

Soutenance de stage - On Demand Buffer

Julien Brelot

03 mars — 28 juillet 2025



- ① Présentation du LIG
- ② Contexte
- ③ Intégration dans une application existante
- ④ Alignement des données
- ⑤ Protocole d'encapsulation
- ⑥ Messages ODB et flux de données

Le Laboratoire d'Informatique de Grenoble (LIG) est une unité mixte de recherche (UMR 5217) rattachée au CNRS, à l'Université Grenoble Alpes (UGA), à Grenoble INP et à l'Inria Grenoble Rhône-Alpes. Créé en 2007, il regroupe environ 500 personnes réparties sur plusieurs sites grenoblois (Saint-Martin-d'Hères, Minatec, Montbonnot).

Axes de recherche principaux :

- Génie logiciel et systèmes d'information
- Méthodes formelles, langages et modèles
- Systèmes interactifs et cognitifs
- Calcul parallèle, distribué et réseaux
- Traitement des données massives et gestion des connaissances

Créée en octobre 2024, KrakOS (Optimiser les couches système des centres de données) est spécialisée dans les systèmes d'exploitation, environnements virtualisés et middlewares cloud. L'équipe, dirigée par Alain Tchana, regroupe enseignants-chercheurs, doctorants et ingénieurs de recherche.

Objectifs de KrakOS :

- ① Améliorer les performances des systèmes (temps d'exécution, débit, latence)
- ② Renforcer tolérance aux pannes et haute disponibilité
- ③ Faciliter le développement, tests et déploiement rapide
- ④ Fournir des API expressives et flexibles aux développeurs
- ⑤ Réduire la consommation énergétique des infrastructures

KrakOS bénéficie d'un environnement scientifique riche, avec des collaborations nationales et internationales (ex. IRISA, IRIT).

Contexte Général (1/3)

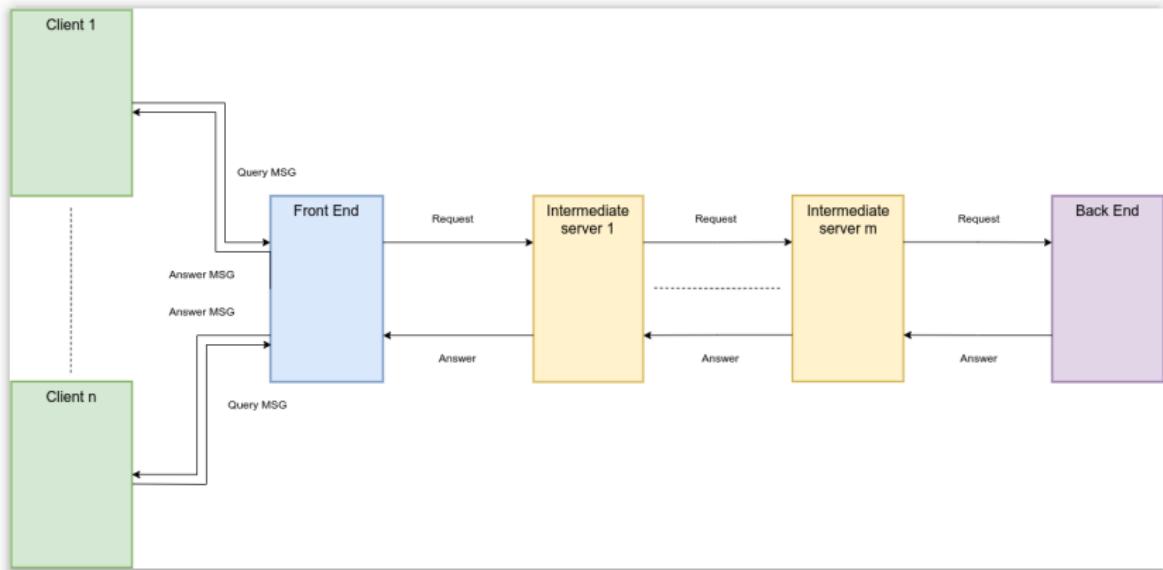
- Les **datacenters** et le **cloud computing** sont au cœur des infrastructures modernes, hébergeant services en ligne critiques : e-commerce, banques, messagerie, réseaux sociaux.
- Les applications suivent souvent des architectures **multi-tiers** :
 - *Front-end* : présentation / point d'entrée,
 - *Middle-tier* : logique métier,
 - *Back-end* : gestion des données.
- Cette séparation assure modularité, maintenabilité et scalabilité.

