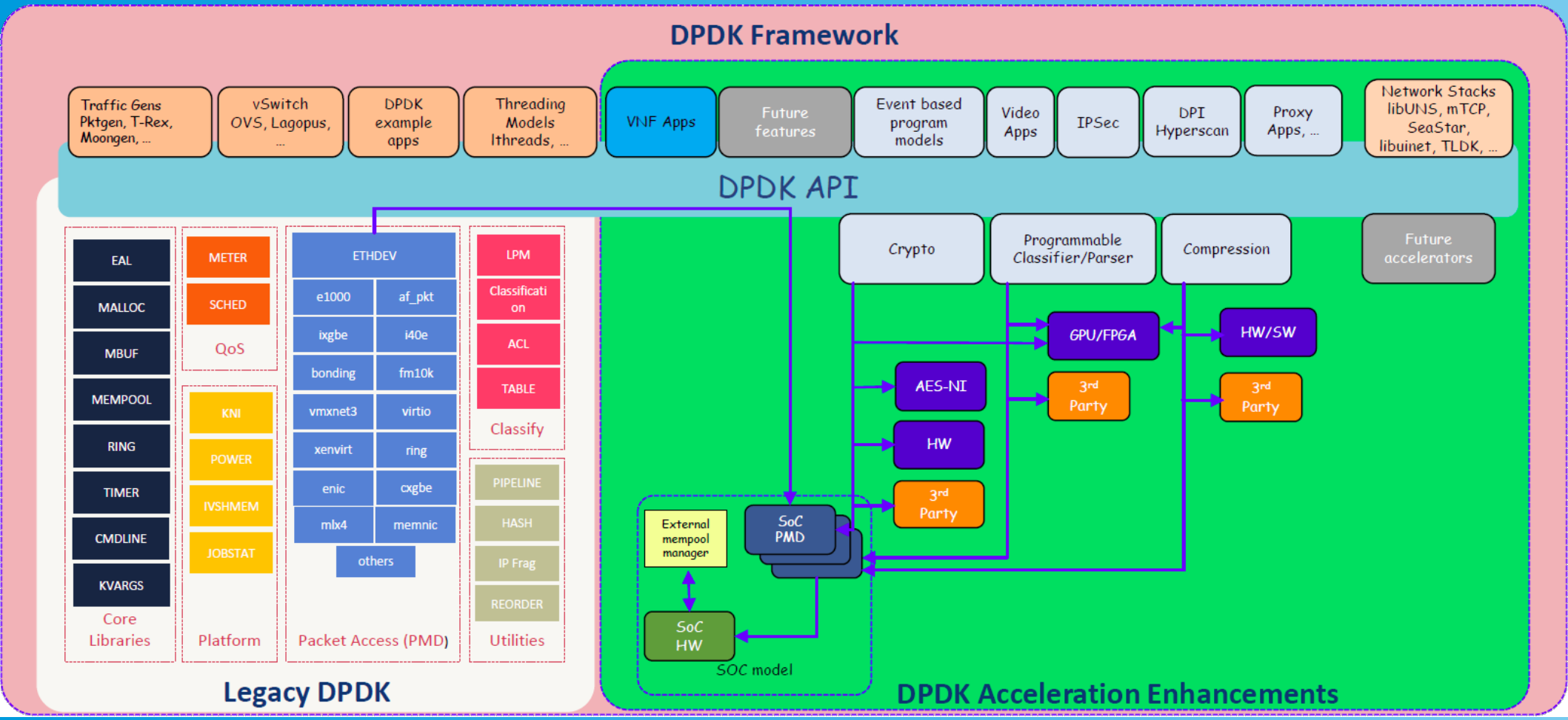
- **EAL (Couche d'abstraction d'environnement)** : Fournit une interface de base entre les applications DPDK et le matériel sous-jacent, en masquant les spécificités du système d'exploitation et les différences matérielles.
- **Gestion de la mémoire** : Inclut la prise en charge des pages énormes, des pools de mémoire et la gestion des tampons, éléments essentiels pour un traitement efficace des paquets.
- **Pilotes en mode interrogation (PMD)** : Ce sont des pilotes optimisés pour diverses interfaces réseau, contournant la pile réseau du noyau afin de réduire la latence et d'augmenter le débit.
- **Tampons circulaires** : Utilisés pour des mécanismes de mise en file d'attente efficaces, permettant une communication interprocessus à haut débit.
- **API pour le traitement des paquets** : Offre un ensemble de fonctions et de bibliothèques pour la manipulation des paquets, notamment l'analyse des en-têtes, la classification et le transfert des paquets.
- **Cryptographie et sécurité** : Fournit des bibliothèques et des pilotes pour la prise en charge des opérations cryptographiques et des communications sécurisées.
- **Développement événementiel et gestion du temps** : Pour la programmation événementielle et les fonctionnalités de gestion du temps, facilitant la planification et l’exécution des tâches dans les délais impartis.


mempool
mbuf
Packet Processing
Ethernet
IP Fragmentation
TCP Segmentation
Flow Offload
Tunnels
Classification
Distribution
QoS
Crypto
Baseband
Compression
HW Management
Memory
Threads
Devices
Performance & Compatibility Across CPUs, Kernels and C Libraries
Kernel
Tx
Rx polling / event mode
DRIVERS
HW / VM / Container

Architecture

En tirant parti des bibliothèques DPDK, les développeurs peuvent créer des chemins de traitement de paquets optimisés, gérer des temporisateurs pour l'exécution asynchrone de fonctions et utiliser une large gamme de pilotes et de bibliothèques conçus pour un traitement rapide des paquets :

- **librte_eal** : Couche d'abstraction de l'environnement : Fournit l'API de base pour DPDK, facilitant l'accès aux ressources matérielles telles que la mémoire, les temporisateurs et les journaux.
- **librte_mempool** : Gestionnaire de pool de mémoire : Gère les pools de mémoire pour une gestion efficace et rapide des paquets.
- **librte_ring** : Gestionnaire de tampons circulaires : Implémente des files d'attente FIFO sans verrou, permettant une communication à haut débit entre les différents composants DPDK.
- **librte_mbuf** : Gestion des tampons de paquets : Gère les tampons de paquets, essentiels à la transmission et à la réception des paquets.
- **librte_ethdev** : API de périphérique Ethernet : Offre une API pour la configuration et l'interrogation des périphériques Ethernet. Elle prend en charge diverses opérations, notamment l'envoi et la réception de paquets. librte_net – Bibliothèque d'assistance réseau : Fournit des API d'assistance pour la gestion des protocoles réseau.
- **librte_ip_frag** : Fragmentation et réassemblage IP : Gère la fragmentation et le réassemblage des paquets IP, compatible avec IPv4 et IPv6.
- **librte_kni** : Interface réseau du noyau : Facilite la communication entre les applications DPDK et la pile réseau du noyau Linux ; principalement utilisée pour le débogage ou l'interaction avec les services réseau Linux existants.



https://benisontech.com/the-origins-and-modern-day-applications-for-ebpf-linux-development/

Usages
(a quoi peut servir DPDK ?)

Routeurs et commutateurs logiciels très rapides (vRouter / vSwitch)

DPDK est largement utilisé pour créer des routeurs et switchs virtuels capables d'atteindre des dizaines de millions de packets/s.

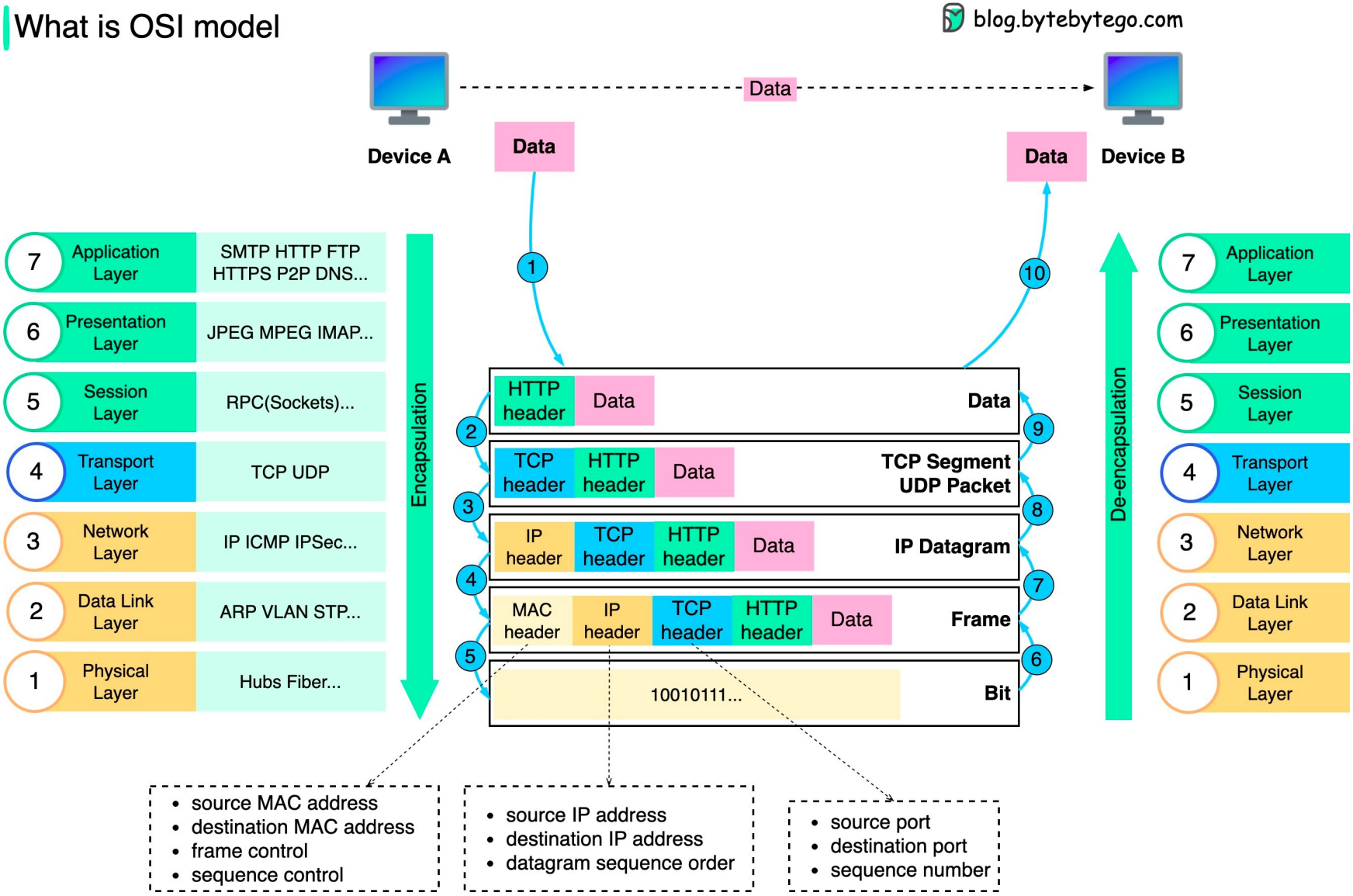
Exemples :

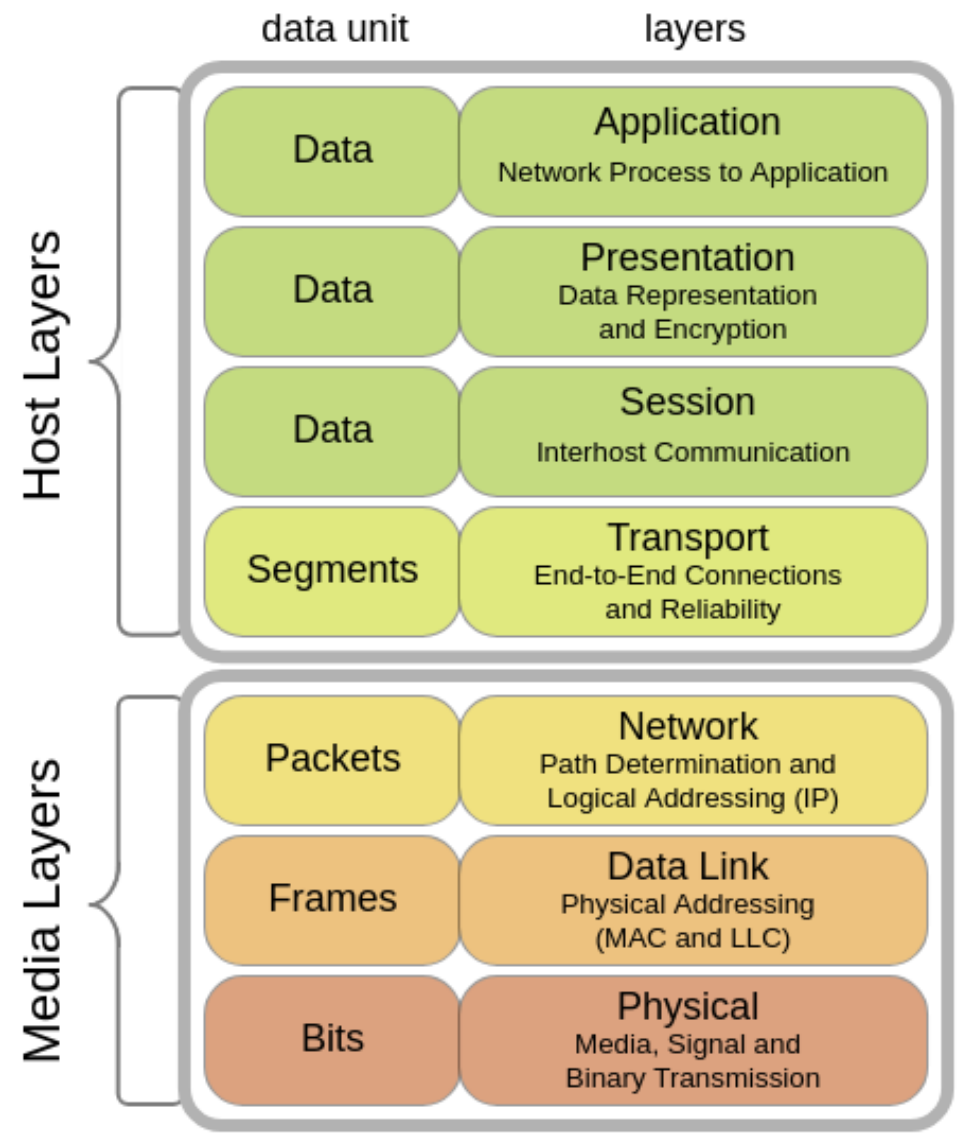
- Open vSwitch avec DPDK (OVS-DPDK)
- Routeurs virtuels dans OpenStack / Kubernetes (Tungsten Fabric, VPP)

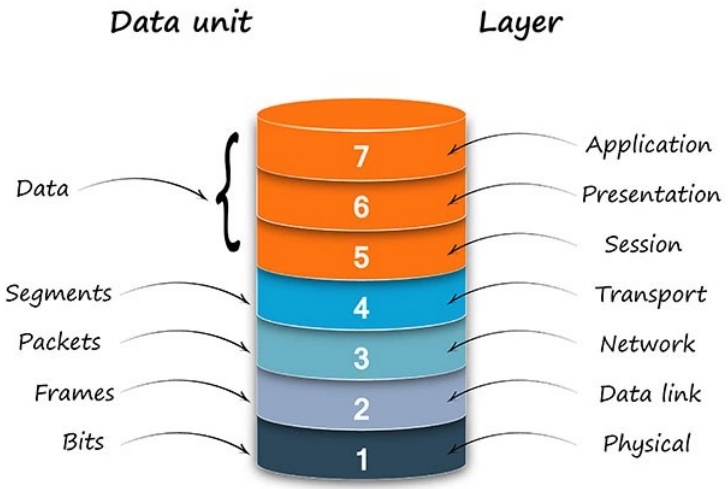
Usage typique :

- dans les **data centers cloud publics ou privés**, où ces dataplanes logiciels servent de « fabric virtuel » pour interconnecter les workloads et mettre en œuvre des réseaux overlay multi-tenant
- dans l'**infrastructure NFV (Network Functions Virtualization / virtualisation des fonctions réseau)** : vRouters, vFW, vLB, EPC/5G core virtualisé, où les VNFs consomment un dataplane DPDK/OVS/VPP pour atteindre des débits proches du fil de fer sur matériel générique x86/ARM.
- dans les **architectures SDN (Software-Defined Networking / réseau défini par logiciel) :** ces briques assurent le rôle de data plane programmable, piloté par un contrôleur central qui pousse les règles de forwarding, les politiques de sécurité et la segmentation réseau de façon dynamique.

What is OSI model



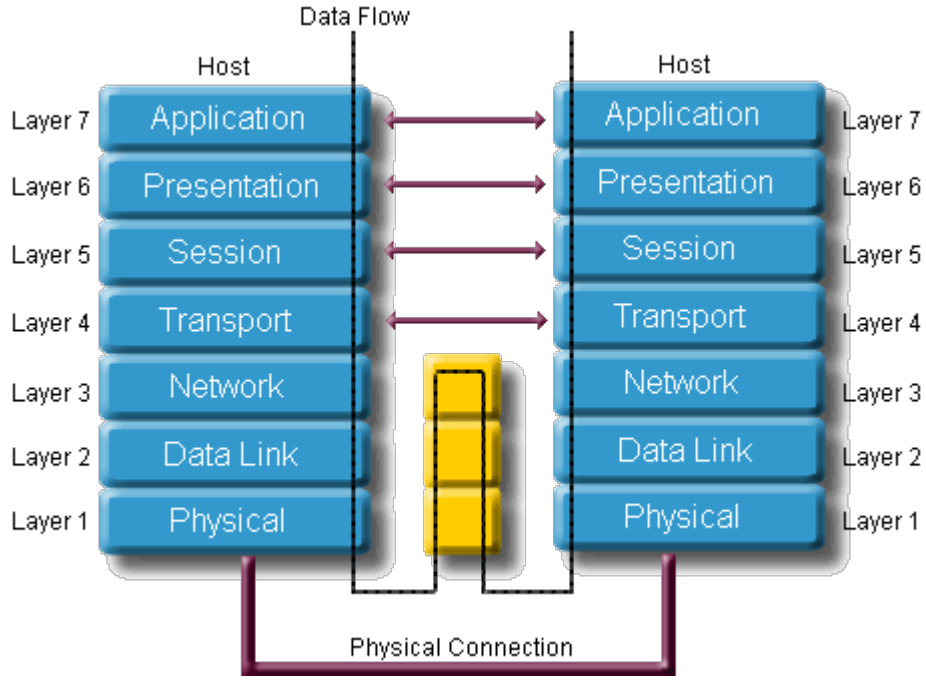

data unit
layers
Host Layers
Data
Application
Network Process to Application
Data
Presentation
Data Representation and Encryption
Data
Session
Interhost Communication
Segments
Transport
End-to-End Connections and Reliability
Media Layers
Packets
Network
Path Determination and Logical Addressing (IP)
Frames
Data Link
Physical Addressing (MAC and LLC)
Bits
Physical
Media, Signal and Binary Transmission


Data unit
Layer
Data
Segments
Packets
Frames
Bits
7
6
5
4
3
2
1
Application
Presentation
Session
Transport
Network
Data link
Physical

OSI model

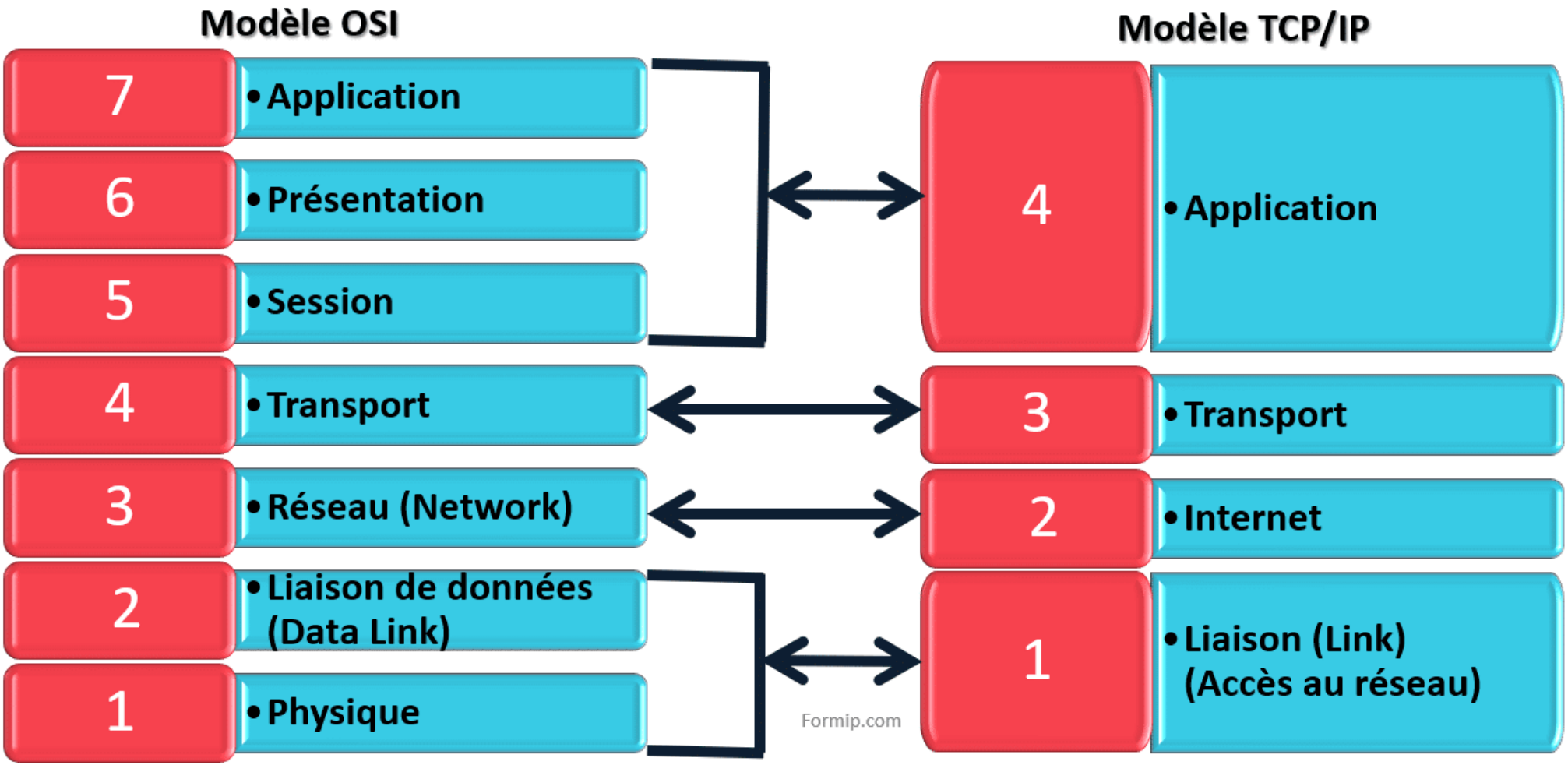
Modèle OSI

7	Application	Services applicatifs au plus proche des utilisateurs.
6	Présentation	Encode, chiffre et compresse les données utiles.
5	Session	Établit des sessions entre des applications.
4	Transport	Établit, maintient et termine des sessions entre des périphériques terminaux.
3	Réseau	Adresse les interfaces globalement et détermine les meilleurs chemins à travers un inter-réseau.
2	Liaison de données	Adresse localement les interfaces, livre les informations localement, méthode MAC.
1	Physique	Encodage du signal, câblage et connecteurs, spécifications physiques.



