UDP hautes performances

H. Grasland, S. A. Cherrati, N. Dosme, J. Odier

2026-01-12

Contexte : Traitements *online*

- Détecteurs toujours plus gros, plus précis
  - Débits plus gros, davantage de pairs réseau
- Stockage qui ne suit pas → Il faut réduire/choisir les données
  - Calculs supplémentaires de réduction/filtrage
- On fait face avec du matériel + logiciel spécialisés

UDP

- **L'électronique simplifie sa vie avec une astuce simple !**
- Moins coûteux **pour eux** que TCP : Pas de suivi paquets donc…
  - Pas de garantie d'ordre ni de bonne réception
  - Pas d'attente des destinataires à la traîne
- Côté logiciel d'acquisition (DAQ), c'est un enfer
  - Temps réel mou sur un OS pas conçu pour
  - Trop de charge CPU ? On perd des données.

UDP vs CPU

- À ≥10 Gbps, **CPU / bus RAM souvent saturé avant réseau**
- Beaucoup de CPU dépensé pour traiter les paquets UDP
  - *Polling* et interruptions de carte réseau (NIC)
  - Décodage en-têtes
  - Changements de contextes / appels systèmes
  - Copies mémoire
- Peut-on optimiser cette utilisation CPU ?

R&D UDP à IJCLab

- Trois pistes explorées à IJCLab
  - **udipe :** Mieux utiliser la pile réseau Linux
  - **DPDK :** Remplacer la pile réseau
  - **GPUDirect :** Remplacer le CPU
- Ces pistes sont complémentaires
  - DPDK et GPUDirect difficiles → udipe pour “petits” besoins
  - GPUDirect seulement utile pour les traitements GPU

udipe : Mieux utiliser Linux

Ce qu’on fait déjà

- Gros savoir-faire existant côté **config système**…
  - Taille tampons NIC, noyau, *socket* (requêtes + données)
  - Durée de *polling* avant passage en mode interruption
  - Nombre de FIFO matérielles utilisées
  - Éviter de partager l’IRQ associée à la NIC
  - *Pinning* NUMA matériel + logiciel
  - Utilisation de *jumbo frames*

Mais côté code…

- Utilisation du noyau Linux ~stable depuis des décennies
  - socket()/connect()/bind()/recv()
  - 1 thread/process par pair réseau
- Cette approche a des **coûts incompressibles :**
  - ≥1 appel système par paquet reçu
  - ≥2 copies des paquets en RAM (HW/kernel + kernel/appli)
  - *Oversubscription* CPU si Npairs > Ncpu
- Rien de neuf depuis les *sockets* 4.2BSD (1983) ?

Linux face à Internet

- 2002 : **epoll** = gestion O(1) de connexions concurrentes
- 2007 : **eventfd** = Ouvre epoll aux événements arbitraires
- 2010 : **recvmmsg** = N paquets par appel système
- 2016 : **SO_INCOMING_CPU** = Affinité CPU pour les *sockets*
- 2016 : **XDP** = Filtrage/traitement paquets avec eBPF
- 2017 : **MSG_ZEROCOPY** = Éviter la copie noyau-appli
- 2019 : **UDP_GRO** = N paquets au sein de la pile réseau
- 2019 : **io_uring** = 1 appel système pour N tâches diverses
- …liste non exhaustive…

Ma proposition : udipe

- Projet ultra-WIP de **bibliothèque C11 pour l'UDP optimisé**
  - C11 pour être accessible depuis C/++, Ada, Go, Rust…
  - CPU optimisé comme du HPC (*thread-per-core*, cache…)
  - Synchro optimisée comme du temps réel mou (*lock-free*, …)
  - Évalue et intègre les évolutions de l'API Linux
- **API flexible**, du plus facile au plus puissant selon besoin
  - Objectif : Approprié par des manips de taille variable

ATTENTION : Projet ultra-WIP

Tout ce qui suit va évoluer selon l’expérience d’implémentation

Contraintes d'implémentation *workers*

- UDP ~ temps réel mou → Il faut être **prévisible et robuste**
  - *Pinning* CPU/NUMA + ordonnancement SCHED_RT si dispo
  - Éviter *oversubscription* CPU → N connexions / coeur
  - Appels système en phase DAQ minimisés + non bloquants
  - mlock() si dispo pour éviter tout *swapping*
  - Privilégier état privé, idéalement ≤ cache L2
  - Synchro rare, rapide, non bloquante, indépendante du client
- **Abstraire le client** de cette complexité si possible !