- **Problèmes majeurs des architectures multi-tiers :**
 - Multiplication des **copies de données** : buffers applicatifs, mémoire noyau, transferts réseau → utilisation inefficace CPU et bande passante.
 - **Backpressure** : accumulation de buffers lorsque les couches amont produisent plus vite que les couches aval peuvent traiter → augmentation de latence.
 - Opérations **I/O** coûteuses : appels système, basculement mode utilisateur/noyau, copies multiples → saturent CPU et bus mémoire.

Contexte Général (3/3)

- Objectif du projet : optimiser les performances des architectures multi-tiers dans le cloud.
- Solution proposée : **On-Demand Buffer (ODB)**
 - Déetecte de façon transparente l'usage réel des données par les intermédiaires.
 - Transfère uniquement les données nécessaires, réduisant les copies inutiles.
 - Mitige la contre-pression et améliore la performance I/O.
- Bénéfices attendus : réduction CPU, réduction latence, meilleur débit et meilleure utilisation des ressources pour les applications modernes.

Contexte : Illustration



- **Objectif d'ODB** : fonctionner avec une application web existante sans modifier le code source.
- Contraintes clés :
 - Ne pas perturber les communications existantes (HTTP, FTP, SMTP...).
 - Envoyer et recevoir des données, sans bloquer l'application.
 - Séparer les données ODB des buffers de l'application.
 - Gérer les erreurs ODB sans impacter l'application.
- Nécessité de **sockets connectées et séquentielles** (TCP) pour garantir la cohérence des échanges.

Alignement des données

- ODB utilise `mprotect()` pour protéger l'accès aux données.
- **Contraintes de `mprotect` :**
 - Protection à la granularité d'une page.
 - Mémoire alignée sur les frontières de pages.
- Buffers non-alignés :
 - **Head** : début non aligné (i page)
 - **Body** : zone alignée et multiple d'une page
 - **Tail** : fin non alignée (i page)
- Données Head/Tail doivent être transférées "réellement", Body peut rester virtualisé.

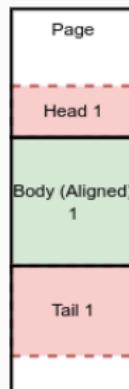
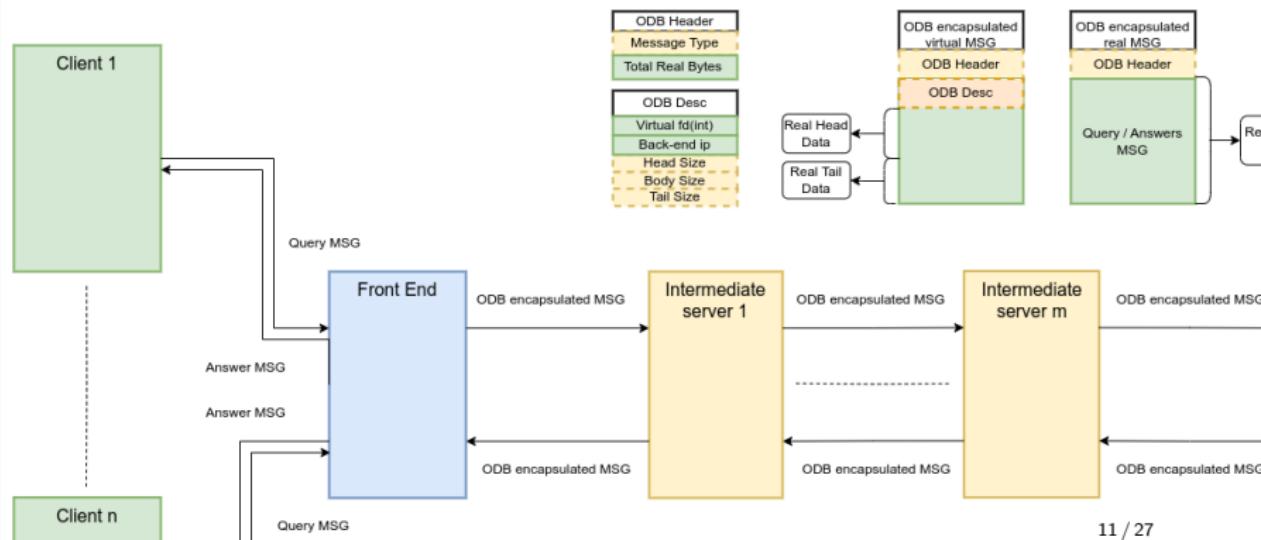