Protocole TCP / IP Transmission Control Protocol Internet Protocol

TCP = Protocole de contrôle de transmission IP = Protocole Internet



Fonctions réseau virtuelles (NFV / VNF)

DPDK est un pilier de la virtualisation réseau pour construire des fonctions performantes :

- **Vfirewall** : pare-feu logiciel exécuté dans une VM, un conteneur ou un processus dédié, qui applique des politiques de filtrage L3/L4 (et parfois L7) : ACL, états de connexion, règles par IP/port/protocole, etc. Il s'intègre au plan de données virtuel (OVS-DPDK, VPP, SR-IOV, etc.) pour inspecter et accepter/bloquer le trafic à haut débit entre segments réseau ou tenants.
- **Vrouter** : routeur IP virtualisé assurant le routage L3, parfois avec support d'overlays (VXLAN/GRE), VRF, BGP, et services associés (NAT, QoS, politiques). Il tourne sur serveur généraliste et traite les paquets via un dataplane optimisé (DPDK, OVS, VPP) pour connecter VM/pods et réseaux externes dans les clouds ou infrastructures NFV.
- **vLoad Balancer** : répartiteur de charge logiciel qui distribue le trafic entre plusieurs backends selon différentes politiques (round-robin, hash, santé des instances, etc.). Il peut fonctionner en L4 (TCP/UDP) ou L7 (HTTP, HTTPS) et est souvent utilisé pour exposer des services applicatifs dans des architectures cloud ou microservices.
- **vIDS / vIPS (virtual Intrusion Detection System) :** analyse le trafic pour détecter des signatures d'attaques ou des comportements anormaux, sans forcément bloquer (mode détection). Un vIPS (virtual Intrusion Prevention System) va plus loin en bloquant ou modifiant en temps réel les flux suspects, en s'intégrant au chemin de données virtuel pour réagir à la volée.

Fonctions réseau virtuelles (NFV / VNF)

- **NAT haute performance** : réalise la traduction d'adresses et de ports à grande échelle (CGNAT, multi-tenant) en maintenant d'énormes tables d'états tout en restant proche du débit fil de fer. Implémentée comme VNF sur serveur avec DPDK ou équivalent, elle est typiquement utilisée par les FAI et opérateurs mobiles pour gérer la pénurie d'IPv4 et certaines topologies IPv4/IPv6 mixtes.
- **vDPI (Deep Packet Inspection)** : inspecte le contenu des paquets au-delà des en-têtes (jusqu'à la couche applicative), pour classifier les flux, appliquer des politiques, détecter des applications ou menaces. C'est une fonction très consommatrice de CPU/mémoire qui tire profit d'un dataplane accéléré (DPDK, offloads matériels) pour rester en ligne sur des débits élevés.
- **vVPN / IPsec gateway** : passerelle VPN logicielle qui établit et termine des tunnels chiffrés (souvent IPsec) entre sites, utilisateurs distants ou VPC. Elle chiffre/déchiffre le trafic, gère les SA, l'authentification et la négociation (IKE) et peut être intégrée à une stack DPDK ou SR-IOV pour supporter de nombreux tunnels à haut débit sur du matériel standard.

DPDK permet à une VNF d'éviter le kernel et d'atteindre les débits nécessaires au cœur réseau.

Telco / 4G / 5G Core et Edge

DPDK est incontournable dans le réseau mobile, notamment :

- UPF (User Plane Function) 5G
- EPC / PGW / SGW pour la 4G
- MEC (Multi-access Edge Computing)

Les opérateurs utilisent massivement DPDK pour atteindre la latence requise.

Appliances de cybersécurité haut débit

DPDK est utilisé dans de nombreuses solutions de sécurité qui doivent analyser ou manipuler d'énormes volumes :

- système de détection d'intrusion (IDS), système de prévention d'intrusion (IPS) (Suricata, Snort++)
- pare-feux nouvelle génération
- DDoS mitigation
- détection d'anomalies
- boîtiers de surveillance réseau (probe, sondes) (eg : testtree developpé à Rennes)

Pourquoi DPDK ?

- Zéro copie
- Bypass du kernel
- Très faible latence
- Traitement massif de paquets

Load balancers (L4/L7 ou HAProxy-like accélérés)

DPDK permet de construire :

- des load balancers L4 ultra rapides
- des gateways HTTP / TCP capables d'utiliser RSS et multi-queue
- des reverse proxies à très haute capacité

HAProxy peut utiliser DPDK pour atteindre des débits bien supérieurs aux stacks classiques.

Sondes réseaux et monitoring haute précision

DPDK est utilisé pour :

- NetFlow/IPFIX exporters haute performance
- Packet capture massifs (remplacement de PF_RING, AF_PACKET, etc.)
- Analyse réseau en temps réel avec timestamping de précision

Utile pour :

- SOC
- NOC
- Analyse de trafic exigeant des pertes zéro.

Network Function Offload / SmartNIC / FPGA

DPDK sert de chaînon entre les SmartNICs/DPUs et les apps :

- Nvidia BlueField
- Intel IPU
- Mellanox ConnectX
- FPGA offloaded packet processing

DPDK permet d’exposer des queues RX/TX de la carte vers le userspace très efficacement.

Traitement vidéo ou industriel temps réel sur Ethernet

Dans des systèmes non-télécom, DPDK est aussi utilisé pour :

- transport vidéo temps réel (SMPTE 2110, SDI-over-IP)
- systèmes industriels (TSN – Time Sensitive Networking)
- robotique
- broadcast TV

Pourquoi ?

- latence très faible
- contrôle précis du scheduling
- zéro interruption

Systèmes de capture/ré-écriture / middleboxes haute performance

DPDK est utilisé pour construire des middleboxes :

- packet brokers
- duplicateurs/mirroring réseaux
- outils de replay
- modificateurs de paquets (réécriture d'entêtes)
- encapsulation VXLAN/GRE/GTP-U très rapide

Recherche académique et prototypes réseau

Dans l’écosystème recherche :

- benchmarks réseaux
- expérimentations SDN
- prototypes d’architectures réseau
- simulateurs à très haut débit

DPDK est apprécié car il permet d’avoir une base neutre et prévisible pour tester de nouvelles idées.

Liste des cartes réseaux compatibles avec DPDK

Intel — cartes Ethernet & serveurs NIC

- **Contrôleurs “classiques”** :
  - e1000 (82540, 82545, 82546)
  - e1000e (82571, 82572, 82573, 82574, 82583, ICH8/9/10, PCH, PCH2, I217, I218, I219)
- **Contrôleurs plus modernes / serveur / 1-10Gb/s** :
  - série igb (82575, 82576, 82580, I210, I211, I350, I354, DH89xx)
- **Cartes 10 / 25 / 40 / 50 / 100 GbE haute performance** :
  - séries ixgbe (Intel 82598/82599, X540, X550, X520, …), i40e / XL710 / X710 / X722, E810, E822/E823, etc…

→ Ces cartes sont souvent utilisées pour des environnements DPDK haute vitesse.

https://doc.dpdk.org/guides-25.07/linux_gsg/nic_perf_intel_platform.html

Mellanox / NVIDIA — cartes réseau pour datacenters / 10Gb-200Gb+

- **Famille ConnectX** :
  - ConnectX-4, ConnectX-4 Lx, ConnectX-5, ConnectX-5 Ex, ConnectX-6, ConnectX-6 Dx, voire ConnectX-7 (pour les versions récentes) sont supportées.
- **Exemples de NIC spécifiques** :
  - ConnectX-4 10G/25G/40G/50G, ConnectX-5 100G, ConnectX-6 Dx 100G, ConnectX-7 200G.
- Les cartes DPU / SmartNIC / BlueField (NVIDIA) sont aussi officiellement prises en charge.

Broadcom / autres fabricants

- Contrôleurs Broadcom NetXtreme-C / NetXtreme-E / NetXtreme (et leurs variantes) sont dans la liste des NIC supportées.
- **D’autres vendors** :
  - Chelsio (cxgbe)
  - QLogic / Cavium (bnx2x, qede, liquidio,
  - thunderx / octeontx, selon modèle),
  - Cisco (enic),
  - etc...

Pourquoi la compatibilité dépend souvent du pilote / firmware / version

- Pour les cartes haut débit (10G, 25G, 40G, 100G, etc.), il faut souvent que :
  - le firmware soit à jour
  - que les pilotes (PMD DPDK) correspondants soient activés.

Par exemple pour les cartes Mellanox (via mlx5 PMD), la version OFED/firmware importe.

- Certaines variantes “OEM” ou “rebadgées” d’une carte peuvent ne pas être testées — la compatibilité réelle peut varier.
- Pour de très hautes vitesses (≥ 25/40/100 Gb/s), il est souvent recommandé d’utiliser un slot **PCIe 3.0 x8 (ou plus)** pour avoir suffisamment de bande passante.

Que faire pour vérifier la compatibilité sur ta machine ?

- Vérifier le modèle exact de ta carte réseau (via lspci, ethtool, etc.) et le comparer à la matrice des NIC supportées de DPDK.
- Vérifier que le firmware de la carte est à jour — particulièrement pour les cartes 10G+ / SmartNIC.
- Vérifier que le pilote/PMD DPDK correspondant est activé dans ta configuration DPDK.
- Pour les NIC virtuelles ou “non standard” (USB Ethernet, cartes Wi-Fi, chipsets bas de gamme…), ne pas s’attendre à une compatibilité fiable avec DPDK.

Prérequis : Matériel CPU / RAM

- CPU x86_64 type Xeon ou équivalent (Ivy Bridge/Haswell ou plus récent recommandé pour de bonnes perfs).
- Support des hugepages 2 MiB et idéalement 1 GiB via les flags CPU pse (2M) et pdpe1gb (1G).
- Plateforme NUMA supportée (2 sockets ou plus) fortement recommandée pour des débits élevés.
- RAM suffisante, avec au moins un DIMM par canal mémoire (typique: ≥4 Go par canal) pour saturer la bande passante.

Prérequis : Cartes réseau (NIC)

- NIC supportées par les PMD DPDK (Intel 82599, X540/X550, i40e/ice, E810, Mellanox ConnectX-4/5/6, Broadcom NetXtreme, etc.).
- Firmware à jour et configuration PCIe adaptée (x8 ou x16 Gen3/4/5 pour 25–100G).
- Accès PCIe direct depuis l’hôte (pas de mode bridgé exotique ou NIC USB / Wi-Fi, très mal supportés).

Prérequis : Kernel et OS

- Kernel Linux ≥ 3.16 pour les anciennes versions, les releases récentes sont validées sur des kernels plus modernes (RHEL/CentOS 7+, Ubuntu 20.04/22.04, Fedora, etc.).
- glibc ≥ 2.7 pour la gestion des cpusets et les fonctionnalités nécessaires.
- Libnuma (libnuma-dev / numactl-devel) recommandée, voire requise, pour le mode mémoire par défaut sur systèmes NUMA.

Prérequis : Configuration kernel / boot

- Hugepages activées dans le kernel (CONFIG_HUGETLBFS, CONFIG_HUGETLB_PAGE) et montées (ex: /dev/hugepages ou /mnt/huge).
- Réservation de hugepages 2M ou 1G soit à chaud via /sys/kernel/mm/hugepages/.../nr_hugepages, soit au boot via hugepages=, hugepagesz= et default_hugepagesz=.
- Pour VFIO : IOMMU activé et configuré en passthrough, typiquement intel_iommu=on iommu=pt (ou équivalent AMD) dans la ligne de commande kernel.
- Isolation des cœurs réservés à DPDK avec isolcpus, nohz_full, irqaffinity pour éviter les IRQs et tâches kernel parasites sur ces cores.

Prérequis : Espace utilisateur / drivers

- Possibilité de binder la NIC vers un driver user-space (VFIO recommandé, UIO igb_uio ou uio_pci_generic possible).
- DPDK compilé avec un compilateur C11 (depuis 23.11, support C11 obligatoire, notamment pour les atomiques).

En pratique, sur un serveur x86_64 moderne type Xeon/EPYC avec Ubuntu 22.04 ou RHEL 8/9, NIC 10–100G Intel ou Mellanox, hugepages configurées et IOMMU activé, tu remplis tous les prérequis matériels et kernel pour DPDK.

Compilation de la cible DPDK à partir du code source

GITHUB OFFICIEL

- git clone https://github.com/DPDK/dpdk.git

Sur cette organisation GitHub (DPDK), il y a aussi quelques autres dépôts liés au projet (par exemple grout, dpdk-stable, etc.), mais le principal est dpdk/dpdk.

La gestion de développement officielle (envoi de patches, revue, etc.) se fait principalement via la liste de diffusion et les outils Git du projet, pas exclusivement via les pull requests GitHub comme pour beaucoup d'autres projets open source.

Si tu veux aussi le lien « officiel » hébergé par le projet (qui redirige souvent vers des tags et archives) :

Le lien suivant fournit des sources, docs et téléchargements. :

- https://core.dpdk.org/

Prérequis

- **Version du noyau >= 5.4 :**

L'API espace utilisateur Linux est compatible entre les différentes versions, mais présente certaines restrictions. Le noyau le plus ancien testé par l'infrastructure de test DPDK est la plus ancienne version stable à long terme (LTS) maintenue au moment de sa publication.

Certains pilotes et composants matériels peuvent nécessiter des noyaux plus récents ; consultez la documentation.

La version du noyau utilisée peut être vérifiée à l'aide de la commande : **$ uname -r**

- **glibc >= 2.7 (pour les fonctionnalités liées au cpuset) :**

La version peut être vérifiée à l'aide de la commande.ldd –version

- **Configuration du noyau :**

Sous Fedora OS et d'autres distributions courantes, telles qu'Ubuntu ou Red Hat Enterprise Linux, les configurations du noyau fournies par le fournisseur peuvent être utilisées pour exécuter la plupart des applications DPDK.

Pour les autres versions du noyau, les options à activer pour DPDK sont les suivantes :

- **HUGETLBFS**
- Prise en charge de **PROC_PAGE_MONITOR**
- Les options de configuration **HPET** et **HPET_MMAP** doivent également être activées si la prise en charge HPET est requise.

Utilisation des Hugepages dans l'environnement Linux

La prise en charge des pages énormes est requise pour l'allocation de mémoire importante utilisée pour les tampons de paquets (l'option HUGETLBFS doit être activée dans le noyau en cours d'exécution, comme indiqué dans la section précédente).

L'utilisation de pages énormes améliore les performances car moins de pages sont nécessaires, et donc moins de tampons de traduction d'adresses (TLB, caches de traduction haute vitesse), ce qui réduit le temps de traduction d'une adresse de page virtuelle en une adresse de page physique.

Sans pages énormes, avec une taille de page standard de 4 Ko, le taux d'échec des TLB serait élevé, ce qui ralentirait les performances.

Réservation de pages volumineuses pour l'utilisation de DPDK

La réservation de pages mémoire de grande taille (hugepages) peut être effectuée à l'exécution.

Pour ce faire, on écrit le nombre de pages nécessaires dans un nr_hugepagesfichier du /sys/kernel/répertoire correspondant à une taille de page spécifique (en kilo-octets). Sur un système à nœud unique, la commande à utiliser est la suivante (en supposant que 1 024 pages de 2 Mo soient nécessaires) :

```
$ echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

Sur une machine NUMA, la commande ci-dessus répartit généralement le nombre de pages énormes de manière égale entre tous les nœuds NUMA (en supposant que chaque nœud dispose de suffisamment de mémoire).

Toutefois, il est également possible de réserver explicitement des pages sur chaque nœud NUMA à l'aide d'un nr_hugepagesfichier situé dans le /sys/devices/répertoire :

```
$ echo 1024 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages

$ echo 1024 > /sys/devices/system/node/node1/hugepages/hugepages-2048kB/nr_hugepages
```

Cet outil dpdk-hugepages.pypeut être utilisé pour gérer des pages volumineuses.

Réservation de pages volumineuses pour l'utilisation de DPDK

Certaines versions du noyau peuvent ne pas permettre la réservation de pages mémoire de 1 Go à l'exécution ; il est donc possible que la seule solution soit de les réserver au démarrage. Veuillez consulter les instructions ci-dessous :

Dans la plupart des cas, réserver des pages mémoire de grande taille lors de l'exécution ne pose aucun problème, mais dans les cas d'utilisation où une grande quantité de mémoire physiquement contiguë est nécessaire, il est préférable de réserver ces pages au démarrage, car cela permettra d'éviter une fragmentation importante de la mémoire physique.

Pour réserver des pages mémoire de grande taille au moment du démarrage, un paramètre est transmis au noyau Linux sur la ligne de commande du noyau.

Pour les pages de 2 Mo, il suffit de passer l'option hugepages au noyau. Par exemple, pour réserver 1024 pages de 2 Mo, utilisez :

```
hugepages=1024
```

Pour les autres tailles de pages énormes, par exemple les pages de 1 Go, la taille doit être spécifiée explicitement et peut également être définie comme taille de page énorme par défaut pour le système. Par exemple, pour réserver 4 Go de mémoire de pages énormes sous la forme de quatre pages de 1 Go, les options suivantes doivent être transmises au noyau :

```
default_hugepagesz=1G hugepagesz=1G hugepages=4
```

Réservation de pages volumineuses pour l'utilisation de DPDK

Sur l'architecture Intel, la taille des pages énormes (hugepages) prises en charge par un processeur est déterminée par ses indicateurs (flags) :

- ai l'indicateur `**pse**` est présent, les pages énormes de 2 Mo sont prises en charge ;
- si l'indicateur `**pdpe1gb**` est présent, les pages énormes de 1 Go sont prises en charge.
- sur l'architecture *IBM Power*, les tailles de pages énormes prises en charge sont de 16 Mo et 16 Go.
- pour les applications **64 bits**, il est recommandé d'utiliser des pages mémoire de 1 Go si la plateforme les prend en charge.

Dans le cas d'un système NUMA à deux sockets, le nombre de pages énormes réservées au démarrage est généralement divisé de manière égale entre les deux sockets (en supposant qu'il y ait suffisamment de mémoire sur les deux sockets).

Utilisation des Hugepages avec le DPDK

Si la prise en charge des processus secondaires n'est pas requise,

DPDK peut utiliser les pages mémoire de grande taille sans aucune configuration grâce au mode « en mémoire ». Veuillez consulter les paramètres EAL pour plus de détails.

Si la prise en charge des processus secondaires est requise, des points de montage pour les pages énormes doivent être créés. Sur les distributions Linux modernes, un point de montage par défaut pour les pages énormes est fourni par le système et se trouve à l'emplacement indiqué /dev/hugepages.

Ce point de montage utilisera la taille de page énorme par défaut définie par les paramètres du noyau, comme décrit précédemment.

Toutefois, pour utiliser des tailles de pages énormes autres que celles par défaut, il est nécessaire de créer manuellement des points de montage pour ces tailles de pages énormes (par exemple, des pages de 1 Go).

Pour rendre les pages énormes de 1 Go disponibles pour une utilisation avec DPDK, les étapes suivantes doivent être effectuées :

```
$ mkdir /mnt/huge

$ mount -t hugetlbfs pagesize=1GB /mnt/huge
```

Le point de montage peut être rendu permanent après un redémarrage en ajoutant la ligne suivante au /etc/fstabfichier :

```
nodev /mnt/huge hugetlbfs pagesize=1GB 0 0
```

Téléchargement & décompression

Commencez par décompresser l'archive et accédez au répertoire source DPDK décompressé :

```
$ tar xJf dpdk-<version>.tar.xz
$ cd dpdk-<version>
```

$Le DPDK est composé de plusieurs répertoires, notamment :

- **Documentation DPDK**
- **licence** : informations de licence DPDK
- **lib** : Code source des bibliothèques DPDK
- **Pilotes** : Code source des pilotes DPDK en mode polling
- **application** : Code source des applications DPDK (tests automatiques)
- **Exemples** : Code source d’exemples d’applications DPDK
- **config, buildtools** : Scripts et configuration liés au framework
- **usertools** : Scripts utilitaires pour les utilisateurs finaux des applications DPDK
- **outils de développement** : scripts à l’usage des développeurs DPDK
- **noyau** : modules du noyau nécessaires à certains systèmes d’exploitation

Configuration de DPDK

Pour **configurer** une version DPDK, utilisez :

```
$ meson setup <options> build
```

où « build » correspond au répertoire de compilation souhaité, et « <options> » peut être vide ou contenir l'une des options de compilation spécifiques à Meson ou DPDK, décrites plus loin dans cette section.

Le processus de configuration se termine par un récapitulatif des bibliothèques et pilotes DPDK à compiler et à installer, ainsi que la raison de chaque désactivation. Ces informations permettent, par exemple, d'identifier les paquets manquants pour un pilote.

Une fois **configuré**, pour **compiler** puis **installer** DPDK à l'échelle du système :