API flexible

- Dans tous les cas, on sera **orienté connexion**
  - Simplifie/optimise l'implème, pas un problème en *online*
  - Connexion ~ socket + FIFO de requêtes send/recv
- Choix entre trois compromis ergonomie/perf :
  - **Synchrone :** API send()/recv() bloquante traditionnelle
  - **Asynchrone :** Retourne une future au lieu de bloquer
  - ***Streaming* :** *Callback* utilisateur pour chaque paquet

Mode facile : synchrone/asynchrone

- **API asynchrone** basée sur des **futures**
  - `udipe_start_recv()`, etc. retourne `udipe_future_t` avant la fin
  - `udipe_finish(future)` attend + retourne le résultat/erreur
- Plus flexible et optimisable côté client que le code synchrone
  - Recouvrement I/O vs traitement sans *threads* client
  - `udipe_start_join()` = Attente optimisée de N futures
  - `udipe_start_unordered()` = Résultats par ordre d'arrivée
- **API synchrone** = Raccourci pour start() suivi de finish()

Garanties d’ordre

- Quel lien **ordre requêtes / ordre traitements ?**
  - Ordre identique ? → Parallélisme/concurrence impossible !
  - Aucune garantie ? → Peu intuitif, risque de *bugs* élevé !
- Compromis privilégié : **Ordonné par connexion**
  - Requêtes séquentielles au sein d’une connexion
  - Concurrence entre N connexions
  - Satisfaisant si connexions assez nombreuses/homogènes
- Extensible → Ordre à N connexions, connexion non ordonnée…

Conception des futures udipe

- Conçues pour **maximiser le potentiel de perf**
  - *Lock-free* pour le *worker,* coût signal O(1) vs Nclients
  - Recyclage *thread-local* → Peu d'allocations, de synchro
  - Stockage *inline* qui tient dans une ligne de cache
- Compromis : **Moins flexibles** que dans d'autres *frameworks*
  - Stockage *inline* + recyclage + type C → Type résultat borné
  - Isolé de la complexité client → Pas d'opérations map() etc
  - Résultat lu par 1 seul *thread*, idéalement commanditaire

Limites du mode asynchrone

- **Requête send/recv = synchro *threads* client/worker**
  - Coût CPU de synchro qui limite la performance max
  - Localité cache/NUMA des données réduite
- **Tampons fournis par l'utilisateur**
  - *Working set* non borné → Localité de cache non garantie
  - Pas de mlock() → Absence de *swapping* non garantie
  - Support utilisateur requis pour recyclage tampons, GRO…
  - Impossible d'utiliser `io_uring_register_buffers()` & cie

Aller plus loin : mode *streaming*

- Souvent, on ne fait **pas des send()/recv() au hasard**
  - DAQ typique = recv() en boucle
    - send() en boucle utile pour test/bench DAQ simple
    - recv()/send() alterné utile pour test/bench plus avancé
  - Point commun: **en fin d'opération, on connaît la suite**
- Peut-on exploiter cette logique prévisible ?
  - Oui si l'API connexion devient un ***callback* utilisateur**
  - Cas typique : **Traite un paquet + émet la requête suivante**
  - Cas spéciaux à gérer : démarrage (pas de paquet), arrêt

Avantages du *streaming*

- **Tampons de données gérés côté *worker***
  - Réutilisation sans effort utilisateur, mlock() garanti
  - Taille + nombre sous contrôle → Localité de cache
  - Taille sous contrôle → GRO/recvmmsg peut être abstrait
  - Accès aux APIs avancées type “*registered buffers*”
- Génération/traitement et send()/recv() sur le **même CPU**
  - Localité cache/NUMA/PCIe optimales
  - Ni synchro ni changement de contexte OS

Prix à payer

- **Le *callback* s'injecte dans un *worker*** qui gère N connexions
  - Doit être écrit prudemment comme tout code *worker*
  - Toute erreur ralentit N-1 autres connexions
  - Modèle nettement moins abstrait/ergonomique !
- Quand est-ce que le *streaming* vaut le coup ?
  - Si même bien utilisé (GRO etc), l'asynchrone reste cher
  - Sur des traitements simples type filtrage/routage

Statut et perspective

- **Projet encore très jeune**, mais qui évolue bien
  - Infra build/logs/test ~finale
  - Infra benchmark en cours de dev
  - Futures et *workers* : conception bien avancée, quelques briques implémentées, beaucoup reste à faire
- Objectif : Premier *backend* simple + tests + benchs puis...
  - Ecrire et comparer des *backends* plus sophistiqués
  - En parallèle, valider sur cas réel : IPBus → DAQ KAIROS