Figure: Zones mémoire Head / Body / Tail

Protocole d'encapsulation

- Besoin : garantir la cohérence et la compatibilité avec POSIX malgré les non-alignements.
- Solution : **protocole d'encapsulation ODB**
 - En-tête ODB_Header : type de données (réelles ou virtuelles)
 - Pour les données virtuelles : ODB_Desc décrit Body/Head/Tail et back-end
 - Interception des appels système pour ajouter/extraitre ces informations au niveau applicatif
- Nécessité de sockets séquentielles pour traiter les en-têtes ODB avant les données applicatives.



Messages ODB et flux de données

- Messages ODB contiennent :
 - Identifiant virtuel de la page, IP/port back-end
 - Tailles des zones Head, Body, Tail
- Données virtualisées vs données réelles :
 - Head/Tail : transférées effectivement
 - Body : virtualisé, transfert à la demande
- Objectif : transparence totale pour l'application, cohérence des échanges garantie.

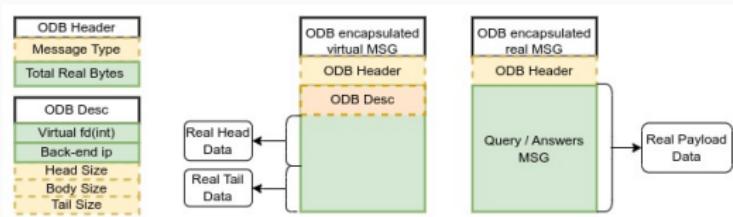


Figure: Détail des messages ODB

Utilisation du protocole ODB

- ODB nécessite que **les deux applications** utilisent le protocole pour garantir la cohérence des échanges.
- Deux approches pour gérer l'activation d'ODB :
 - ① Fichier de configuration : indique les ports où ODB est actif.
 - ② Mode *stand-alone* : détection automatique via un *magic number* + CRC-8.
- Objectif : permettre au front-end de communiquer correctement avec le back-end ODB tout en restant compatible avec des clients classiques.

Gestion des buffers non-alignés

- Head/Tail : parties non-alignées, transmises en données réelles.
- Body : zone alignée, peut rester virtualisée.
- Stratégies d'envoi :
 - **Statique** : taille fixe (*unaligned_size*), simple mais limite l'hétérogénéité.
 - **Dynamique** : taille variable, téléchargement à la demande depuis le BE.
 - **Anticipation** : hybride, BE envoie Head/Tail vers le IS suivant pour réduire les requêtes.
- Gestion des buffers de réception insuffisants : téléchargement segmenté Head/Body/Tail et maintien de l'état de réception.

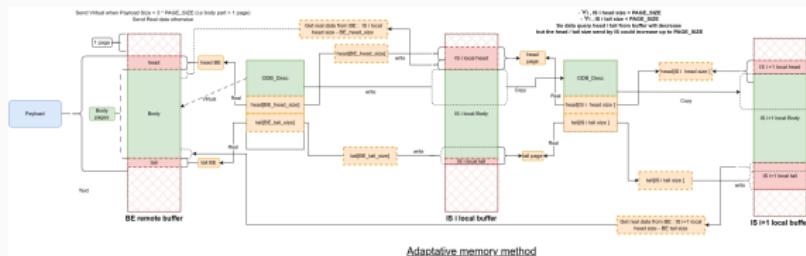


Figure: Stratégie d'anticipation pour buffers non-alignés

Développement de serveurs tests

- Objectif : tester et déboguer ODB.
- Composants :
 - FE : envoie des requêtes et reçoit des fichiers.
 - IS : transfère les données entre BE et FE.
 - BE : lit le fichier et envoie les données au FE via les IS.
- Paramétrable via flags pour tester différentes tailles de buffers.
- Application cible pour tests réels : **nginx**, serveur web/ reverse-proxy majoritairement utilisé.