```
$ cd build

$ ninja

$ sudo meson install

$ sudo ldconfig
```

Les deux dernières commandes ci-dessus doivent généralement être exécutées en tant que root, l' étape d'installation de meson copiant les objets construits vers leurs emplacements finaux à l'échelle du système, et la dernière étape forçant le chargeur dynamique ld.so à mettre à jour son cache pour prendre en compte les nouveaux objets.

Sur certaines distributions Linux, comme Fedora ou Red Hat, les chemins d'accès dans /usr/local ne font pas partie des chemins par défaut du chargeur. Par conséquent, sur ces distributions, /usr/local/lib et /usr/local/lib64 doivent être ajoutés à un fichier dans /etc/ld.so.conf.d/ avant d'exécuter ldconfig .

Options de configuration

Paramètre / Option	Rôle / Effet principal	Exemple d'utilisation	Remarques clés
buildtype ▸	Définit le type de compilation (optimisation, symboles de debug, etc.).	`meson setup -Dbuildtype=debug build` ou `meson configure -Dbuildtype=debug`	Valeur par défaut : **debugoptimized** ▸. Permet de passer en build purement debug.
Options Meson générales	Ensemble d'options génériques de l'outil Meson utilisées par DPDK (chemins, warnings, optimisation, etc.).	`meson configure` dans le répertoire de build	Documentées sur le site de Meson (options « builtin »).
platform ▸	Choisit un profil de configuration global (flags CPU, options DPDK, etc.).	`-Dplatform=native`, `-Dplatform=generic`, `-Dplatform=<SoC>` ↓	Pilote la configuration du jeu d'instructions et d'autres paramètres dépendants de la plateforme.

Options de configuration

platform=native ▶	Adapte la configuration à la machine de compilation (profil « natif »).	`meson setup -Dplatform=native build`	Configure automatiquement le jeu d'instructions CPU en mode natif.
platform=generic ▶	Utilise une configuration générique compatible avec toutes les machines de même architecture.	`meson setup -Dplatform=generic build`	Cible une base d'instructions minimale commune pour assurer la portabilité.
platform=<SoC> ▶	Optimise la configuration pour un SoC Arm particulier.	`meson setup -Dplatform=thunderx build` (exemple)	La liste des SoC supportés est définie dans le dictionnaire **socs** ▶ de `config/arm/meson.build`.
cpu_instruction_set ▶	Spécifie explicitement le jeu d'instructions CPU utilisé pour la compilation (x86, Power, etc.).	`meson configure -Dcpu_instruction_set=corei7` ↓	Sur x86/Power : permet de cibler **-march** ▶, **-mcpu** ▶, **-mtune** ▶. Non recommandé seul sur Arm.

Options de configuration

max_lcores	Définit le nombre maximal de **lcores** que DPDK peut utiliser.	`meson setup -Dmax_lcores=256 build` ou `meson configure -Dmax_lcores=256`	Permet d'ajuster DPDK à des plateformes avec un grand nombre de cœurs logiques.
examples	Liste des applications d'exemple à compiler automatiquement avec DPDK.	`meson setup -Dexamples=l2fwd,l3fwd build`	Valeurs multiples séparées par des virgules ; géré aussi après coup via `meson configure -Dexamples=...`.
examples=all	Demande la compilation de toutes les applications d'exemple compatibles avec le système courant.	`meson setup -Dexamples=all build`	Meson vérifie les dépendances et ajoute uniquement les exemples compilables dans la liste de cibles **Ninja**.

Migration d’un application existante

Clarification du périmètre de migration

Pour migrer une appli socket classique vers DPDK, il faut en pratique réécrire tout le dataplane en changeant de modèle (kernel → user-space PMD).

L'approche réaliste est donc une migration progressive par couches, pas un simple refactor des appels socket().

- DPDK remplace uniquement la partie « dataplane » (I/O paquet, parsing, forwarding), pas la logique métier ni le control-plane (REST, CLI, etc.).
- Les appels socket()/send()/recv() disparaissent côté dataplane et sont remplacés par rte_eth_rx_burst() / rte_eth_tx_burst() sur des NIC bindées à DPDK.

Idéalement, il faut identifier un module « I/O réseau » dans ton appli et tu le remplaces par une implémentation DPDK en gardant au maximum les interfaces internes identiques.

Préparer l’environnement DPDK

Avant même de toucher au code de l’appli :

- Choisir/valider une NIC compatible, binder le driver DPDK (vfio-pci ou uio) et réserver les hugepages correctement.
- Compiler DPDK + un exemple (l2fwd, testpmd) et vérifier que tu atteins bien le débit attendu sur ta machine.

Extraire et isoler le dataplane existant

Dans l’appli socket actuelle :

- Localiser la boucle réseau principale (souvent epoll()/select()/poll() ou recvfrom()/sendto() dans un thread).
- Isoler les parties :
  - Parsing/proto (L2/L3/L4 ou applicatif)
  - Décision (routing, ACL, LB, etc.)
  - I/O (syscalls réseau)

But : transformer la boucle en 3 briques distinctes : receive() -> process() -> send() pour pouvoir ne remplacer que receive()/send() par DPDK.

Concevoir l’architecture DPDK cible

Choisir un modèle :

- Single-core « run-to-completion » : une boucle RX → traitement → TX sur un seul lcore.
- Multi-core pipeline : un core RX, un ou plusieurs cores traitement, un core TX, communiquant via rte_ring.

Pour une migration directe d’une appli socket, un premier port en run-to-completion 1–2 cores est souvent plus simple, quitte à passer en pipeline ensuite.

Implémenter un squelette DPDK minimal

Créer une nouvelle binaire ou réorganiser :

1) Initialisation EAL (rte_eal_init) avec cores et hugepages.
2) Création d'un mempool de mbuf pour les paquets.
3) Configuration du port Ethernet : rte_eth_dev_configure, rte_eth_rx_queue_setup, rte_eth_tx_queue_setup, rte_eth_dev_start.
4) Boucle de traitement :
   - ➔ rte_eth_rx_burst(port, queue, pkts, BURST)
   - ➔ pour chaque mbuf, appeler la logique métier existante (adaptée aux buffers DPDK)
   - ➔ rte_eth_tx_burst() pour renvoyer les paquets.

À ce stade, tu as déjà un « hello world DPDK » qui peut remplacer grossièrement le flux recv()/send().

Adapter la logique de parsing et de protocole

Dans l’appli sockets, tu travailles sur des buffers char *buf issus de recv().

Avec DPDK :

- Tu récupères un struct rte_mbuf *m.
- Le pointeur vers les données est rte_pktmbuf_mtod(m, void *), la longueur rte_pktmbuf_data_len(m) (ou pkt_len).

Il faut donc :

- Adapter les fonctions de parsing pour travailler sur ces buffers (ou wrapper en une API interne packet_view qui abstrait DPDK).
- Gérer toi-même l’offset des headers (Ethernet/IP/TCP/UDP) au lieu de laisser le kernel parser et te donner juste le payload.

Si ton appli est purement applicative L7 (HTTP, etc.), il peut être plus pertinent de garder la stack TCP kernel et d’utiliser DPDK seulement comme « capture rapide + réinjection via KNI/TAP », mais c’est un autre design.

Remplacer la gestion des connexions / sessions

Avec BSD sockets, le kernel gère :

- Connexions TCP, états, retransmissions, congestion.
- Table de sessions, timeouts, 4-tuple, etc.

En DPDK pur tu dois soit :

- Implémenter ta propre gestion de sessions (hash table clé = 5-tuple, état applicatif, timers), pour des applis type firewall/LB UDP/« TCP stateless ».
- Ou intégrer un user-space TCP stack (mTCP, F-Stack, Seastar, etc.) qui tourne au-dessus de DPDK.

C’est là que la migration devient lourde : une appli HTTP/TCP classique devient en pratique un projet avec stack TCP user-space complète si tu veux rester full-bypass.

Assurer l’interop avec le kernel (si nécessaire)

Si ton appli doit continuer à communiquer avec des processus classiques (sockets kernel) :

- Utiliser KNI, TAP PMD ou PCAP PMD pour réinjecter certains flux dans la stack kernel.
- Ou dédier une NIC à DPDK et une NIC séparée (ou une VF) au reste du système.

Cela permet une migration progressive : certains flux passent en DPDK, d’autres continuent à utiliser la pile kernel.

Gestion des threads, CPU et NUMA

Une appli socket classique se repose sur le scheduler Linux. Avec DPDK :

- Tu fixes tes threads/lcores via masques CPU, rte_lcore_foreach(), rte_eal_remote_launch().
- Tu pin-nes les threads sur des cœurs dédiés et t’assures que mempool/queues sont sur le bon NUMA node.

Il faut donc retirer les appels pthread_create() qui géraient le I/O bloquant et les remplacer par le modèle lcore de DPDK.

Tester et comparer

Pour valider la migration :

- Mesurer pps/latence avec l’ancienne version (sockets) et la nouvelle (DPDK) sous trafic artificiel (TRex, pktgen, testpmd).
- Vérifier la stabilité (no drop, no leaks mbuf, pas de starvation CPU).

L’intérêt de DPDK se voit surtout à partir de quelques millions de paquets/s ou de contraintes de latence très strictes.

Comment isoler une NIC du kernel

Comment isoler une NIC du kernel

Par défaut, ta NIC utilise un driver kernel (ex : ixgbe, i40e, mlx5_core).

Pour l'utiliser avec DPDK, il faut la « détacher » du kernel et la binder à vfio-pci ou uio_pci_generic.

Vérifier l'état des NIC :

$ sudo dpdk-devbind.py –status

```
Network devices using kernel driver
  0000:04:00.0 '82599ES 10-Gigabit' if=eth1 drv=ixgbe
Network devices not managed by kernel
  (vide pour l'instant)
```

Charger le driver vfio-pci (recommandé)

```
$ sudo modprobe vfio-pci
```

Si IOMMU est désactivé :

```
$ sudo modprobe vfio enable_unsafe_noiommu_mode=1
```

Binder la carte réseau via dpdk-devbind.py :

$ sudo dpdk-devbind.py --bind=vfio-pci 0000:04:00.0

Network devices using DPDK-compatible driver

0000:04:00.0 '82599ES 10-Gigabit' drv=vfio-pci

La NIC ne possède plus d’interface Linux (plus de eth1).

Autre solution en désactivant les IRQ :

$ /etc/modprobe.d/blacklist-ixgbe.conf blacklist ixgbe

Débinder (pour revenir au kernel) :

$ sudo dpdk-devbind.py --bind=ixgbe 0000:04:00.0

Architecture de référence haute performance (100Mpps)

- Core 1: RX → classify → distribute
- Core 2-5: worker cores (NAT, firewall, L7 parsing)
- Core 6: TX aggregator
- Core 0: management (stats, health)

DPDK EAL : le cœur du framework

EAL (Environment Abstraction Layer)

EAL (Environment Abstraction Layer) est chargé de :

- **Réservation mémoire**
  - HugePages 2 Mo ou 1 Go
  - mapping NUMA-aware
  - DMA coherent memory

Terme	Définition	Importance pour DPDK
HugePages 2 Mo/1 Go	Pages mémoire de grande taille	Performance + DMA + mémoire contiguë
NUMA-aware mapping	Allocation mémoire optimisée par nœud CPU	Latence min., débit max.
DMA coherent memory	Mémoire cohérente entre CPU et NIC	Accès direct du NIC sans incohérence

Exemple configuration :

```
$ echo 4096 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ mount -t hugetlbfs nodev /mnt/huge
```

HugePages : 2 Mo ou 1 Go

Les HugePages sont des pages mémoire de grande taille utilisées par le noyau Linux.

Normalement, Linux utilise des pages de 4 Ko, mais pour les applications gourmandes (DPDK, bases de données, VM…), cela devient inefficace.

DPDK utilise préférentiellement des HugePages pour plusieurs raisons :

► Pourquoi 2 Mo ou 1 Go ?

Linux supporte plusieurs tailles :

- 2 Mo (HugePages standard)
- 1 Go (GigaPages)

► Avantages pour DPDK

✔️ Moins de pages → moins d'entrées dans la table mémoire → moins de TLB misses

✔️ Meilleures performances en DMA

✔️ Alloue un large bloc contigu de mémoire, nécessaire pour les mbufs et la mémoire packet-buffers

DPDK se sert massivement des HugePages pour alimenter son mempool.

Mapping NUMA-aware

Les processeurs modernes ont plusieurs nœuds NUMA (Non-Uniform Memory Access).

Chaque nœud comprend :

- des cœurs CPU
- un contrôleur mémoire
- sa propre RAM locale

► NUMA-aware mapping signifie :

DPDK alloue la mémoire en fonction du nœud NUMA du CPU qui va traiter le trafic.

► Pourquoi ?

✔️ Un CPU accédant à sa mémoire locale est beaucoup plus rapide

✔️ Limite les accès inter-nœuds (cross-NUMA), qui augmentent la latence

✔️ Permet de maximiser le débit réseau et de réduire les bottlenecks

NUMA : principe et architecture

Architecture mémoire utilisée sur les serveurs multi-cœurs / multi-sockets

- La mémoire est physiquement distribuée par nœuds NUMA
- Chaque nœud NUMA =
  - 1 CPU (ou socket)
  - ses cœurs
  - sa mémoire locale

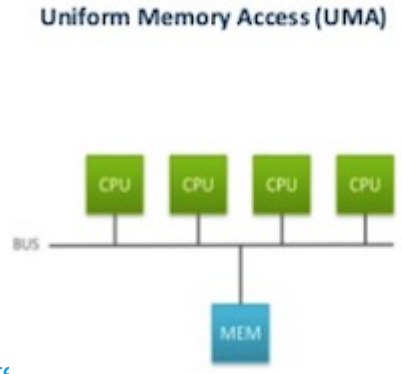
Accès mémoire

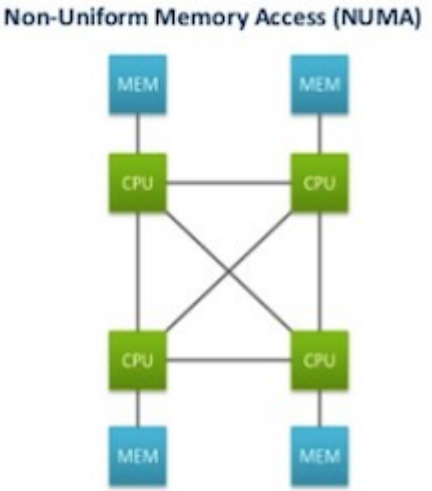
- Mémoire locale → accès rapide
- Mémoire distante (autre nœud) → plus lent (latence + bande passante réduite

Pourquoi NUMA ?

- Scalabilité (beaucoup de cœurs)
- Réduction des goulots d'étranglement mémoire
- Architecture standard sur x86 (Xeon / EPYC) et ARM serveur

📌 La latence mémoire dépend de la localisation des données.





NUMA : impact et bonnes pratiques

Impact sur les performances

Accès mémoire distant = +30 % à +100 % de latence !

Mauvaise affinité CPU/mémoire → perte massive de performance

Crucial pour :

- DPDK
- bases de données
- virtualisation
- HPC

Bonnes pratiques

- Lier les threads à un CPU (CPU pinning)
- Allouer la mémoire sur le nœud local
- Éviter les partages mémoire inter-nœuds

Un cœur ↔ une queue ↔ mémoire locale (DPDK)

- Outils Linux :

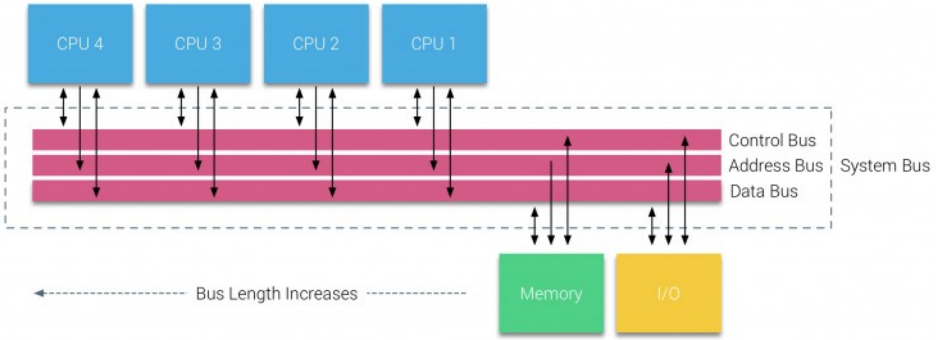
$ numactl --hardware

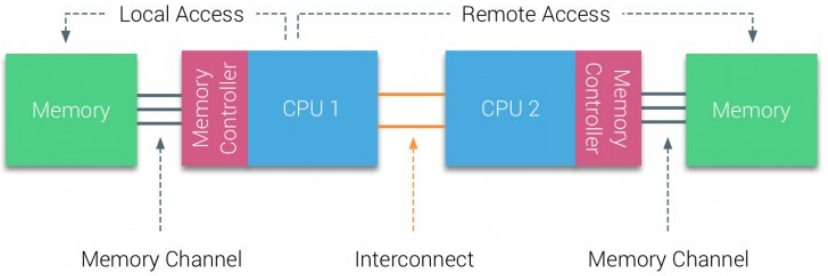
$ numactl --cpunodebind=0 --membind=0 ./app

$ lscpu | grep NUMA

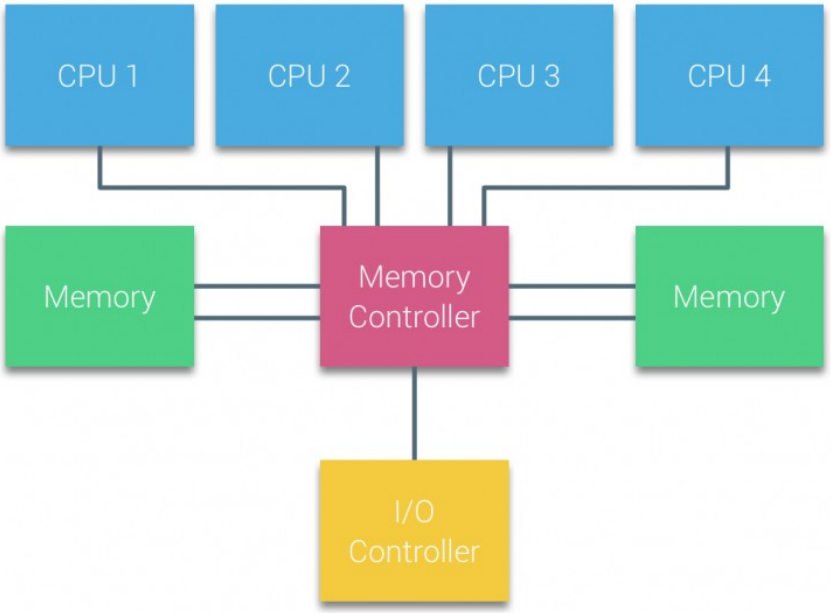
📌 NUMA bien maîtrisé = performance maximale.

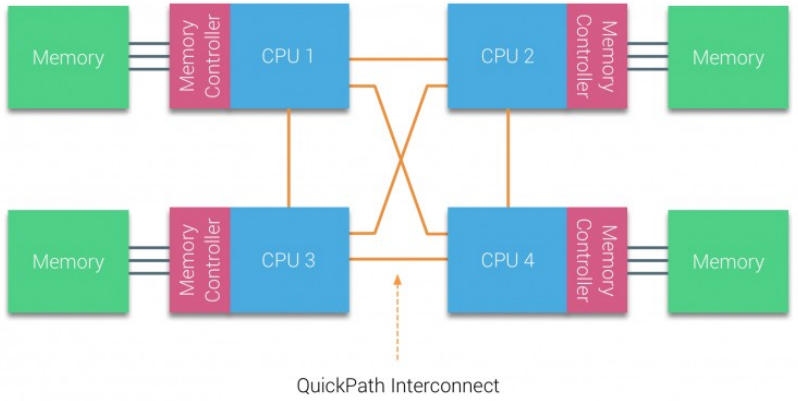
De UMA a NUMA





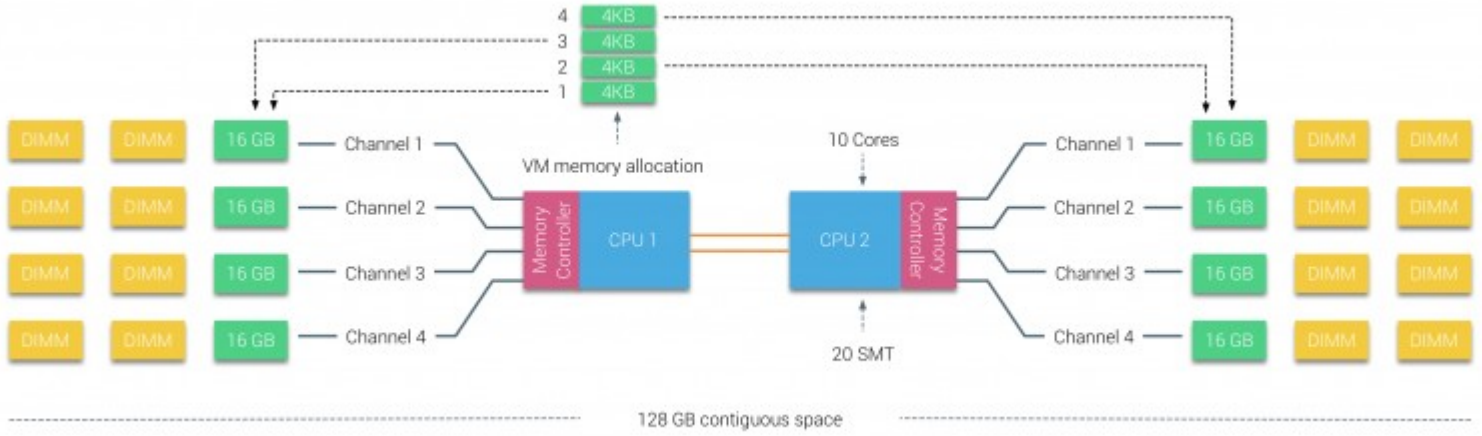
Non-Uniform Memory Access organization

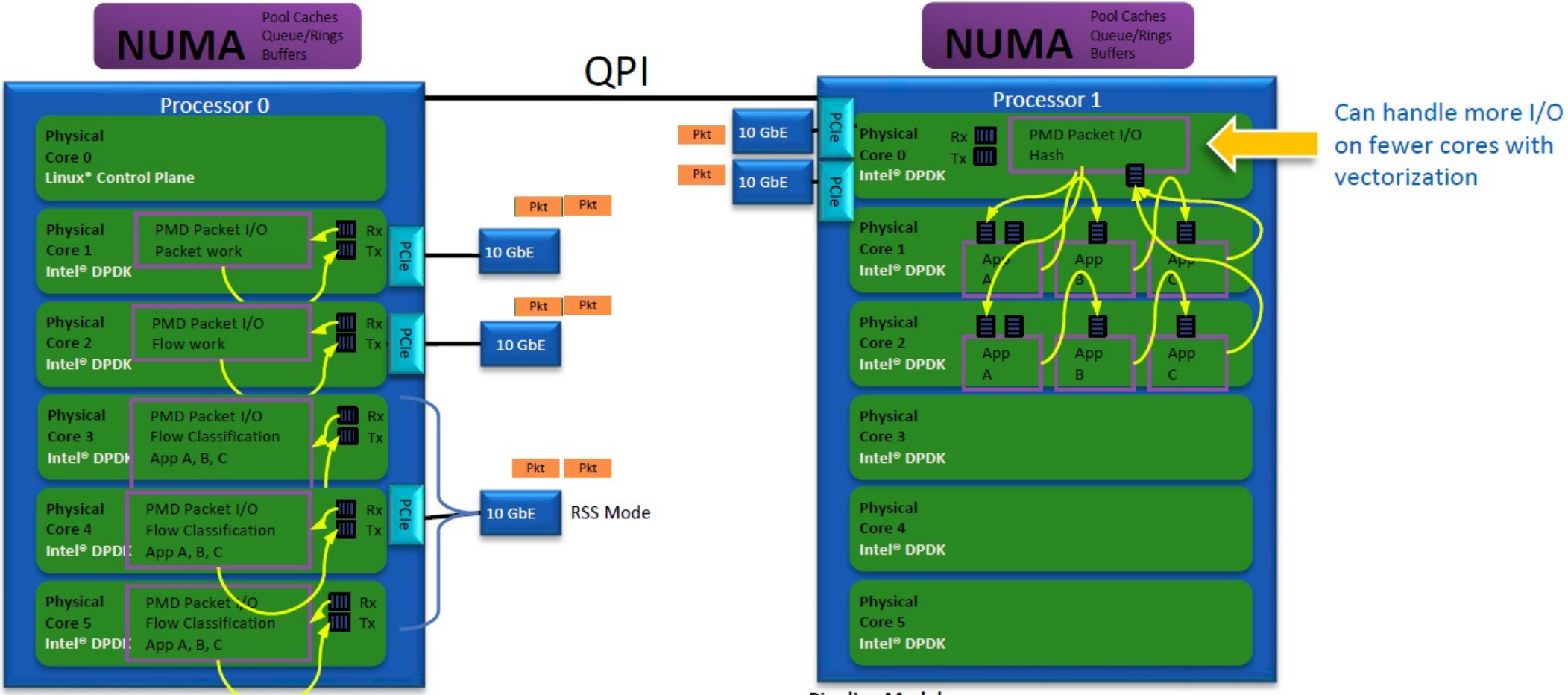




Point-to-Point interconnect

Non-interleaved enabled NUMA = SUMA





Run to Completion Model

- I/O and Application workload can be handled on a single core
- I/O can be scaled over multiple cores

Pipeline Model

- I/O application disperses packets to other cores
- Application work performed on other cores

DMA coherent memory

Le DMA (Direct Memory Access) permet à la carte réseau (NIC) de lire/écrire directement en RAM, sans passer par le CPU (libère donc le bus de données de la carte mere) :

► **DMA coherent memory signifie :**

C'est une zone mémoire où :

- le CPU et la carte réseau voient la même version des données
- le matériel n'a pas besoin de flush/invalidate des caches
- la synchronisation CPU ↔ NIC est immédiate

► **Dans DPDK :**

DPDK utilise énormément de mémoire DMA pour :

- les descriptors RX/TX
- les ring buffers
- les mbufs
- les zones mémoire partagées avec les NIC

Cette mémoire doit être :

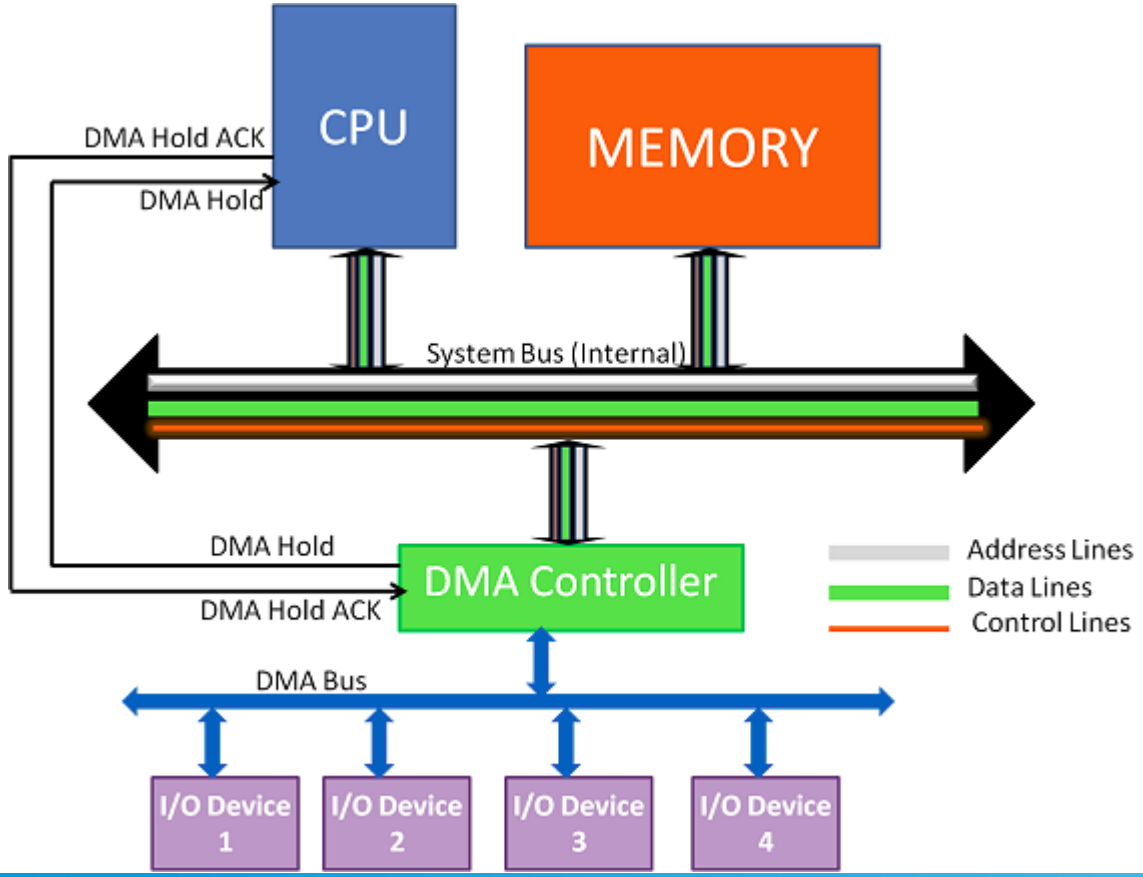
✔️ physiquement contiguë

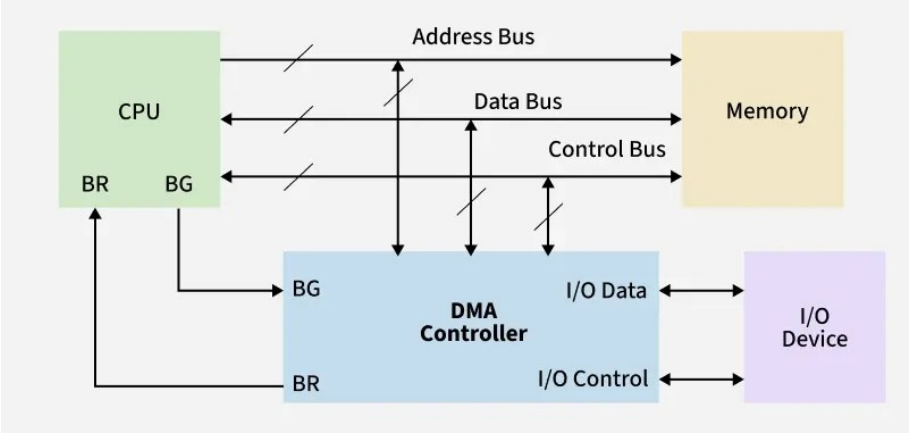
✔️ non paginée

✔️ DMA-safe

C'est pour cela que DPDK utilise les HugePages + l'allocateur EAL.

Direct Memory Access (DMA)





EAL (Environment Abstraction Layer)

- **Gestion des lcores**

DPDK ne crée pas des threads classiques. Il utilise un système lcore abstrait :

- lcore0 = master
- lcore1..n = workers
- mappés via masques CPU

Exemple :