DPDK : Remplacer Linux

Contexte : Mouche vs bazooka

- La pile réseau Linux est complexe car flexible
  - Gestion des infos de contrôle (ARP, ICMP, DHCP…)
  - Filtrage et routage virtuel avec priorisation
  - Aggrégation de liens
  - Abstraction des différences matérielles
- Le DAQ a rarement besoin de cette flexibilité
  - Peut-on **spécialiser la pile pour l'accélérer ?**

DPDK

- Utilise l'infrastructure **VFIO** des hyperviseurs de VMs
  - Carte réseau (NIC) détachée de la pile réseau standard
  - Application en contact direct avec pilote spécialisé
- Prix à payer : Il faut **tout faire soi-même**
  - Plus de gestion "gratuite" ARP/ICMP/DHCP
  - Code assez spécifique à la NIC utilisée

Configuration système

- Prérequis :
  - **NIC** supportant les fonctions voulues*
  - **IOMMU** actif (base du VFIO)
- Optimisations :
  - **Hugepages 1Go** pour réduire les coûts TLB
  - **Pinning NUMA** appli DPDK = carte réseau

* Le niveau de support dépend du matériel mais aussi des *drivers* fournis par DPDK.

État d’avancement

- **Environnement OK** (IOMMU, NUMA, huge pages…)
- **Receive Side Scaling OK**
  - La NIC sépare les paquets en N files
  - Traitement par N *threads* applicatifs
- **GPUDirect en cours** : Partie suivante !

Premiers résultats

- **25 Gbps sans perte** (jumbo frames 8 KB)
- **Conso CPU < Linux naïf** en mode interruption
- Traitement parallélisé sur **1 CPU par file NIC**

GPU direct : Remplacer le CPU

Contexte : GPU en online

- Utilisation du GPU pour **gagner du temps CPU** précieux
  - L'efficacité GPU n'est pas très importante !
- Problème : Beaucoup de **travail CPU/RAM/PCIe** résiduel
  - Copie NIC → RAM via PCIe quand un paquet arrive
  - Coût CPU associé au traitement noyau/appli
  - Copie RAM → VRAM via PCIe
  - …et enfin on lance le traitement GPU
- Comment réduire cette pression ?

GPUDirect = DMA NIC-GPU

- **Pair-à-pair NIC-GPU** possible sous conditions
  - Cartes connectées à la même racine PCIe
  - Compatibilité matériel NIC/GPU + support logiciel DPDK
  - En pratique, Nvidia Connect-X ≥ 5 recommandé
- La NIC peut alors **écrire les paquets en VRAM**
  - Problème : Notification GPU et lancement traitement ?

Synchro NIC-GPU : Deux options

1. **La NIC notifie le CPU** qui lance le traitement GPU
   - Léger coût CPU, mais surtout latence ajoutée

2. **Attente active sur GPU** (*flag* en VRAM)
   - Coûteux en cycles GPU, mais latence minimale

État d’avancement

- Communication GPU via **gpu_dev de DPDK**
  - Transfert avec intermédiaire NIC-CPU-GPU OK
  - Transfert direct NIC-GPU en attente de réception NIC NVidia

Conclusion

- L'UDP simplifie la vie de l'électronique…
  - …mais il complique la vie du logiciel DAQ en face
  - Sur réseaux ≥ 10 Gbps, l'utilisation CPU devient un problème
- Trois pistes explorées à IJCLab pour réduire cette utilisation
  - Mieux utiliser la pile réseau Linux (udipe)
  - Contourner la pile réseau (DPDK)
  - Contourner le CPU et la RAM (GPUDirect)

Merci pour votre attention !

Temps réel mou ?

- Pas de contrainte de temps en apparence, sauf qu'on a…
  - Un débit entrant ~constant N paquets/seconde
  - Un tampon d'entrée pouvant garder M paquets en attente
- Conséquence : M/N secondes pour traiter chaque paquet
  - C'est une contrainte de temps réel
- Plus lent que ça, on perd des données → OK si rare
  - C'est le régime de temps réel mou

udipe : Connexions désordonnées ?

- Un gros chantier d'implème → Pas avant udipe v2
  - Commandes internes *broadcast* à N *workers* (ex: connect)
    - Nécessite canal dédié pour être efficace
      → Pas naturellement compatible avec sémantique FIFO
    - Gestion d'erreurs complexe (annulations !)
  - Besoin d'équilibrage de charge plus fin (chaque send/recv)
  - En *streaming*, les *callbacks* doivent être *thread-safe*
  - Pose la question de la localité CPU/NIC (*pinning* des IRQs)
- N'est correct et donc utile que pour certains protocoles !