Utilisation avec Nginx

- Nginx utilise plusieurs processus :
 - Master : gestion configuration et processus
 - Cache Loader / Manager : gestion mémoire cache
 - Worker : boucle événementielle pour traiter les connexions
- Boucle d'événements :
 - Surveillance des descripteurs (epoll/poll/kqueue)
 - Traitement séquentiel des événements
 - Sockets non-bloquantes nécessaires
- Thread pools pour gérer I/O lourdes sans bloquer les workers.

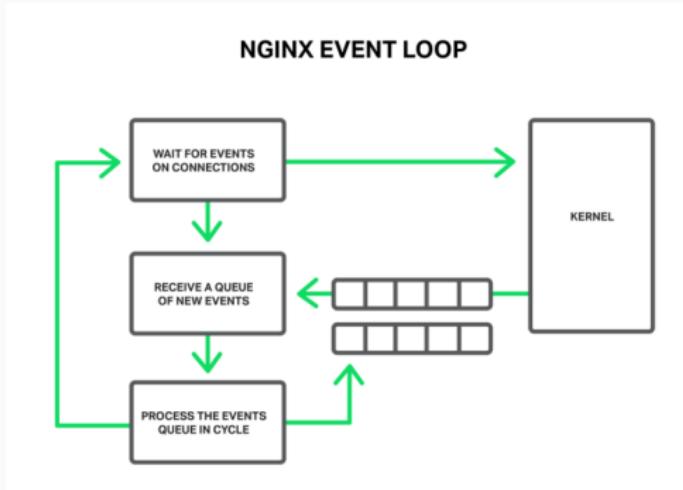


Figure: Boucle événementielle des workers Nginx

Problème avec epoll Edge Triggered (5/5)

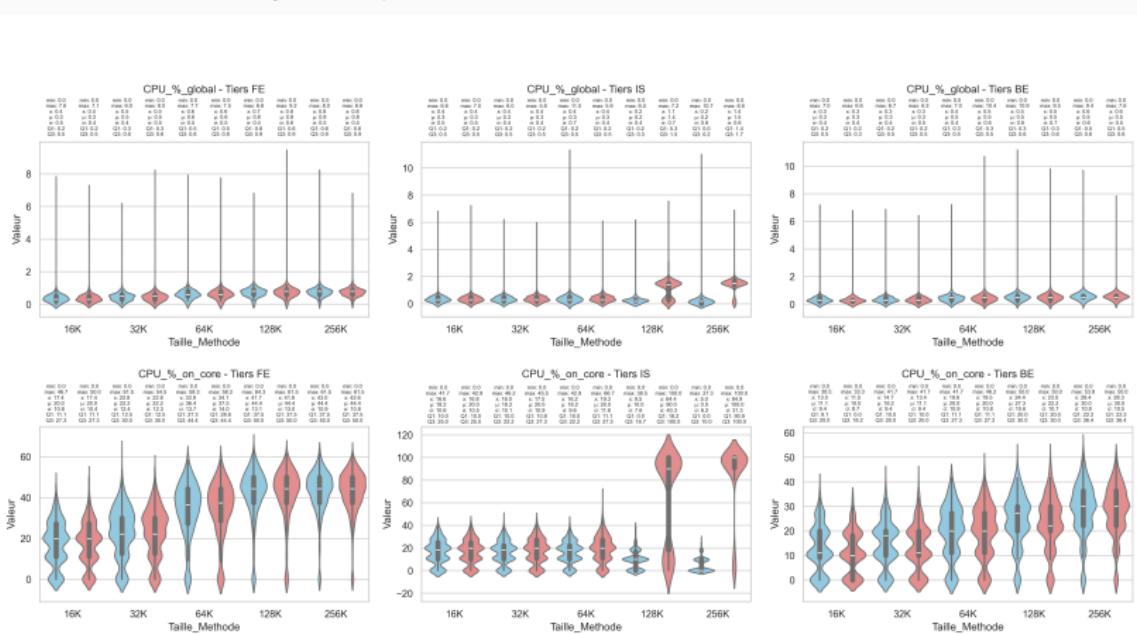
- Mode Edge Triggered : notification seulement lors du changement d'état de la socket.
- Problème : ODB virtualise les données, Nginx peut ne pas lire tout le buffer restant.
- Solutions envisagées :
 - ① Vérifier après chaque `recv` et forcer un événement EPOLLIN si nécessaire.
 - ② Intercepter `epoll_create` et `epoll_ctl` pour passer en mode Level Triggered.
- Conséquence : complexité accrue et risque de latence supplémentaire sur la boucle événementielle.
- Conclusion : intégration ODB dans Nginx nécessite de gérer finement la lecture/écriture sur sockets pour éviter les pertes de données.