```
$ dpdk-app -l 1,2,3 --socket-mem=1024,1024
```

EAL (Environment Abstraction Layer)

- **Gestion PCIe / probing**

EAL scanne automatiquement les cartes PCI puis charge les PMD correspondants.

Cela nécessite parfois d’isoler la NIC du kernel.

Gestion mémoire : Hugepages, memzones, mempools, mbufs

Gestion mémoire

- **DPDK supprime les interruptions NIC :**

Les threads tournent en boucle (busy loop) et scrutent les RX queues.

Avantages :

- latence minimale
- Évite les softirq
- Prévisible en load extrême

Inconvénients :

- CPU consommé à 100%
- Chauffe / énergie

Réseau classique : modèle interruptions (IRQ)

Dans un système Linux standard :

- La NIC reçoit un paquet
- Elle déclenche une interruption matérielle (IRQ)
- Le CPU interrompt ce qu'il faisait
- Le driver lit le paquet
- Le traitement réseau continue

👉 Avantage :

CPU peu sollicité quand il n'y a pas de trafic

👉 Inconvénients :

- coût élevé des interruptions
- context switches fréquents
- latence variable
- très mauvais à haut débit (des millions d'IRQ/s)

DPDK : suppression des interruptions NIC

DPDK désactive les IRQ réseau et adopte un modèle polling.

Ce que fait DPDK :

- Chaque thread DPDK est attaché à un cœur CPU
- Il exécute une boucle infinie (busy loop)
- Il scrute en permanence les files RX de la carte réseau

Pseudo-code simplifié :

```
while (1) {
    nb_rx = rte_eth_rx_burst(port, queue, pkts, BURST_SIZE);
    if (nb_rx > 0) {
        process_packets(pkts, nb_rx);
    }
}
```

➡️ Le thread ne dort jamais

➡️ Il ne rend jamais le CPU au scheduler

➡️ Le cœur est occupé 100 % du temps

Pourquoi ce choix est volontaire ?

1. Éliminer le coût des interruptions

- Pas d'IRQ
- Pas de réveil de thread
- Pas de changement de contexte

2. Latence minimale

- Le paquet est traité dès qu'il arrive
- Pas d'attente d'interruption
- Latence prévisible et très basse

3. Débit maximal

- Permet d'atteindre des dizaines de Mpps
- Scalabilité linéaire : 1 cœur = 1 queue

➡️ Le CPU est utilisé comme un ASIC logiciel.

Pourquoi le CPU est à 100 % même sans trafic ?

Même sans paquets :

- la boucle tourne
- le polling continue
- le CPU exécute des instructions vides

Donc :

- top / htop montrent 100 %
- mais ce n'est pas du travail utile, c'est de l'attente active

📌 C'est un design intentionnel, pas une fuite CPU.

Aspect	IRQ classique	DPDK polling
CPU idle	faible	nul
Latence	variable	minimale
Débit max	limité	très élevé
Prévisibilité	moyenne	excellente

Comment fonctionne un PMD Intel/Mellanox (fonctionnement interne)

Un PMD (Poll Mode Driver) est un driver réseau DPDK en espace utilisateur, totalement indépendant du kernel.

Il se compose de quatre couches principales :

```
Application
    ↓
DPDK RTE API (rte_eth_dev_*)
    ↓
PMD (Intel/Mellanox) — user-space hardware driver
    ↓
NIC hardware (queues, registers, DMA)
```

Pipeline interne d’un PMD (simplifié)

Initialisation :

Lors de rte_eth_dev_configure() :

- le PMD lit les registres PCIe,
- configure les queues RX/TX,
- alloue les rings (descriptor rings) directement en hugepages,
- effectue les mappings DMA.

Exemple (Intel ixgbe, i40e) :

- RX descriptor ring = tableau de struct ixgbe_rx_desc,
- TX descriptor ring = tableau de struct ixgbe_tx_desc.

RX – Réception des paquets :

Le PMD :

1. lit l'état du tail pointer du ring RX dans le registre NIC,
2. vérifie quels descripteurs contiennent un paquet,
3. remplit des rte_mbuf avec :
    - adresse DMA → buf_addr,
    - longueur → data_len,
    - metadata (RSS hash, VLAN, offloads),
1. avance le tail dans le ring RX hardware,
2. renvoie un burst de mbufs via rte_eth_rx_burst().

TX – Envoi des paquets :

Lors d’un TX :

1. le PMD écrit l’adresse et la taille dans un TX descriptor,
2. pousse le tail pointer dans le hardware,
3. la NIC lit les buffers en DMA,
4. quand envoyé, le NIC avance son head pointer.

Le PMD nettoie ensuite (free des mbufs).

Poll Mode Driver (PMD)

Dans DPDK, un PMD est un pilote réseau (ou crypto, ou event, etc.) fonctionnant en mode polling, c'est-à-dire sans interruptions.

Il est conçu pour fonctionner entièrement en espace utilisateur, sans interruption matérielle, en utilisant un mécanisme de polling continu (scrutation active).

- **P : Poll**

  Le driver ne repose pas sur des interruptions NIC.
  Il tourne dans une boucle infinie (busy loop) et interroge en permanence les queues RX/TX.
  → Cela élimine le coût des interruptions (ISR), des softirq et du scheduler kernel.

- **M : Mode**

  Ce n'est pas une API classique : le driver fonctionne dans un mode particulier, optimisé pour :
  - haut débit,
  - faible latence,
  - traitement par lots (burst).

- **D : Driver**

  Le PMD remplace complètement le driver kernel de GNU/Linux (ex. ixgbe, ice, mlx5).

  Il accède directement au hardware via PCIe grâce au mapping mémoire fait par DPDK.

PMD Intel / Mellanox

Pourquoi DPDK utilise des PMD ?

- ultra faible latence
- throughput extrême (10–400 Gbps)
- zéro interruption
- zéro copie inutile
- pipeline user-space optimisé

Exemples :

- ixgbe : 10GbE
- i40e : 40GbE
- ice : 100GbE
- mlx5 : Mellanox ConnectX-5+

Caractéristiques spécifiques par vendor :

- Intel ixgbe/i40e/ice
  - moteur RSS flexible
  - Flow Director
  - support modéré des tunnels

- Mellanox mlx5 (ConnectX-4/5/6)
  - le PMD s’appuie sur mlx5dv_devx
  - accélération flow-table hardware (TCAM + steering)
  - support VXLAN/Geneve hardware
  - performance très supérieure pour SR-IOV, virtio, OVS-DPDK

- **Offloads énergétiques / accélérations :**

Les PMD Intel/Mellanox supportent :

- checksum offload IP/UDP/TCP
- TSO/GSO
- VLAN add/remove
- RSS (hash distribué)
- Flow Director (Intel)
- Mellanox RegEx, parsing avancé, steering matériel
- multi-queue massive (jusqu'à 64–128 queues)

Un « offload » (ou « déchargement ») désigne le fait de transférer un travail coûteux (CPU, mémoire, énergie) d'un composant vers un autre plus adapté, par exemple du processeur généraliste vers une carte réseau ou un accélérateur matériel.

Offloads NIC

DPDK expose tous les offloads modernes :

- **LRO / GRO** :
  - LRO (Large Receive Offload) : la NIC regroupe plusieurs segments TCP reçus consécutifs pour un même flux en un gros paquet avant de les livrer au CPU, ce qui réduit le nombre de paquets à traiter et donc les interruptions et le travail au niveau IP/TCP.
  - GRO (Generic Receive Offload) : même objectif, mais en software côté DPDK ; DPDK fournit une lib qui fusionne plusieurs petits paquets en un plus grand mbuf, ce qui donne un effet similaire au LRO même si la carte ne le supporte pas.
- **TSO / GSO** :
  - TSO (TCP Segmentation Offload) : côté émission, l'application envoie un gros segment TCP (par exemple 64 Ko), et la NIC le découpe en MSS-sized segments avec mise à jour des headers et checksums, évitant au CPU de faire la segmentation.
  - GSO (Generic Segmentation Offload) : implémentation software dans DPDK, qui segmente un gros paquet en plusieurs plus petits (TCP/UDP/VXLAN…) lorsque la carte ne fournit pas de TSO matériel, en conservant des API DPDK similaires.
- **Checksum IP/UDP/TCP** :
  - En RX : la NIC calcule les checksums IP/L4 et place dans le mbuf un état « checksum OK/KO », ce qui évite au CPU de recalculer chaque checksum et se contente de vérifier un flag.
  - En TX : l'application prépare les headers avec les champs de checksum à zéro ou pseudo-remplis, et la NIC calcule/insère les valeurs finales lors de l'envoi, ce qui réduit la charge CPU sur les paquets sortants.

- **VLAN strip/insertion** :
  - VLAN stripping (RX) : la NIC supprime le tag 802.1Q/802.1ad du header Ethernet et place l'ID de VLAN et les flags dans des champs dédiés du mbuf, ce qui simplifie le traitement des paquets dans le plan de données.
  - VLAN insertion (TX) : à l'inverse, l'application renseigne le VLAN ID dans l'mbuf et la NIC se charge d'insérer le tag VLAN dans le frame Ethernet au moment de l'émission.
- **RSS (hash distribué sur queues)** :
  - RSS (Receive Side Scaling) calcule un hash sur certains champs (5-tuple IP, ports, éventuellement tunnel) et choisit la RX queue matérielle à partir de ce hash, typiquement via une table de redirection.
  - Chaque queue est ensuite polled par un core DPDK différent, ce qui permet de paralléliser le traitement des flux tout en conservant l'affinité « un flux → une queue → un core » pour limiter le reorder.
- **Flow director** :
  - Le flow director est un moteur de classification matérielle programmable par la DPDK : on installe des règles (match sur 5-tuple, VLAN, tunnel, etc.) pour diriger un flux vers une queue précise, appliquer un marquage ou un drop.
  - Contrairement au RSS, qui ne fait qu'un hash « statique », le flow director permet une redirection fine par règles, utile pour l'isolation de flux, le load-balancing avancé ou certains use cases NFV.

- **GTP/U/N parsing (Mellanox)** :
  - Les NIC Mellanox modernes savent parser les headers GTP-U/GTP-C/GTP-N des réseaux mobiles, extraire les champs-clés (TEID, ports, IP interne) et les exposer à la logique RSS / flow director.
  - DPDK s’appuie sur ces capacités : les paquets GTP peuvent être distribués correctement sur les queues (RSS aware GTP) ou matchés par des règles matérielles, ce qui est crucial pour des plans de données 4G/5G à très haut débit.

RX/TX Queues & Pipelines

Multiples RX/TX queues

Chaque queue correspond à un ring hardware de la NIC.

On associe généralement 1 queue ↔ 1 lcore.

Pipeline typique :

NIC RX Queue → PMD → RX burst → traitement → TX burst → PMD → NIC TX Queue

Burst mode

DPDK fonctionne par batch :

```
uint16_t nb_rx = rte_eth_rx_burst(port_id, queue_id, pkts, BURST_SIZE);
```

Pas de paquet ? → 0 rempli, boucle continue.

Model pipeline multi-core

[Core 1] RX → Parse → Classify → Ring →

[Core 2] Firewall → NAT →

[Core 3] TX

Communication inter-lcore par ring lockless (rte_ring).

La Flow API : SDN et offload matériel

La Flow API

- Flow API permet de programmer des pipelines dans la NIC :

Exemples (Flow rule simple (match + action) :

```
struct rte_flow_attr attr = {
    .ingress = 1
};


struct rte_flow_action actions[] = {
    { .type = RTE_FLOW_ACTION_TYPE_DROP },
    { .type = RTE_FLOW_ACTION_TYPE_END }
};
```

Types de match supportés :

- IPv4, IPv6
- TCP, UDP, SCTP
- VLAN
- MPLS
- Tunnel (VXLAN, Geneve)
- Metadata RSS
- Flow Director (Intel)
- Registre hardware Mellanox (GID, vport)

Optimisations CPU, NUMA,
cache & prefetch

Pinning / Affinité CPU

Essentiel pour éviter les migrations :

- `rte_cpuset_t cpuset;`

type DPDK compatible avec les macros standard de gestion de jeux de CPU (équivalent à cpu_set_t), qui représente un ensemble de cœurs sur lesquels un thread a le droit de s'exécuter.

- `CPU_ZERO(&cpuset);`

initialise la structure en vidant l'ensemble, donc aucun CPU autorisé au départ.

- `CPU_SET(2, &cpuset);`

ajoute le CPU logique numéro 2 dans cet ensemble ; on demande donc « seulement le CPU 2 ».

Utilisation du Prefetch

- `rte_prefetch0(rte_pktmbuf_mtod(m, void *));`

sert à précharger en cache le début des données du paquet pointé par m avant de les traiter.

- `pthread_setaffinity_np(pthread_self(), sizeof(cpuset), &cpuset);`

appelle l'API POSIX pour restreindre le thread courant (pthread_self()) à l'ensemble de cœurs décrit par cpuset, ici uniquement le CPU 2.

→ En pratique, cela sert en data plane (DPDK, RT, etc.) à associer un thread sur un cœur précis pour mieux contrôler la latence, éviter les migrations de contexte entre cœurs et exploiter au mieux les caches L1/L2 de ce cœur.

Optimisations élémentaires

- éviter les locks
- utiliser rings SP/SC dès que possible
- alignement mémoire 64 bytes
- minimiser les branches (likely/unlikely)
- regrouper les actions (batch processing)

Mesure des performances

- **Cycle counters :**

```
uint64_t t0 = rte_rdtsc();
...
uint64_t t1 = rte_rdtsc();
printf("cycles = %lu\n", t1 - t0);
```

- **PMD stats :**

```
rte_eth_stats_get(portid, &stats);
Stats utiles :
ipackets : reçus
opackets : envoyés
ierrors, oerrors
rx_nombuf : manque d’mbufs → augmenter mempool
```

- **Perf tools :**

rte_latencystats

rte_pdump

top, htop

perf (cycles, LLC misses)

Interface KNI

Pourquoi KNI existe ?

Une interface KNI (Kernel NIC Interface) dans DPDK est un mécanisme qui permet à une application DPDK en espace utilisateur (user-space) de créer une interface réseau visible par le noyau Linux, comme si c'était une interface réseau classique (ethX).

Permettre de faire communiquer le monde user-space DPDK (qui traite les paquets très rapidement en contournant le kernel) avec le stack réseau Linux (iproute2, netfilter/iptables, services système, etc.).

DPDK utilise poll-mode drivers (PMD) et évite le noyau pour obtenir des performances élevées.

Mais parfois, on a besoin que le kernel voie les paquets, par exemple pour :

- utiliser tcpdump ou wireshark
- laisser NetworkManager configurer une interface
- faire transiter certains paquets vers iptables / nftables
- exposer des statistiques via /sys
- permettre un contrôle ou une gestion via des outils standards (ip addr, ip link, etc.)

Le KNI fournit donc un pont rapide entre DPDK et le noyau.

Fonctionnement simplifié

L'application DPDK crée une interface KNI (ex : vEth0).

Cette interface apparaît dans Linux via ip link :

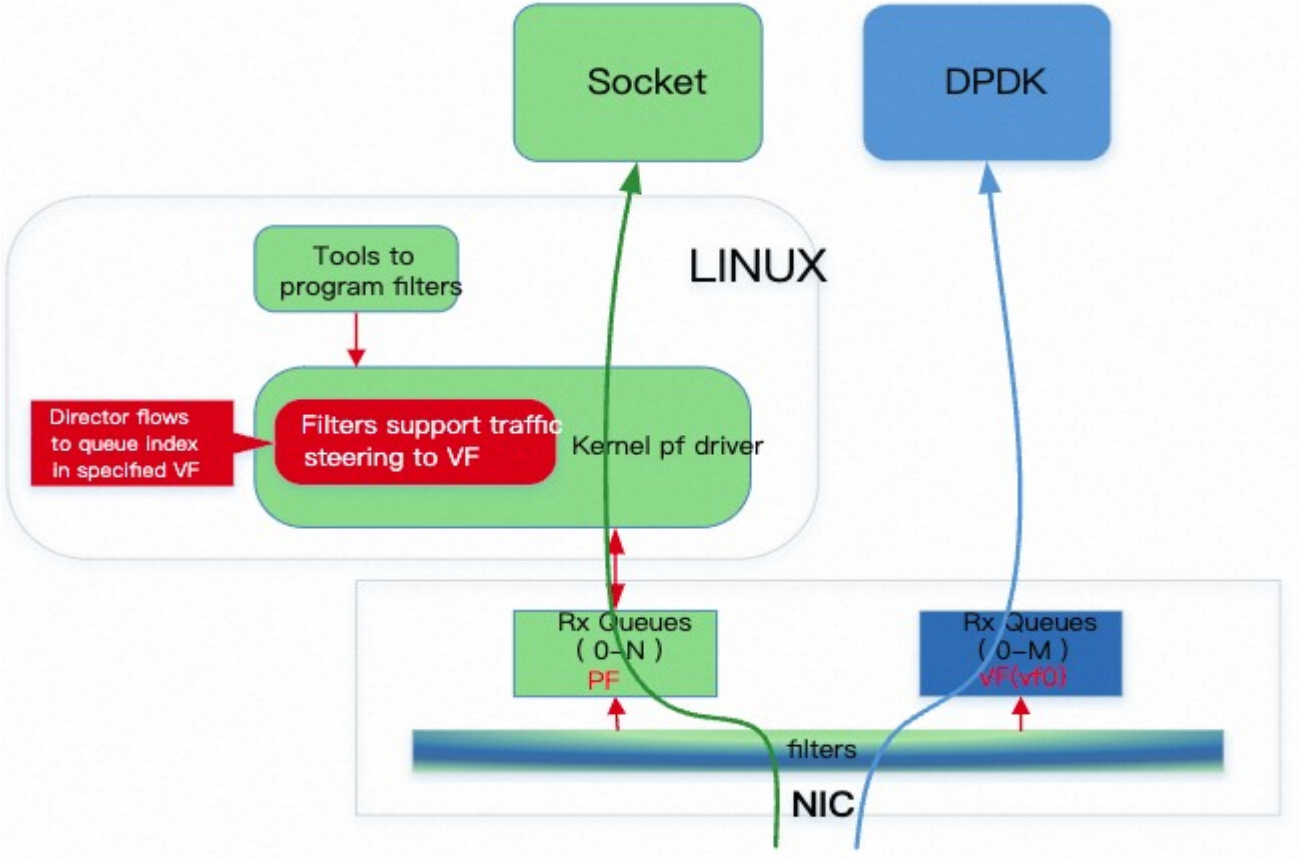
```
$ ip link show vEth0
```

L'application DPDK :

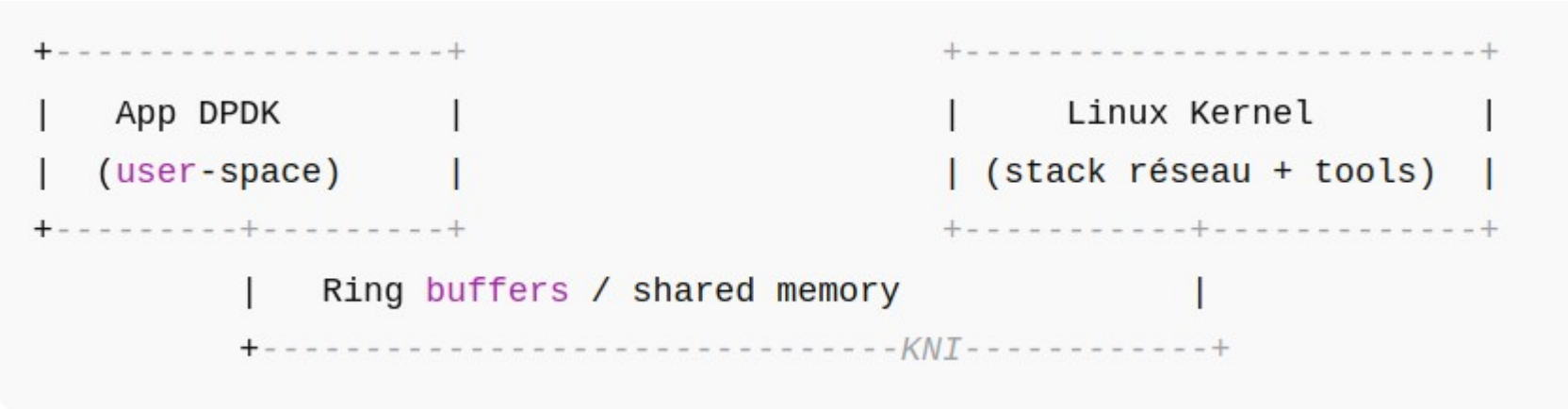
- peut envoyer des paquets vers le kernel (pour que celui-ci les traite via son stack réseau)
- peut recevoir des paquets que le kernel envoie sur cette interface

Techniquement, cela utilise un module kernel KNI fourni par DPDK et un anneau (ring buffer) partagé entre le user-space et le kernel.

Usage mixte (cartes modernes)



Architecture



Différences avec d’autres solutions récentes

KNI a plusieurs alternatives plus modernes :

- **AF_XDP (XDP / eBPF) : voir ma précédente présentation !**

Permet une communication rapide entre user-space et kernel, souvent plus performante et supportée nativement.

- **TAP / TUN**

Plus simple mais beaucoup plus lent que KNI.

TUN (pour TUNnel) opère en couche 3 : elle transporte des paquets IP (IPv4/IPv6) uniquement. Elle est typiquement utilisée pour les VPN « routés » : le trafic IP est encapsulé dans un tunnel point-à-point entre deux machines, sans véhiculer tout le trafic Ethernet (pas de broadcast, pas de MAC).

TAP (comme « network TAP ») opère en couche 2 : elle transporte des trames Ethernet complètes (en-têtes MAC + payload). Elle sert surtout à « étendre » ou virtualiser un réseau Ethernet : pont avec un bridge Linux, VM connectées comme sur un switch, ou VPN « bridgés » qui véhiculent aussi le broadcast et les protocoles non-IP.

Exemples de codes

Programme minimal optimisé

```
int main(int argc, char *argv[]) {
    rte_eal_init(argc, argv);
    rte_eth_dev_configure(port, 1, 1, &port_conf);
    rte_eth_rx_queue_setup(port, 0, nb_rxd, socket_id, NULL, mbuf_pool);
    rte_eth_tx_queue_setup(port, 0, nb_txd, socket_id, NULL);
    rte_eth_dev_start(port);

    while (1) {
        uint16_t nb_rx = rte_eth_rx_burst(port, 0, pkt_burst, BURST_SIZE);
        for (i = 0; i < nb_rx; i++) {
            struct rte_mbuf *m = pkt_burst[i];
            // traitement ici
            rte_eth_tx_burst(port, 0, &m, 1);
    }
}
```

Exemple minimal de code C utilisant un PMD (RX → TX)

```
#include <rte_eal.h>
#include <rte_ethdev.h>
#include <rte_mempool.h>
#include <rte_mbuf.h>

#define NB_MBUFS 8192
#define MBUF_CACHE_SIZE 250
#define BURST_SIZE 32

int main(int argc, char *argv[], char *env[])
{
    int ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_exit(EXIT_FAILURE, "Erreur EAL\n");

    uint16_t portid = 0;

    // Création du mempool
    struct rte_mempool *mbuf_pool = rte_pktmbuf_pool_create(
        "MBUF_POOL",
        NB_MBUFS,
        MBUF_CACHE_SIZE,
        0,
        RTE_MBUF_DEFAULT_BUF_SIZE,
        rte_socket_id()
    );
```

```
    // Configuration du port
    struct rte_eth_conf port_conf = {0};
    rte_eth_dev_configure(portid, 1, 1, &port_conf);

    // Setup RX/TX queue
    rte_eth_rx_queue_setup(portid, 0, 128, rte_socket_id(), NULL,
mbuf_pool);
    rte_eth_tx_queue_setup(portid, 0, 128, rte_socket_id(), NULL);
    rte_eth_dev_start(portid);

    printf("DPDK PMD started.\n");

    struct rte_mbuf *pkts[BURST_SIZE];

    while (1) {
        uint16_t nb_rx = rte_eth_rx_burst(portid, 0, pkts, BURST_SIZE);

        for (int i = 0; i < nb_rx; i++) {
            // traitement éventuel ici
            rte_eth_tx_burst(portid, 0, &pkts[i], 1);
        }
    }