Tests menés

- 3 serveurs Nginx sur 3 nœuds Grid5000, 1 worker par serveur, 1 cœur logique.
- Front-end et serveur intermédiaire : 2 proxies ; back-end : serveur HTTP.
- Proxy buffering désactivé.
- Charge : 50 clients/s jusqu'à 1000 clients, 30k requêtes GET.
- Payloads virtuelles : {16, 32, 64, 128, 256 Ko}.
- Métriques : CPU total, CPU sur cœur du worker, mémoire, latence et débit.
- Variables : alignement des buffers, stratégie ODB (*dynamic vs anticipated*).

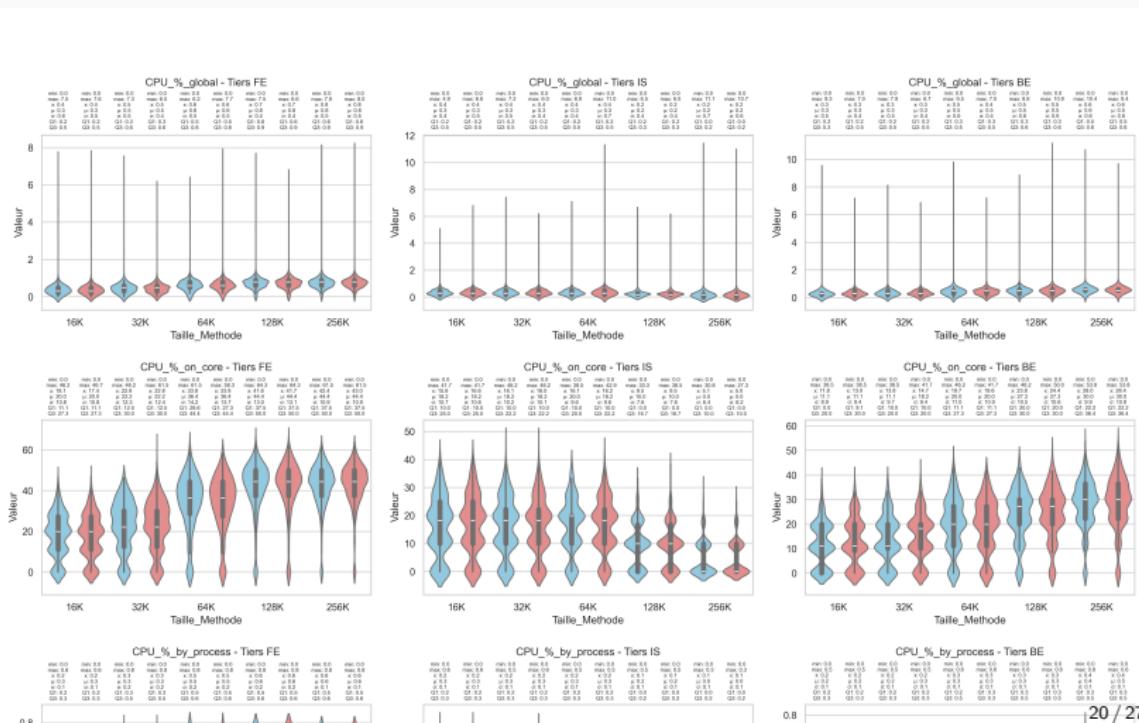
ODB : Stratégies d'envoi (non-aligné)

- *dynamic vs anticipated.*
- CPU similaire pour FE, IS, BE.
- IS : gain notable à partir de 128 Ko avec *dynamic*.
- Latence légèrement meilleure avec *anticipated*, surtout pour grosses payloads.
- Choix retenu : *dynamic* pour réduire CPU.