    return 0;
}
```

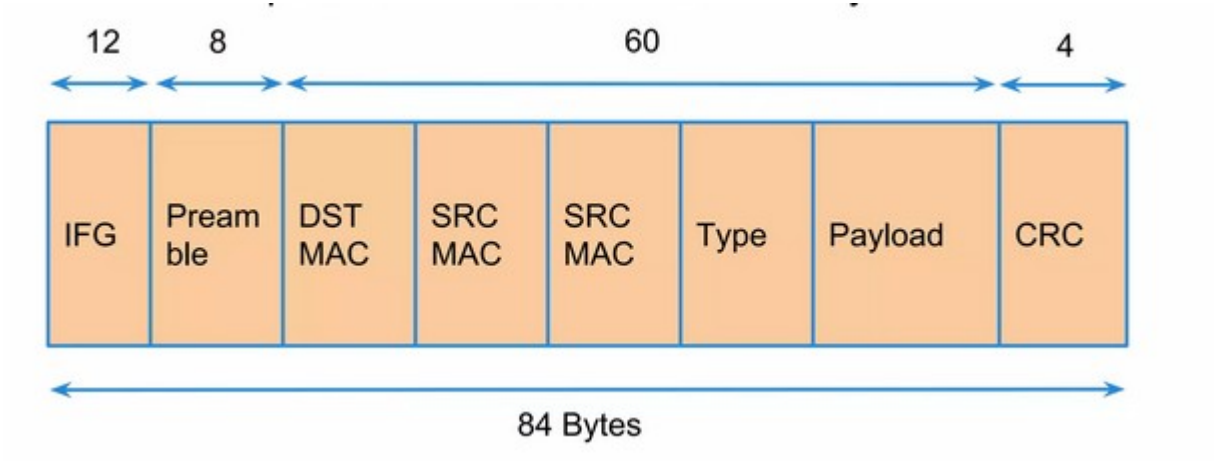
Détail

À quelle vitesse le logiciel peut-il fonctionner ?

14,88 millions de paquets de 64 octets par seconde sur une interface 10G.

1.8 GHz -> 1 cycle = 0.55 ns → 1,8 GHz → 1 cycle = 0,55 ns.

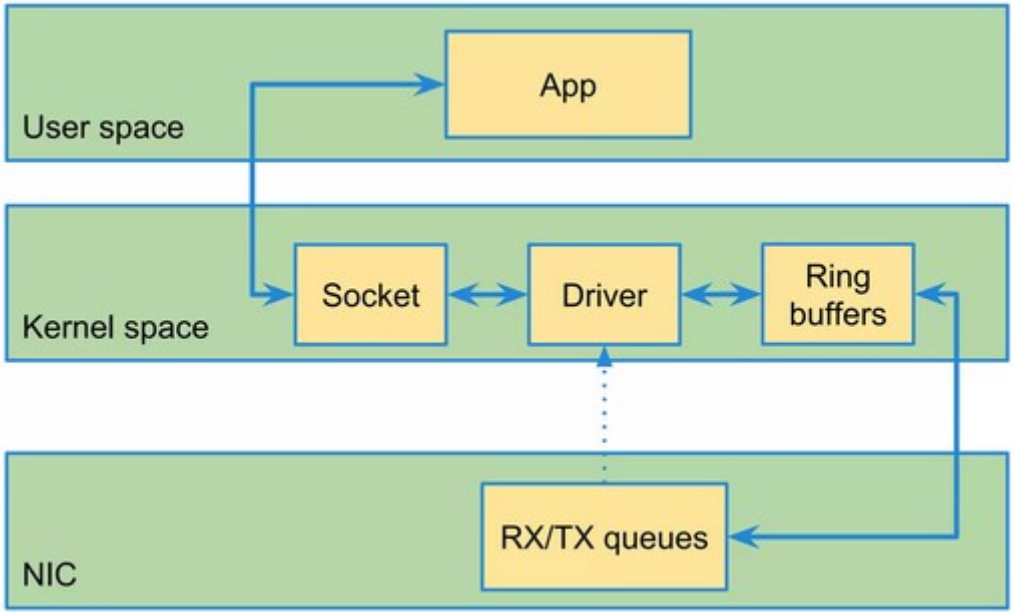
1 packet -> 67.2 ns = 120 clock cycles → 1 paquet → 67,2 ns = 120 cycles d'horloge.



Valeurs comparatives de vitesse.

- Vitesse CPU vers mémoire = 6–8 Go/s.
- Vitesse PCI-Express x16 = 5 Go/s.
- Accès à la RAM = 200 ns.
- Accès au cache L3 = 4 ns.
- Changement de contexte ≈ 1000 ns (3,2 GHz).

Traitement des paquets sous Linux.



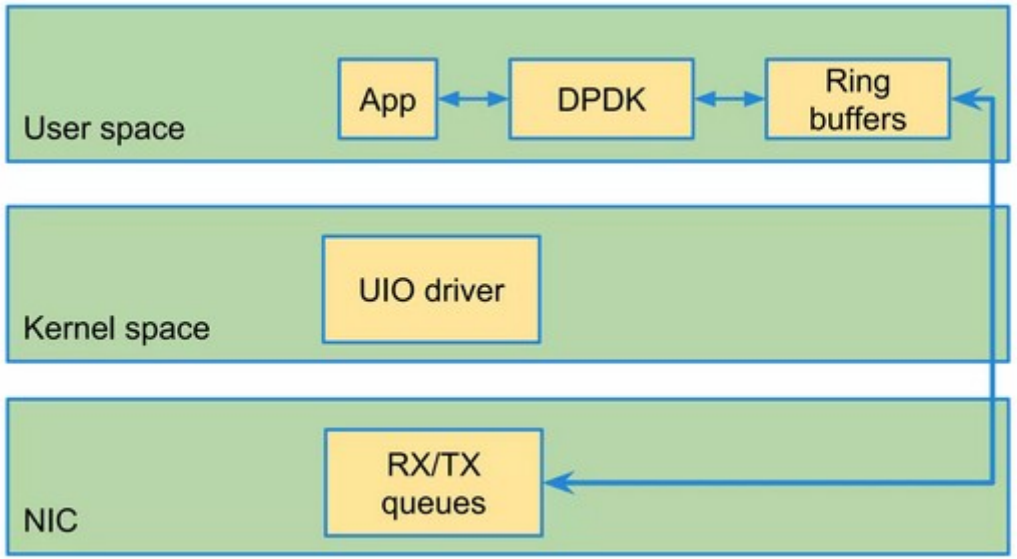
Surcharge du noyau Linux

- Appels système.
- Changement de contexte lors des E/S bloquantes.
- Copie des données du noyau vers l'espace utilisateur.
- Gestion des interruptions dans le noyau.

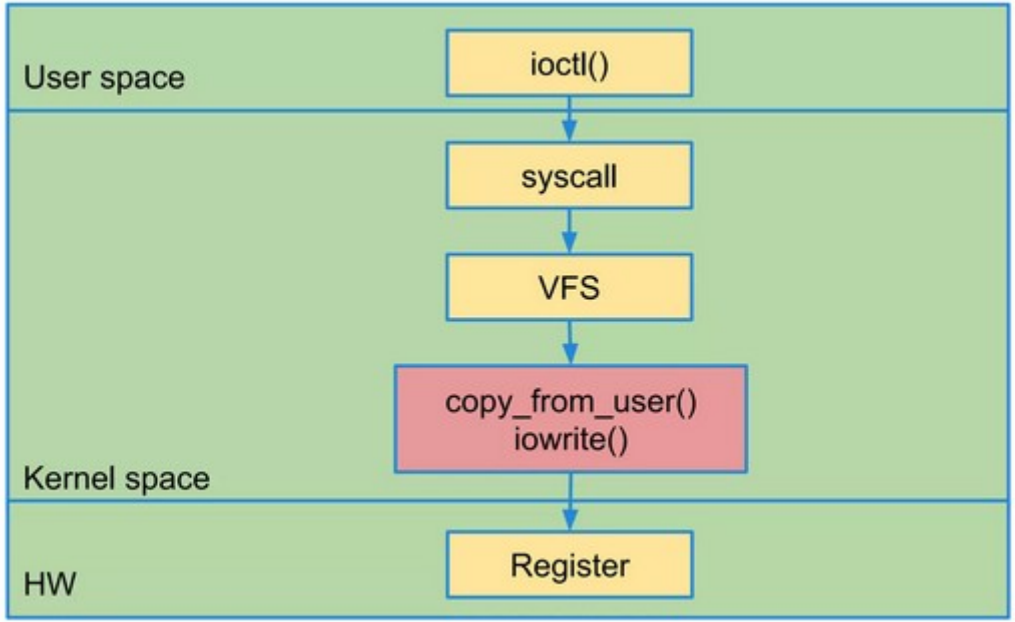
Coût du sendto :

Fonction	Activité	Temps (ns)
sendto	Appel système	96
sosend_dgram	Verrouillage de sock_buff, allocation d'un mbuf, copie en entrée	137
udp_output	Préparation de l'en-tête UDP	57
ip_output	Recherche de route, préparation de l'en-tête IP	198
ether_output	Recherche de l'adresse MAC, préparation de l'en-tête MAC	162
ixgbe_xmit	Programmation du périphérique	220
Total		950

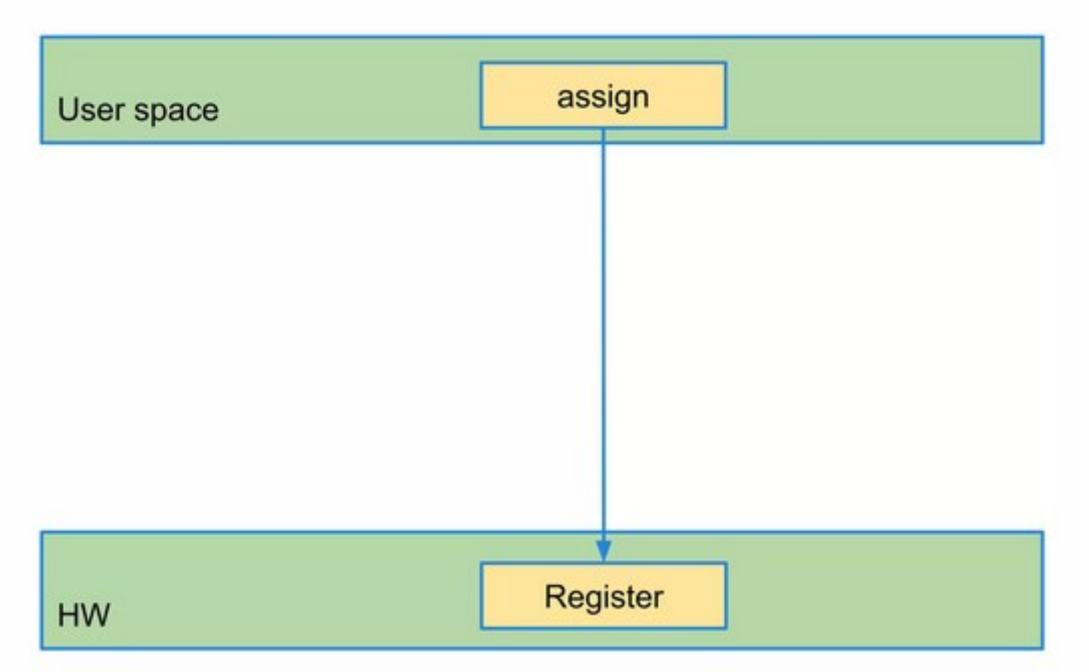
Traitement des paquets avec DPDK



Mise à jour d’un registre sous Linux



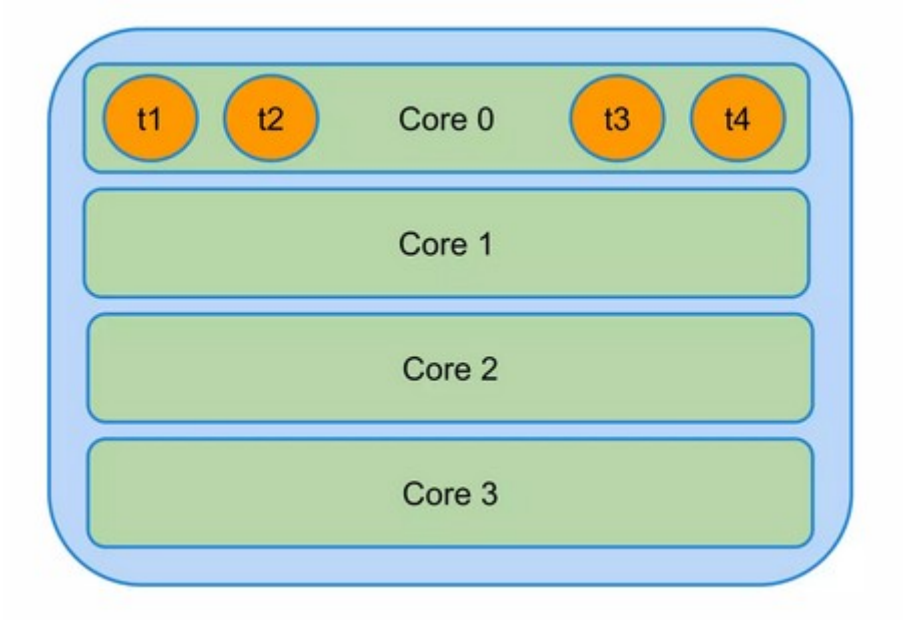
Mise à jour d’un registre avec DPDK



Ce qui est utilisé à l’intérieur de DPDK

- Affinité processeur (cœurs séparés).
- Grandes pages mémoire (pas de swap, TLB).
- UIO (aucune copie depuis le noyau).
- Scrutation/polling (pas de surcharge d’interruptions).
- Synchronisation sans verrou (éviter l’attente).
- Traitement des paquets par lots.
- Utilisation de SSE, prise en compte du NUMA.

Ordonnancement par défaut de Linux



Comment isoler un cœur pour un processus

- Pour diagnostiquer, utiliser top.

« top », appuyer sur « f », appuyer sur « j ».

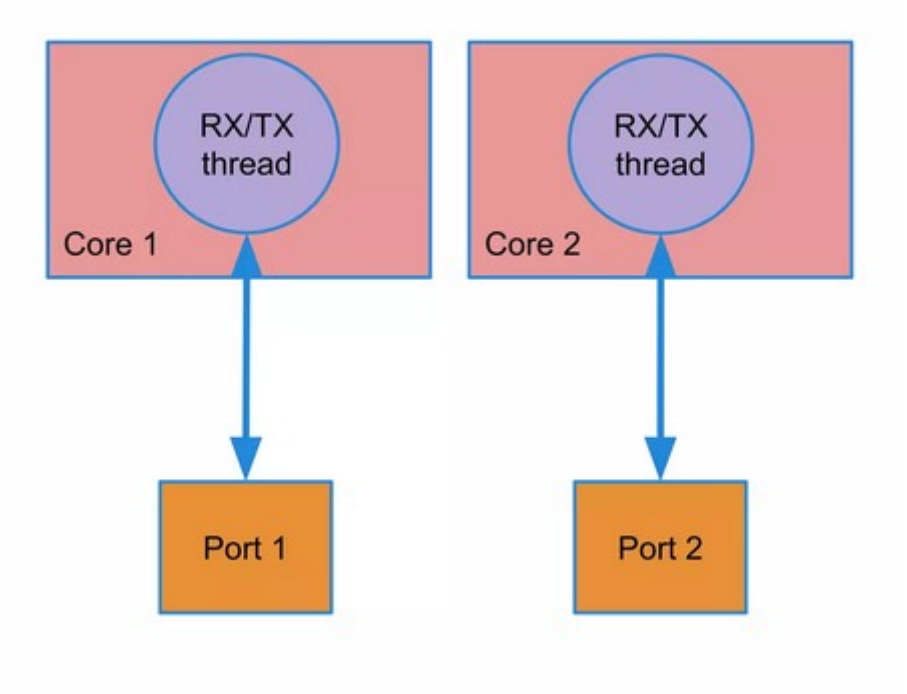
- Avant le démarrage, utiliser isolcpus.

« isolcpus=2,4,6 ».

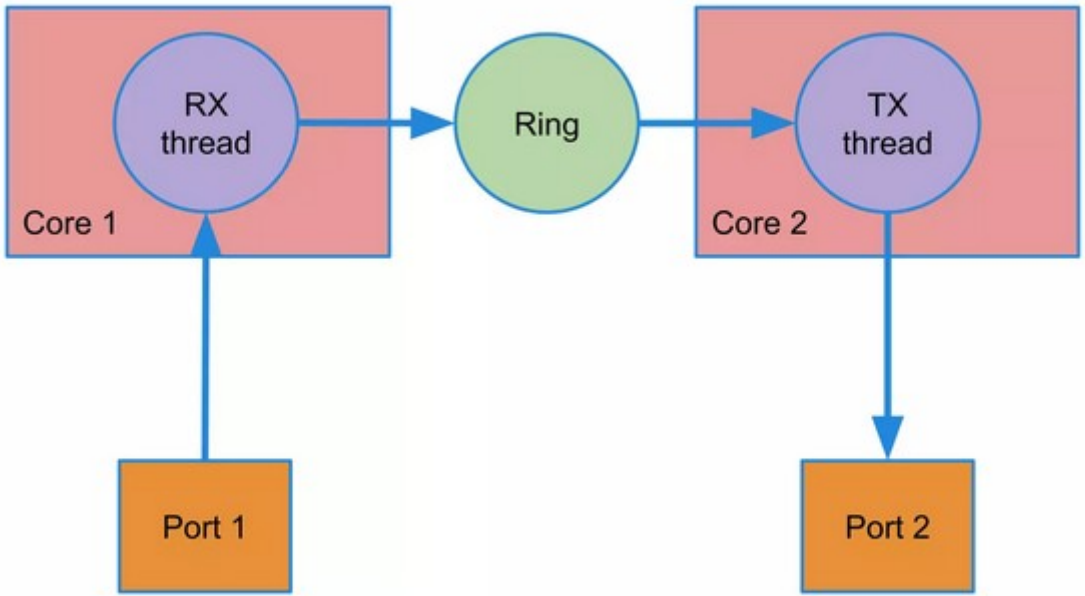
- After boot - use cpuset → Après le démarrage, utiliser cpuset.

« cset shield -c 1-3 », « cset shield -k on ».

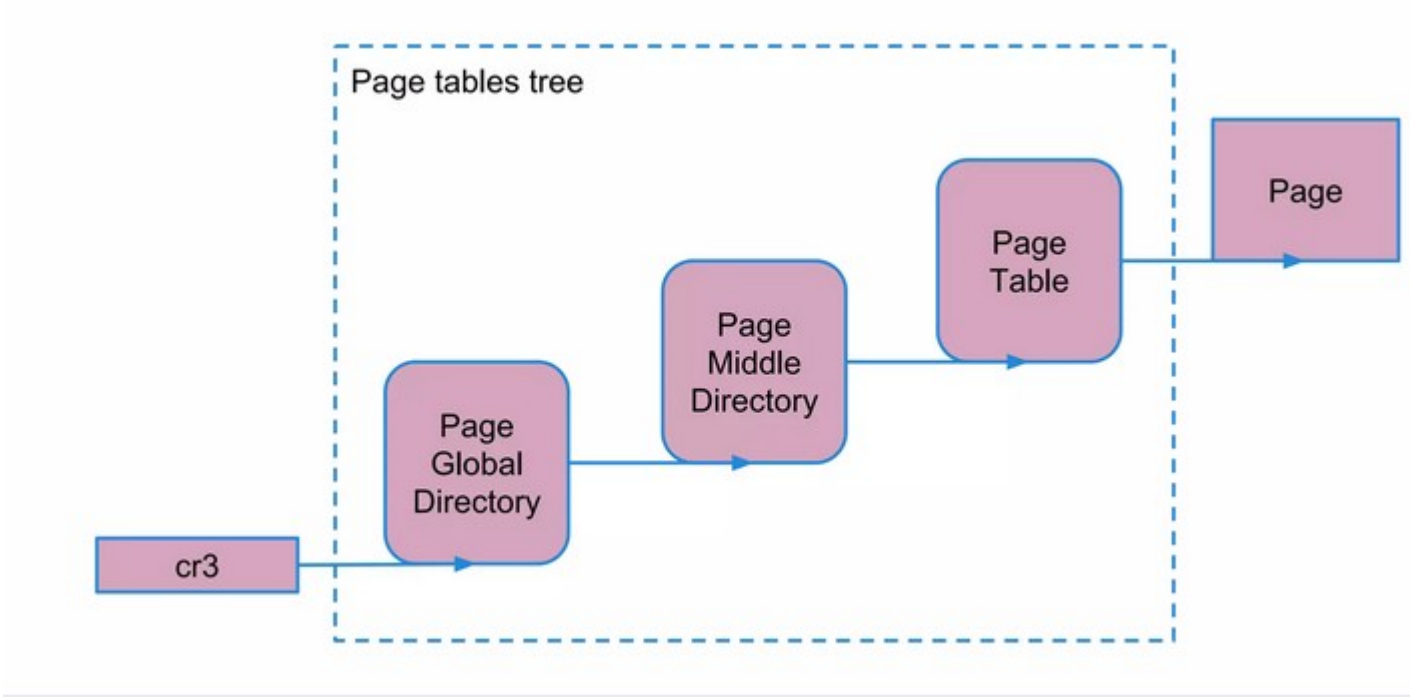
Modèle « exécution jusqu’à achèvement ».



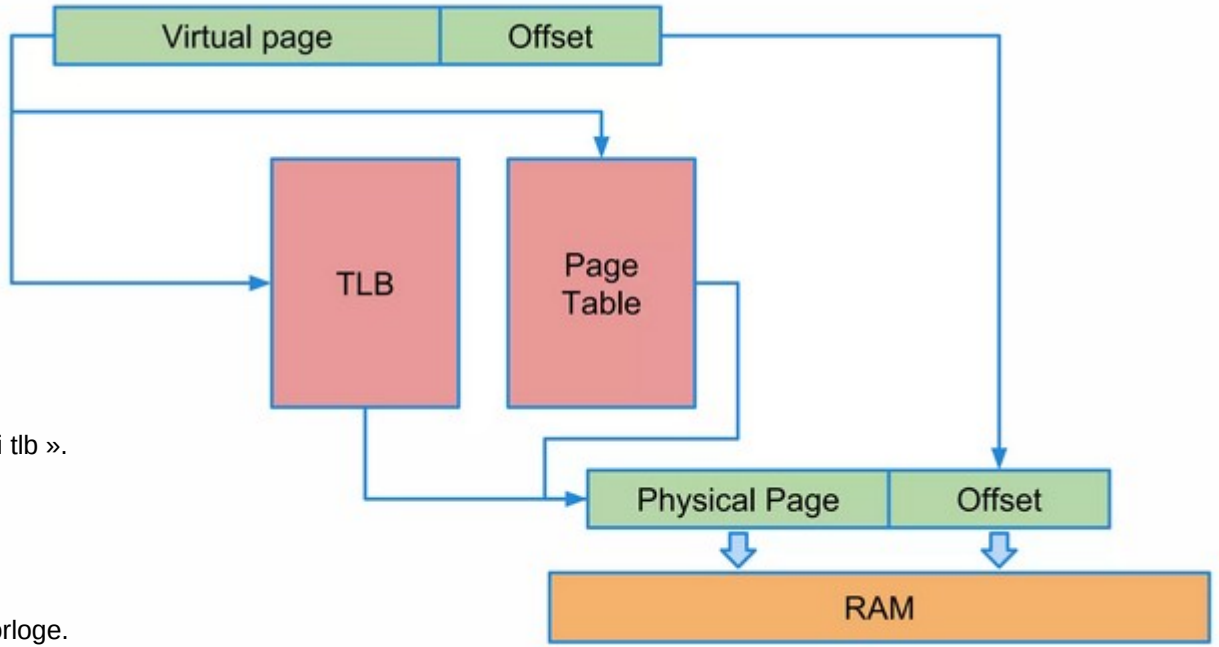
Modèle en pipeline



Modèle de pagination de Linux



TLB



Caractéristiques du TLB :

$ cpuid | grep -i tlb → « $ cpuid | grep -i tlb ».

taille : 12 à 4 096 entrées.

temps de hit : 0,5 à 1 cycle d'horloge.

pénalité de miss : 10 à 100 cycles d'horloge.

taux de miss : 0,01 à 1%.

C'est une ressource très coûteuse !

Hugepages

Avantage : utilisation optimisée du TLB, pas de swap.

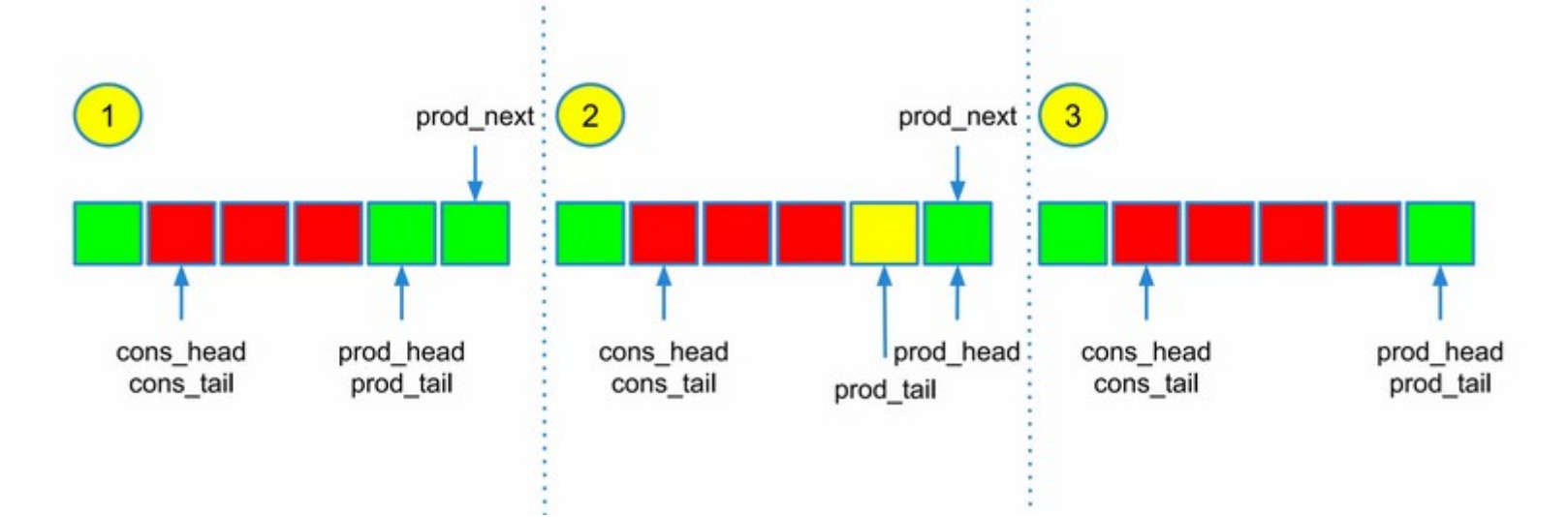
Taille d’une hugepage = 2 Mo.

Utilisation :

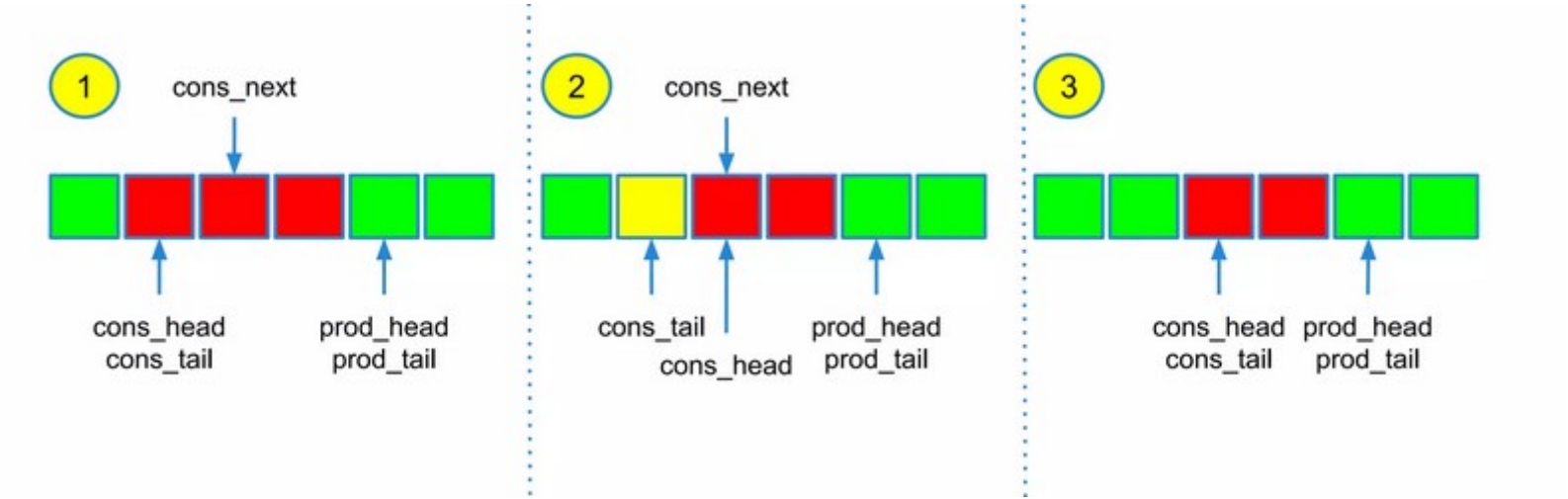
- $ mount hugetlbfs /mnt/huge
- mmap.

Bibliothèque – libhugetlbfs.

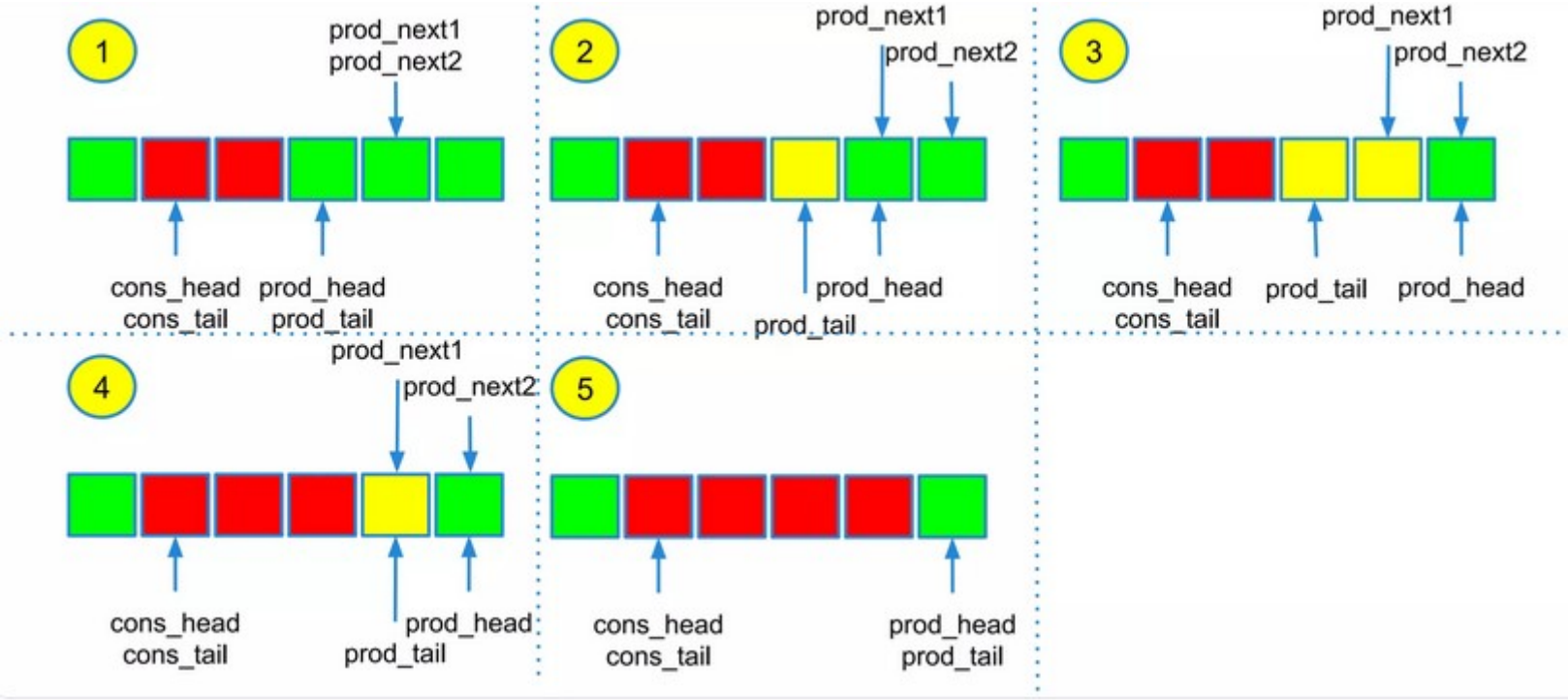
Anneau sans verrou (producteur unique)



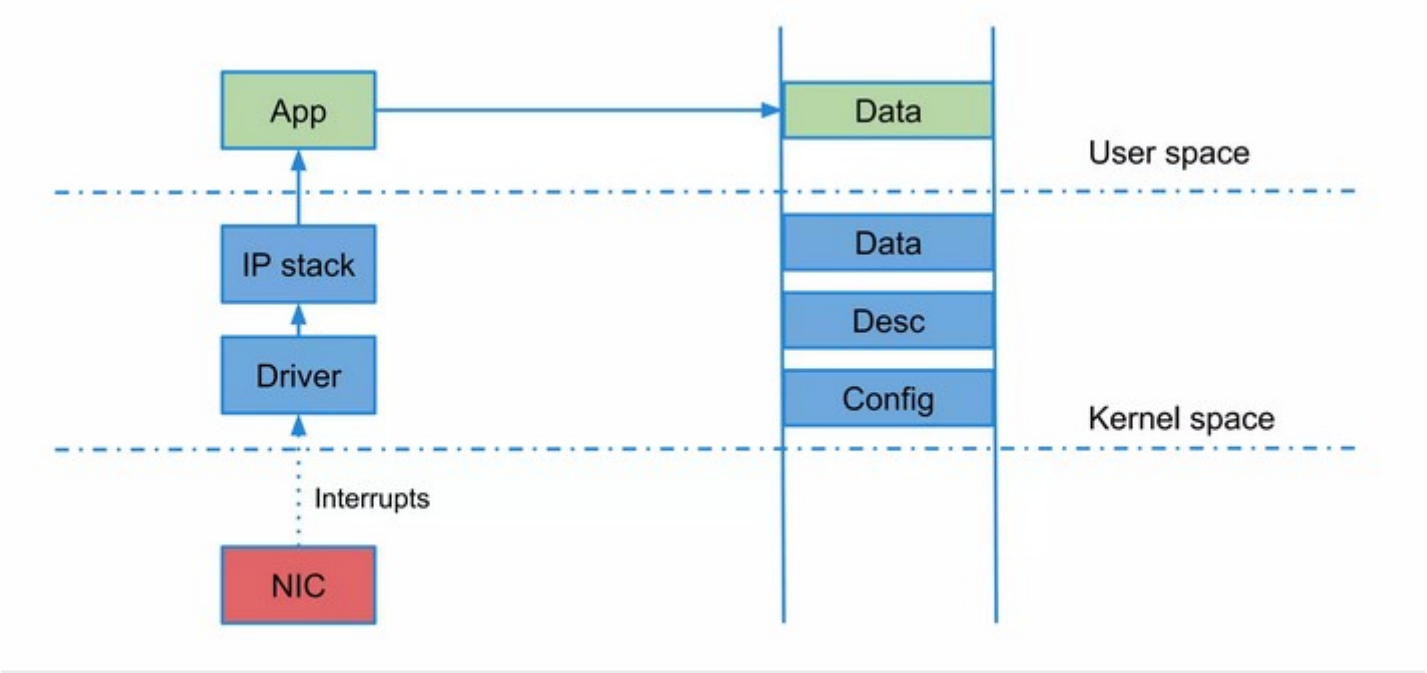
Anneau sans verrou (consommateur unique)



Anneau sans verrou (producteurs multiples)

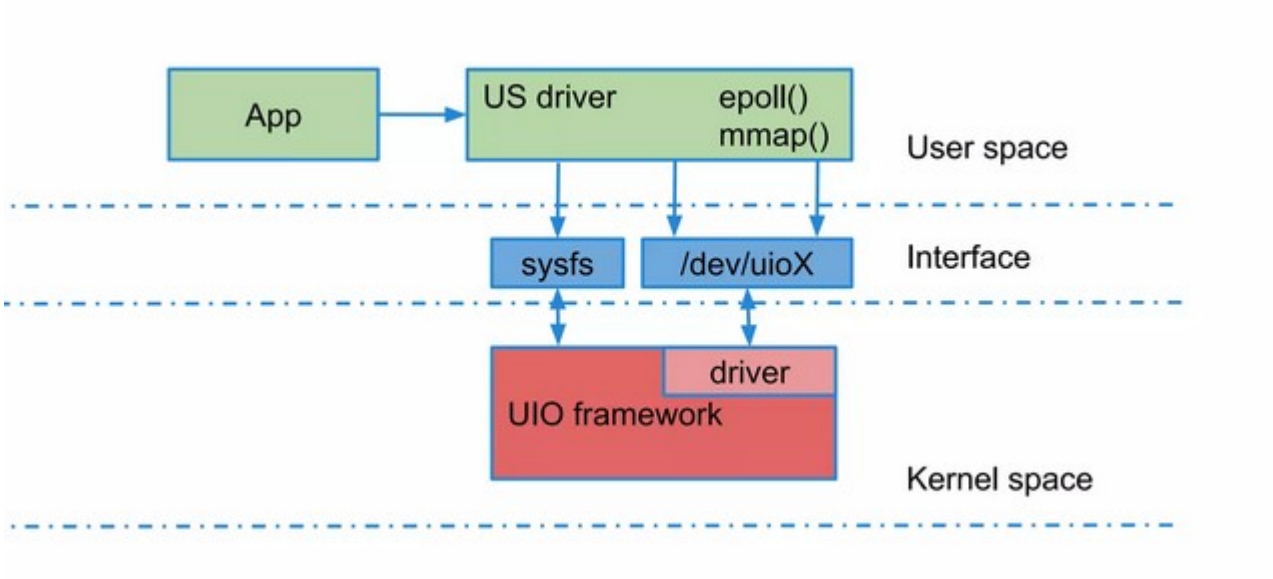


Pilote réseau en espace noyau

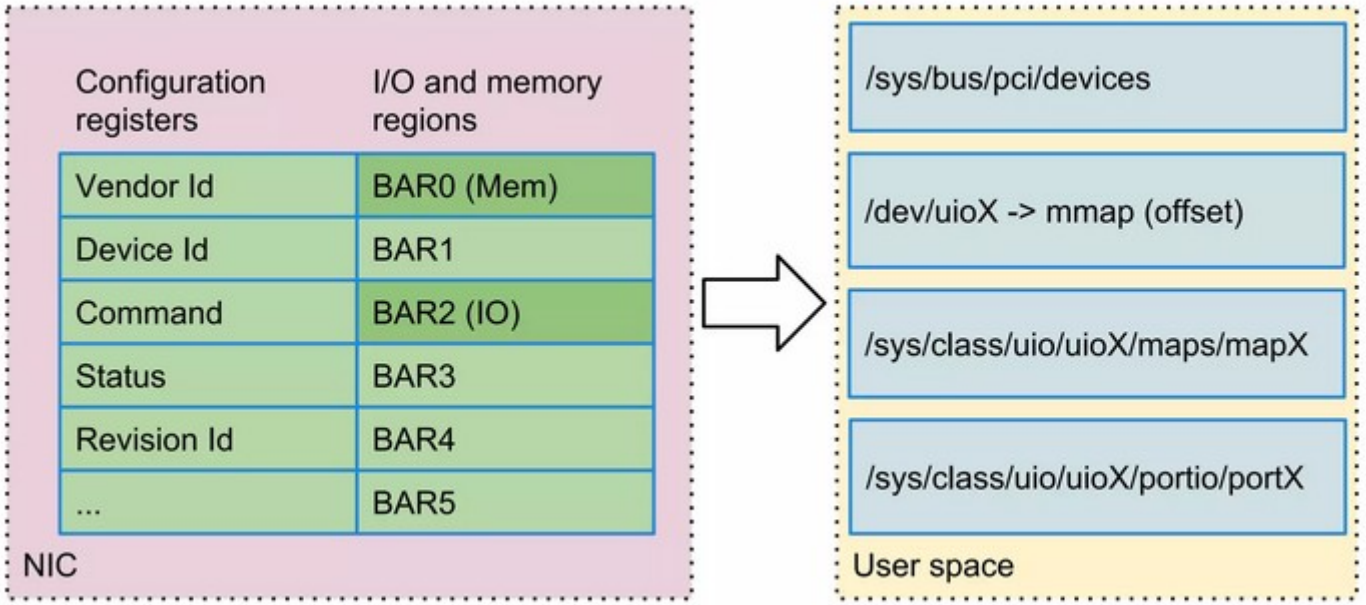


UIO Driver

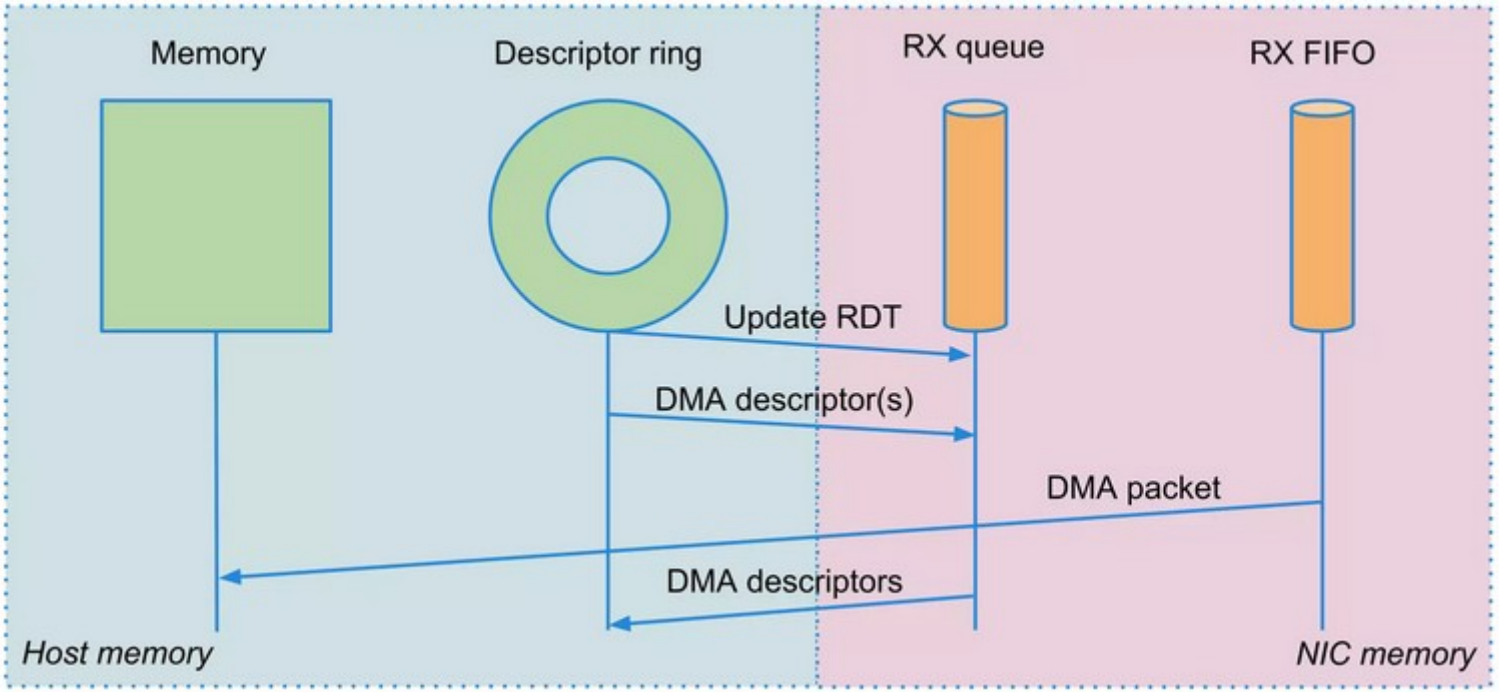
“Les périphériques les plus critiques, comme les interfaces réseau et les périphériques de stockage bloc, ne sont pas gérés directement depuis l’espace utilisateur mais via des pilotes en espace noyau.”



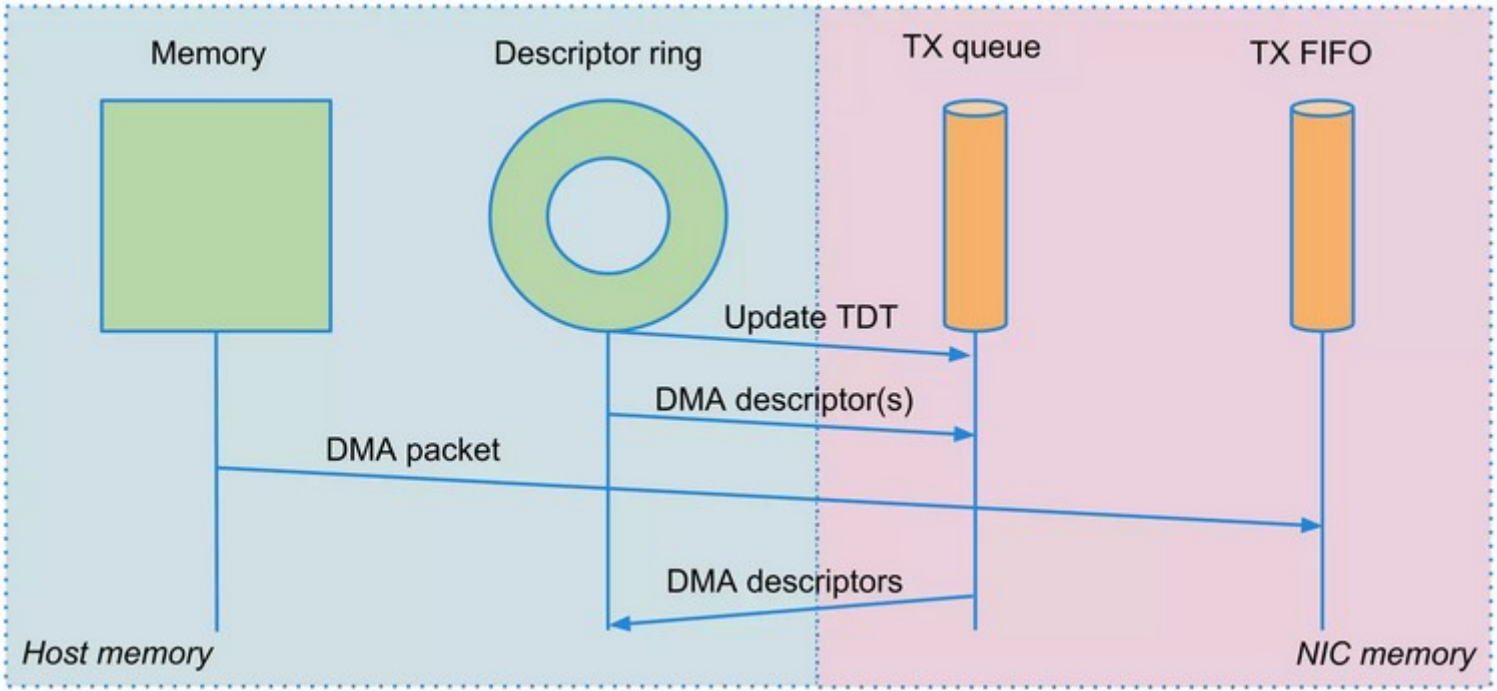
Accès au périphérique depuis l’espace utilisateur



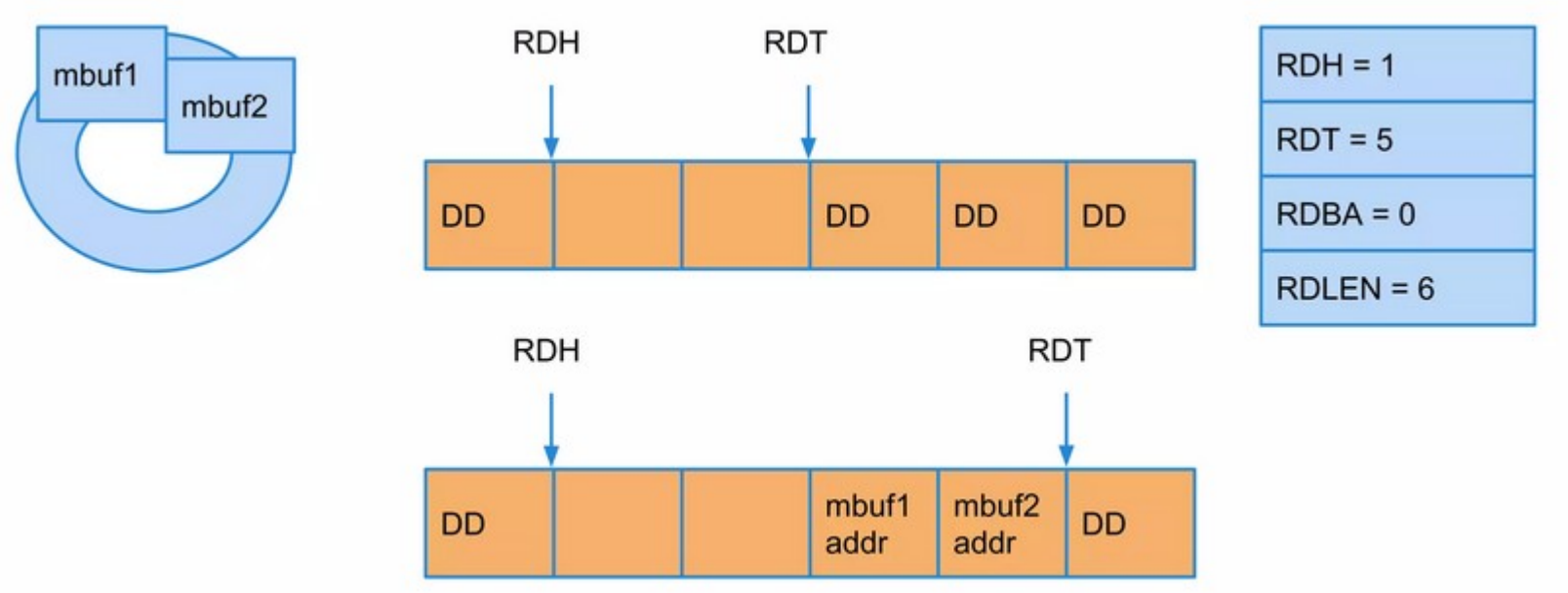
DMA RX


Memory
Descriptor ring
RX queue
RX FIFO
Update RDT
DMA descriptor(s)
DMA packet
DMA descriptors
Host memory
NIC memory

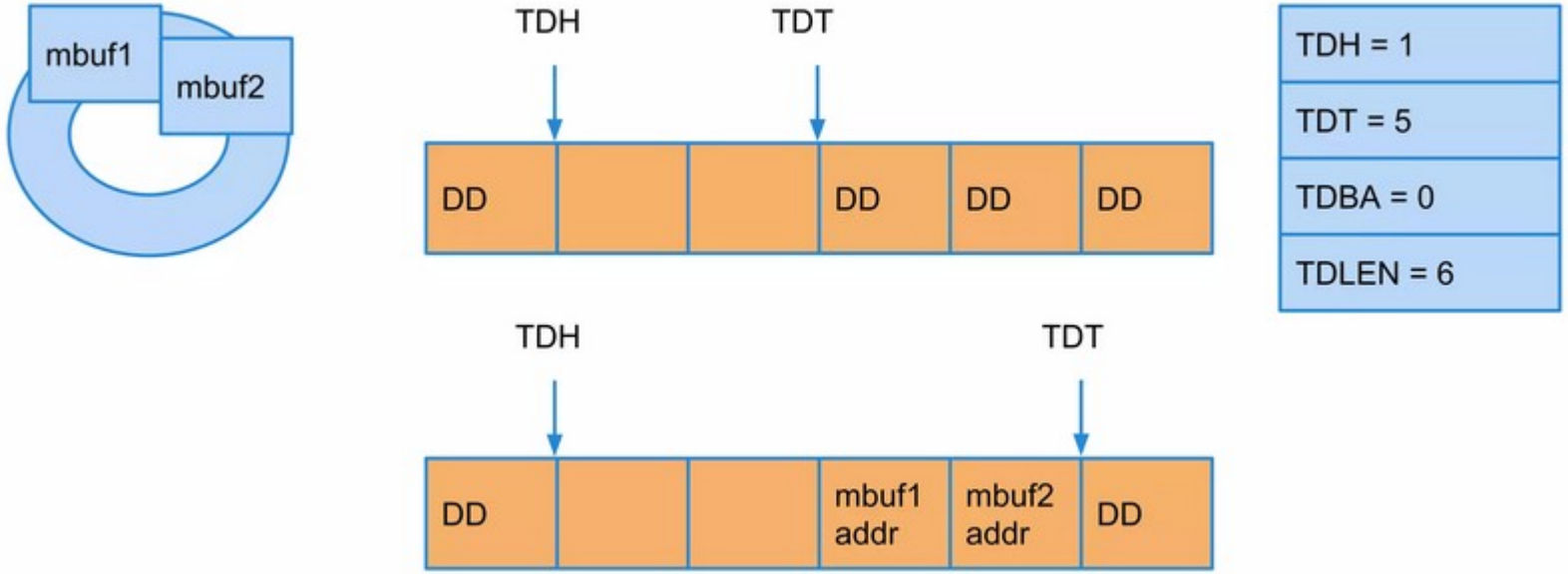
DMA TX


Memory
Descriptor ring
TX queue
TX FIFO
Update TDT
DMA descriptor(s)
DMA packet
DMA descriptors
Host memory
NIC memory

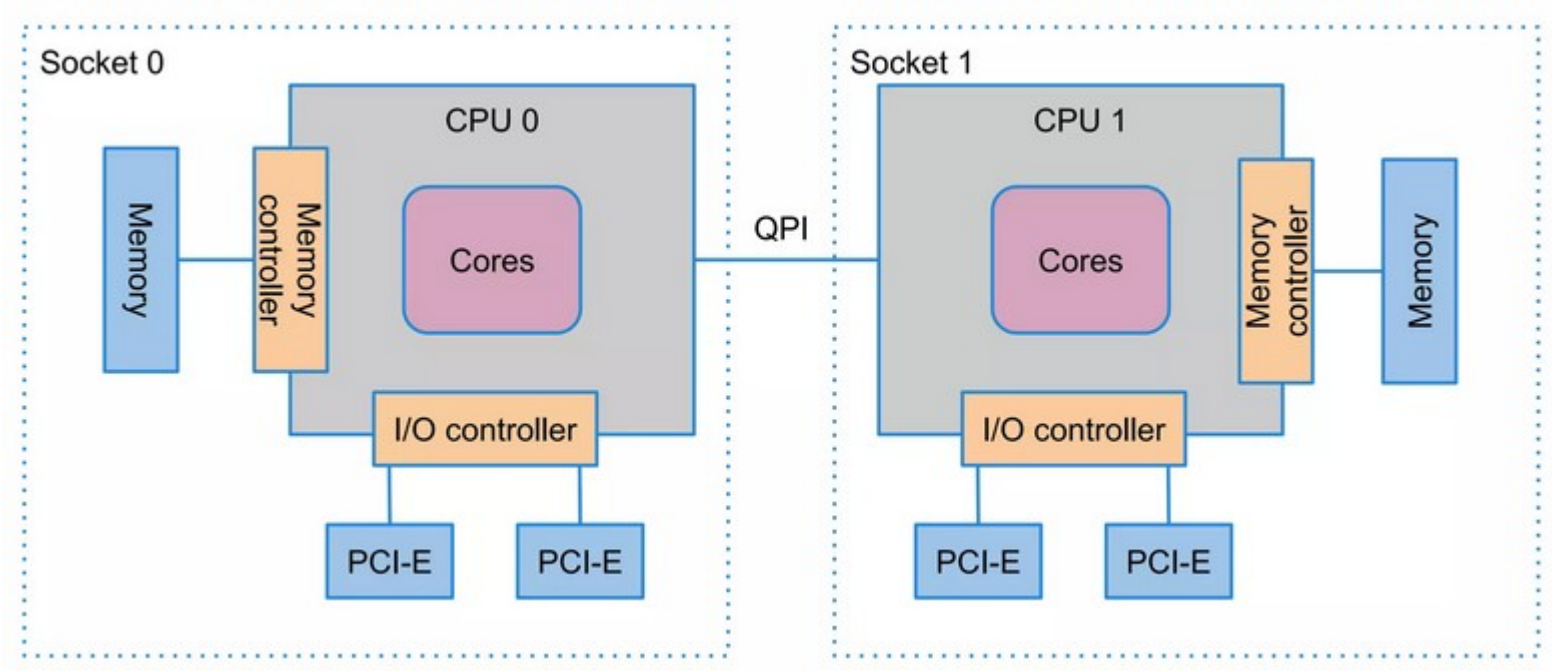
Réception côté logiciel



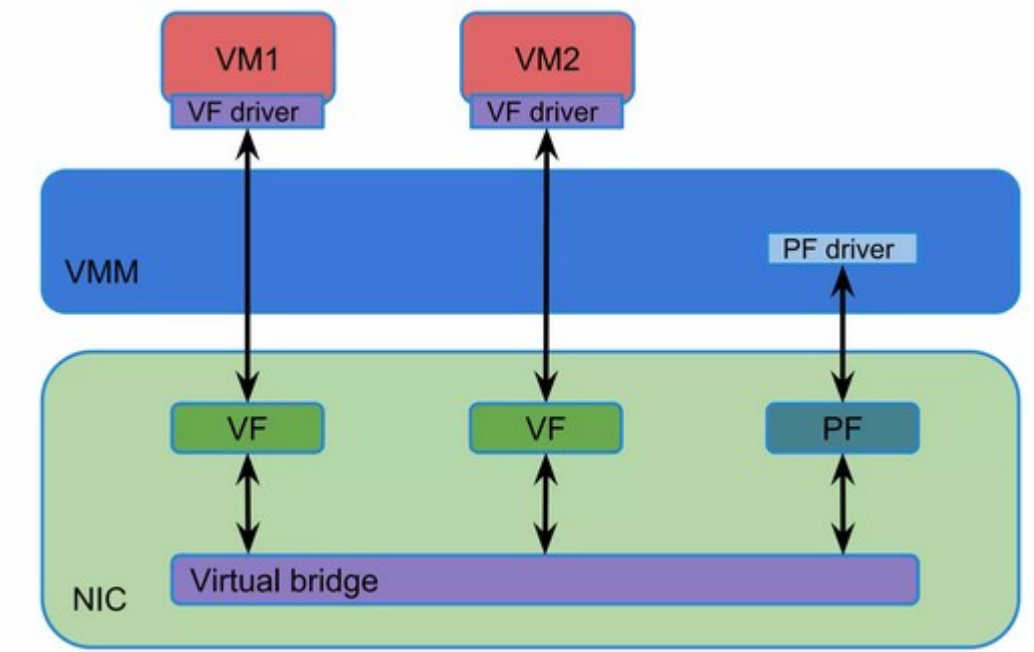
Transmission côté logiciel



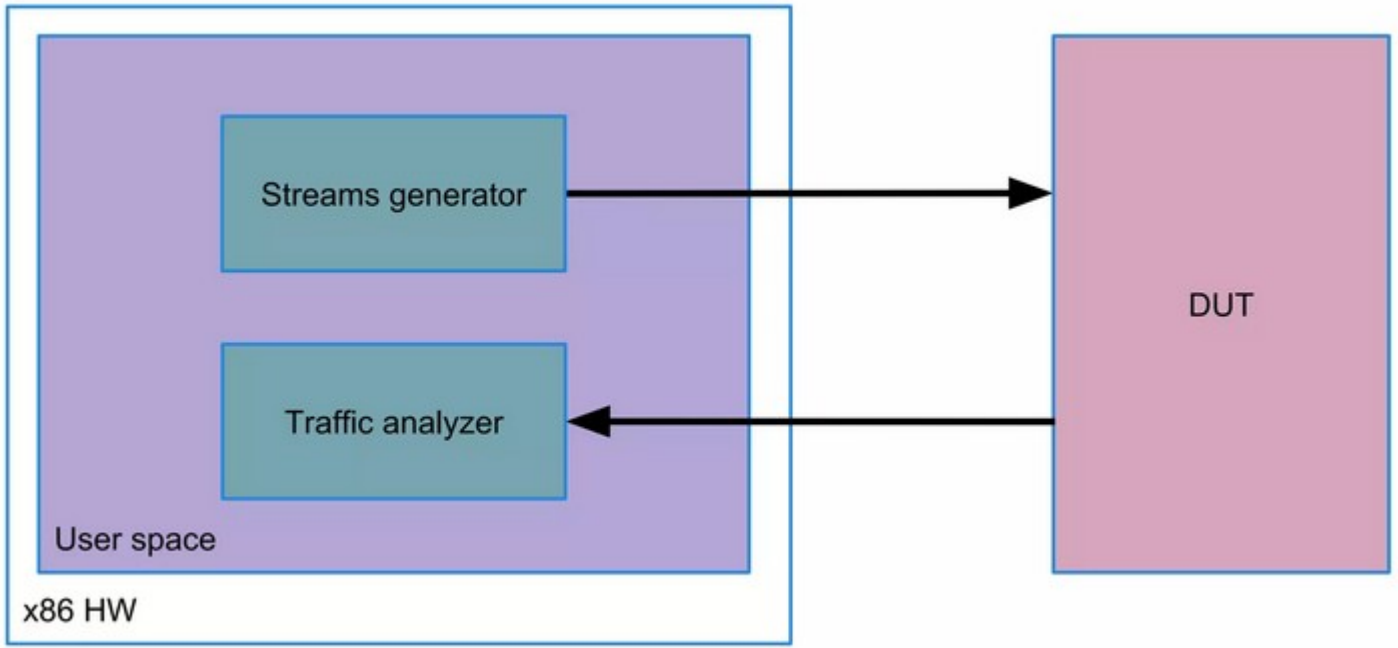
NUMA


Socket 0
CPU 0
Memory
Memory controller
Cores
I/O controller
PCI-E
PCI-E
QPI
Socket 1
CPU 1
Cores
Memory controller
Memory
I/O controller
PCI-E
PCI-E

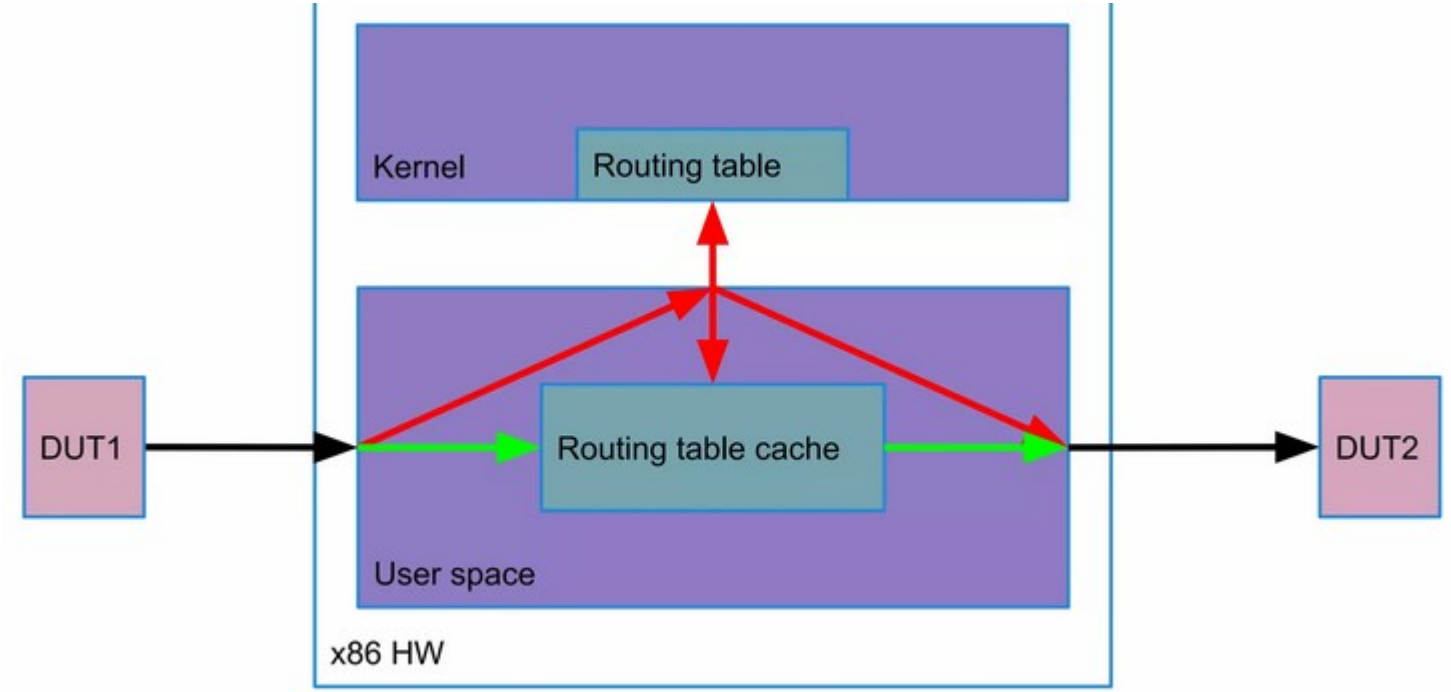
Virtualisation – SR-IOV



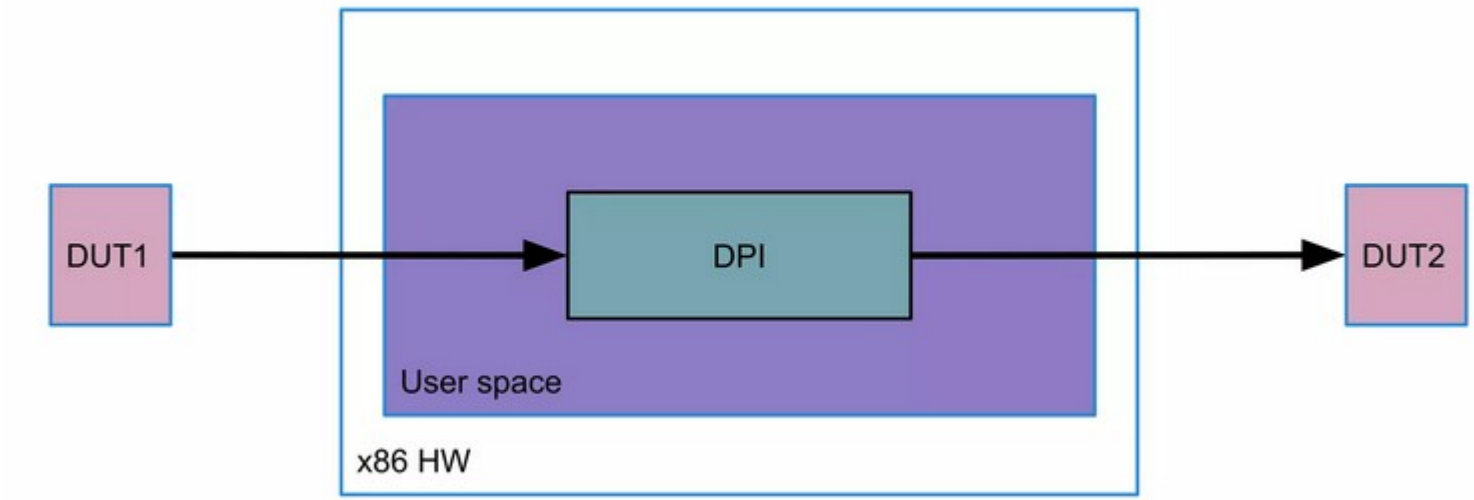
Application 1 - Générateur de trafic



Application 2 - Routeur



Application 3 - Middlebox (boîtier intermédiaire / fonction réseau intermédiaire)



Travaux Pratiques (TP / LABS)

Repo git des tp

- $ git clone https://framagit.org/linuxmaine/formations
- $ cd dpdk/labs/
- $ ls -al

Outils

Outils

1. dpdk-devbind.py

Binder/débinder les cartes réseau avec VFIO / UIO / kernel NIC.

- Affiche les NIC compatibles
- Change leur driver
- Vérifie les ressources PCI

👉 Indispensable pour lancer DPDK.

2. dpdk-hugepages.py

Gestion des hugepages :

- allocation
- vérification
- nettoyage

Outils

3. dpdk-pmdinfo.py

Affiche des informations internes sur un pilote PMD DPDK :

Capability NIC

Offloads disponibles

Configurations PCI

4. dpdk-pmdtest

Test de base des drivers PMD (Poll-Mode Drivers).

Outils

5. dpdk-test

Binaire officiel de test unitaire/mémoire de DPDK :

- mbuf
- EAL
- mempool
- ring
- timers…

C’est le “dpdk test suite”.

6. dpdk-proc-info

Affiche l’état interne d’une application DPDK en cours :

- mbuf,
- mempool,
- ports,
- queues,
- statistiques…

⚙️ Applications exemples (sample apps)

7. testpmd

Un des outils les plus importants.

Permet de :

- configurer une NIC DPDK
- tester le trafic
- activer/desactiver RSS, offloads
- faire du forwarding simple

👉 Très utilisé pour vérifier les performances d'une carte.

8. l2fwd

Forwarder L2 simple (Layer-2).

Outil pédagogique + benchmark basique.

Outils

9. l3fwd / l3fwd-power

Fonctionne en L3 (routage IP).

- utile pour mesurer performances de forwarding IP
- test de latence
- test de débit multi-cœur