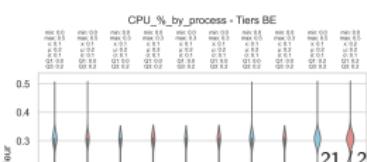
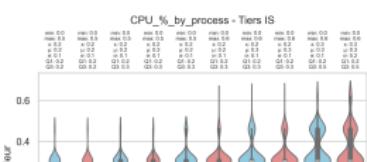
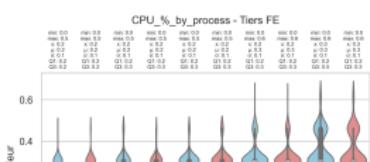
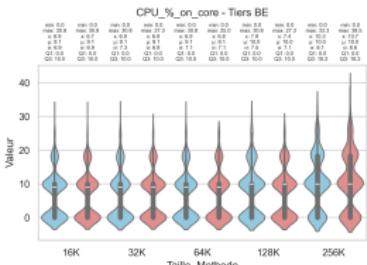
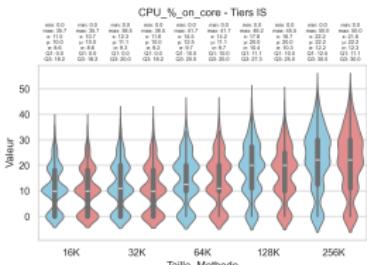
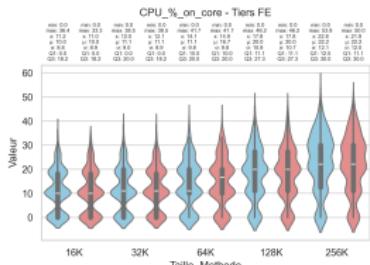
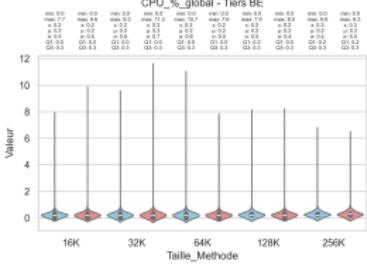
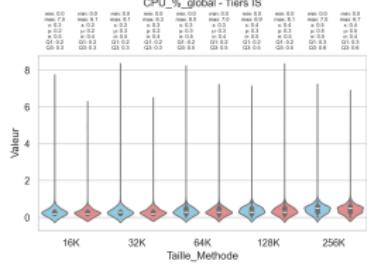
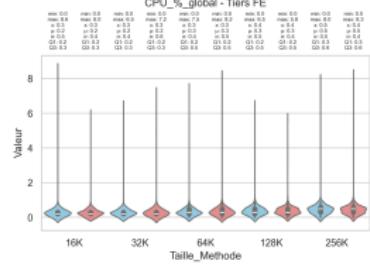
ODB : Alignement vs non-alignement (dynamic)

- Alignement peu impactant à faible payload.
- 128-256 Ko : réduction CPU sur FE et BE.
- Latence moyenne comparable mais légère amélioration pour grosses payloads.
- Explication : réduction des requêtes pour head/tail des buffers.