10. l2fwd-crypto

Forwarder L2 + cryptographie (AES-NI / QAT / etc.)

11. ipsec-secgw

Implement un “IPsec Security Gateway” (IPsec SA, ESP, etc.)

Outils

12. vhost / virtio example apps

Pour tester :

- DPDK vhost-user
- virtio-net
- QEMU VM high-speed networking

📡 **Outils externes pour debugging & tests réseau**

13. pktgen-dpdk

Générateur de trafic 100% basé DPDK.

- envoi/recepetion à très haut débit
- tests NIC
- traffic patterns custom
- GUI CLI + Lua

👉 L’outil de benchmark #1 avec DPDK.

Outils

14. MoonGen

Générateur de trafic haute performance basé sur DPDK et LuaJIT.

15. TRex (Cisco)

Générateur de trafic stateful / stateless 40-100Gbps.

Compatible DPDK.

16. dpdk-vhost-switch

Tester communication vhost / virtio entre VM et DPDK.

17. dpdk-dumpcap (Wireshark)

Capture de paquets en utilisant DPDK au lieu du kernel.

faible overhead

utile pour debug sans perturber les performances

Outils

🔍 Monitoring / Profiling / Performance

18. perf (perf-tools)

Profiling CPU pour analyser :

- cycles
- instructions
- cache misses
- contention

19. ftrace / trace-cmd / kernelshark

Tracing kernel + analyse fine de latence

Outils

20. flamegraph (Brendan Gregg)

Profil CPU visuel :

- stack DPDK
- hot paths
- Bottlenecks

21. PTP / Chrony

Si ton app utilise le timestamping NIC / PTP :

synchronisation haute précision

22. ethtool (côté kernel)

Toujours utile pour :

- vérifier offloads
- voir RSS
- inspecter queues kernel NIC (si non bindées DPDK)

Outils

🧰 Développement & Intégration

23. meson + ninja

Système de build officiel de DPDK.

24. GDB avec scripts DPDK

DPDK fournit :

- macros gdb → lecture structures mempool, mbuf, rings…

25. ABI dumper / ABI compliance checker

Pour vérifier compatibilité API/ABI si tu modifies DPDK.

Outils

🕸 Outils pour virtualisation et containers

26. QEMU / virtio-net / vhost-user

Pour tester :

- fast-path VM ↔ DPDK
- SR-IOV
- VFs
- passthrough PCIe

27. SPDK (Storage DPDK)

Pour le stockage NVMe en user-space lié au modèle DPDK.

Outils

28. OVS-DPDK (Open vSwitch)

Switch virtuel accéléré par DPDK :

- utilisé en data centers
- nécessaire pour NFV / VNF

🧪 Test et validation

29. DPDK Unit Test Suite (DTS)

Outil officiel de test automatisé :

- performance
- fonctionnel
- regression

30. Scapy

Pour générer des paquets custom lors de tests.

Moins rapide, mais indispensable pour tests fonctionnels.

Glossaire

Glossaire

- **API RTE :**

L'ensemble des fonctions publiques fournies par DPDK (préfixe rte_), utilisées pour gérer les buffers, les queues, les ports, la mémoire, les cœurs, etc.

- **Affinity / CPU Affinity :**

Technique consistant à épingler un thread DPDK (lcore) sur un CPU spécifique pour éviter les migrations de tâches et améliorer la latence.

- **Burst :**

Mode d’envoi/réception de paquets par lots (batch) via les fonctions rte_eth_rx_burst() et rte_eth_tx_burst() pour maximiser les performances.

- **Buffer (mbuf) :**

Structure (rte_mbuf) représentant un paquet. Contient :

  - **pointeur vers la zone mémoire du paquet,**
  - **métadonnées (taille, port, RSS, VLAN, etc.),**
  - **références pour le multi-segment.**

Glossaire

- **Core / lcore :**

Cœur logique utilisé par DPDK via -l ou --lcores. Un lcore exécute typiquement une boucle de traitement.

- **CPU Poll Mode :**

Mode où les threads DPDK scrutent activement les queues réseau au lieu d'attendre des interruptions (évite overhead kernel).

- **DPDK EAL (Environment Abstraction Layer) :**

Couche d'abstraction initialisant :

  - **hugepages,**
  - **mapping mémoire,**
  - **PCI probing,**
  - **gestion des cœurs (lcores),**
  - **timers.**

Glossaire

- **Flow API :**

API pour créer des règles de traitement avancé (filtres, redirections, RSS, drop, mirroring) directement sur la NIC.

- **Hugepages :**

Pages mémoire de grande taille (2 Mo ou 1 Go) utilisées pour réduire la fragmentation et améliorer les performances DMA.

- **LPM (Longest Prefix Match) :**

Mécanisme d'accélération pour le routage IP basé sur le préfixe le plus long, implémenté via des tables optimisées (rte_lpm).

- **Lcore ID :**

Identifiant logique d'un thread DPDK, différent du CPU physique.

- **Mbuf :**

Voir Buffer (mbuf). Élément central de la gestion mémoire dans DPDK.

- **Mempool :**

Pool de buffers préalloués (souvent d'mbufs) pour éviter les allocations dynamiques coûteuses.

Glossaire

- **NIC Poll Mode Driver (PMD) :**

Pilote DPDK utilisé pour une carte réseau (ex.: ixgbe, i40e, mlx5). Fonctionne en mode polling, en espace utilisateur.

- **NUMA (Non-Uniform Memory Access) :**

Topologie mémoire où chaque CPU possède sa mémoire locale. DPDK optimise l'allocation en tenant compte des nœuds NUMA.

- **Packet Steering / RSS (Receive-Side Scaling) :**

Mécanisme matériel distribuant les paquets sur différents RX queues via un hash (flots séparés).

- **PCIe Probing :**

Découverte automatique des cartes réseau compatibles PMD lors du lancement de DPDK.

- **Port ID :**

Identifiant d'un port DPDK, correspond généralement à une interface PCI (NIC).

- **Queue RX/TX :**

Queues matérielles utilisées pour recevoir (RX) ou envoyer (TX) des paquets. Plusieurs queues peuvent être assignées à différents lcores.

Glossaire

- **Ring :**

Structure FIFO lockless permettant l'échange multi-producteurs/multi-consommateurs entre threads DPDK.

- **RTE (Runtime Environment) :**

Ensemble des modules de base de DPDK (EAL, Mempools, Rings, Mbufs, Timers, Logs).

- **Secondary Process :**

Mécanisme permettant à plusieurs processus DPDK de partager les mêmes hugepages et mempools.

- **SP/SC – MP/MC :**

Types de rings :

  - SP/SC : Single Producer / Single Consumer
  - MP/MC : Multi Producer / Multi Consumer

- **TSC (Time Stamp Counter) :**

Compteur CPU haute résolution utilisé par les timers DPDK (rte_get_tsc_cycles()).

Glossaire

- **Telemetry :**

Interface permettant d'obtenir des statistiques internes via une API REST ou socket.

- **User-space Networking :**

Principe clé de DPDK : tout le datapath est exécuté en espace utilisateur, sans passer par le kernel.

- **vdev (Virtual Device) :**

Périphérique virtuel géré par DPDK (ex. net_tap, net_pcap, crypto_aesni_mb) permettant tests ou intégrations.

- **Virtio / vhost :**

Pilotes DPDK utilisés dans des environnements virtualisés (KVM, QEMU) pour optimiser les performances I/O réseau.

- **Worker lcore :**

Thread principal effectuant le traitement des paquets dans un pipeline DPDK.

Aide complementaire (liens)

Principales documentations officielles

- Site officiel DPDK — page d’accueil, documentation, lien de téléchargement, etc. dpdk.org+1
- Getting Started Guide for Linux — guide officiel pour installer et configurer DPDK sous Linux, compiler, lier NIC, etc. doc.dpdk.org+2dpdk.readthedocs.io+2
- Programmer’s Guide — documentation pour développeurs : architecture DPDK, EAL, librairies, exemples, guidelines. doc.dpdk.org+2doc.dpdk.org+2
- Quick Start Guide — un guide minimaliste pour démarrer DPDK rapidement (build + test simple). core.dpdk.org+1
- Guides par version / archives — si tu utilises une version spécifique de DPDK, les docs archivées sont accessibles via le site officiel. doc.dpdk.org+1
- Guide d’installation & configuration (ex. PDF « Getting Started Guide for Linux » d’une ancienne version) — parfois utile pour l’historique ou pour des systèmes anciens. Intel+1

Articles, tutoriels introductifs, blogs & présentations

- “Introduction to DPDK: Architecture and Principles” — un article clair décrivant les principes, l’architecture interne de DPDK, ses bibliothèques, etc. Medium
- Tutoriels “hands-on” sur GitHub — un projet libre contenant plusieurs petits exemples : lecture de paquets, envoi, traitement, statistiques, offloading, RSS, queues, ring buffer, etc. Très utile pour débuter. GitHub
- Présentation vidéo “Introduction to DPDK” — une vue d’ensemble rapide pour comprendre ce qu’est DPDK, pourquoi l’utiliser et comment ça fonctionne. https://www.youtube.com/watch?v=1DWxo2gF-RQ&embeds_referring_euri=https%3A%2F%2Fchatgpt.com%2F&source_ve_path=OTY3MTQ

Guides “pratiques / cas d’usage / example apps”

- Exemples dans DPDK — le paquet DPDK contient des applications exemples (“sample applications”) qui montrent comment faire forwarding L2, routage, encapsulation, etc. Le Programmer’s Guide ou les docs “Sample Applications User Guide” donnent les détails. dpdk.readthedocs.io+2buildmedia.readthedocs.org+2
- Cas d’usage “performance / hardware NIC” — le guide explique comment binder une NIC, utiliser hugepages, configurer VFIO/UIO, etc. Important pour obtenir des bonnes performances. doc.dpdk.org+2buildmedia.readthedocs.org+2
- Interopérabilité / compatibilité matériel — la documentation officielle liste les NIC compatibles avec DPDK, ce qui aide à vérifier avant achat ou déploiement. core.dpdk.org+1

Monter en competence sur DPDK

Exemple de plan d’apprentissage avec prérequis GNU/Linux, C/C++, ... :

- Lire le Getting Started Guide for Linux → pour installer DPDK, comprendre la configuration (hugepages, UIO / VFIO, binding NIC, etc).
- Lire le Programmer’s Guide → pour comprendre l’architecture, EAL, mémoire, gestion des cores, concept de “poll-mode driver”, etc.
- Jouer avec les exemples fournis (forwarding, l2fwd, l3fwd, etc.) pour voir comment ça fonctionne “en vrai”.
- Explorer le dépôt dpdk-tutorials sur GitHub — utile pour apprendre “par le code”, pas à pas.
- Lire des articles/présentations externes (comme “Introduction to DPDK: Architecture and Principles”) pour avoir un aperçu conceptuel global.
- Si besoin, consulter les docs “hardware / NIC compatibility” pour s’assurer que ta carte est supportée.
- Suivre des ressources communautaires / forums / mailing-list si tu rencontres des problèmes ou pour voir des cas d’usage avancés.

Formations & ressources communautaires

- Page “Training” officielle de DPDK — recense des ressources d’apprentissage, présentations, parfois des sessions/instructions offertes. core.dpdk.org+1
- Communauté & forums / mailing-list / GitHub — bien que ce ne soit pas un “lien unique”, contribuer ou lire les issues / pull requests permet de suivre les problèmes réels, les retours d’expérience, les bugs, les optimisations. (liens disponibles depuis le site officiel DPDK) dpdk.org+1

Liens

- Device Drivers in User Space : Version web du chapitre correspondant dans le livre Essential Linux Device Drivers (Drivers in User Space) — aperçu payant chez O’Reilly O'Reilly Media
- The Userspace I/O HOWTO : Documentation officielle du noyau : “The Userspace I/O HOWTO” — https://docs.kernel.org/driver-api/uio-howto.html Kernel Documentation+1
- Userspace I/O drivers in a realtime context : Article PDF “Userspace I/O Drivers in Realtime: Performance & Implementation” (Hans J. Koch) — disponible publiquement. studylib.net+1
- The anatomy of a PCI/PCI Express kernel driver : PDF de cours/présentation “The anatomy of a PCI/PCI Express kernel driver” par Eli Billauer — visible en ligne. haifux.org
- Linux Device Drivers, Third Edition : Version PDF libre du livre Linux Device Drivers, 3rd Edition (LDD3) — souvent hébergée sous forme “LDD3 pdf” en ligne. w3.ual.es+1
- LMbench - Tools for performance analysis : Page officielle du projet lmbench — site SourceForge “How do I get LMbench ?” lmbench.sourceforge.net+1
- From Intel® Data Plane Development Kit to Wind River Network Acceleration Platform : Présentation PDF de Wind River / DPDK liant DPDK et la “Network Acceleration Platform” — “Wind River Intel DPDK support” Intel+1

Liens

- https://en.wikipedia.org/wiki/Data_Plane_Development_Kit#Projects
- https://www.intel.fr/content/www/fr/fr/content-details/777309/getting-started-with-dpdk-for-linux.html
- https://learn.microsoft.com/fr-fr/azure/virtual-network/setup-dpdk?tabs=redhat
- https://doc.dpdk.org/guides/linux_gsg/index.html
- https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/getting-started-with-dpdk_configuring-and-managing-networking
- https://pcapplusplus.github.io/v1912/docs/tutorials/dpdk
- https://embedx.medium.com/introduction-to-dpdk-architecture-and-principles-40db9a61a6f5