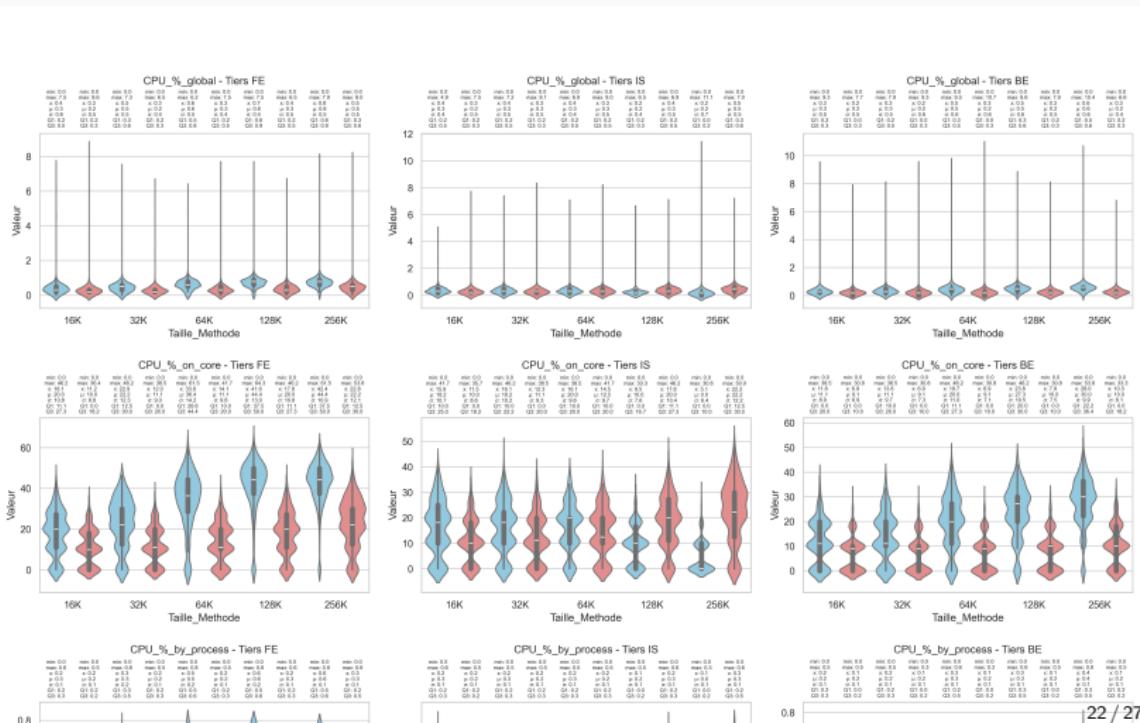
Vanilla : Alignement vs non-alignement

- Pas d'impact notable sur CPU ou QOS.
- Alignement user-space ignoré par le kernel lors des écritures sur socket.



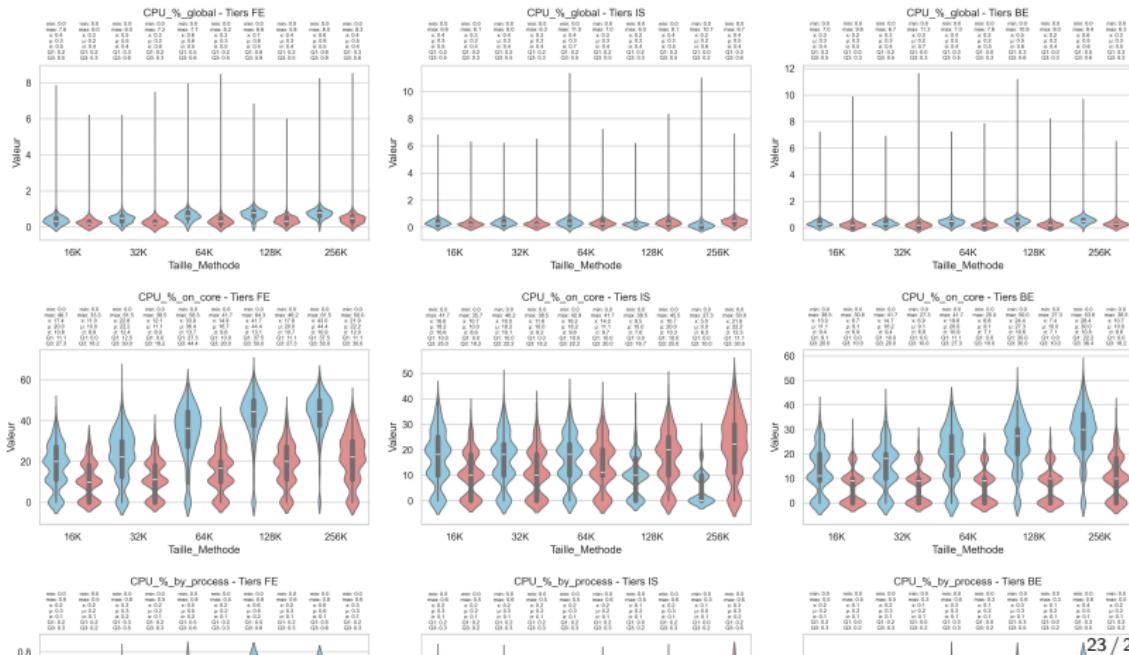
ODB vs Vanilla (aligné)

- IS : réduction CPU notable pour payloads \geq 128 Ko.
- FE et BE : augmentation CPU (BE double pour 256 Ko).
- Latence ODB \geq Vanilla, surtout pour grosses payloads.
- Explication : sockets bloquantes pour transfert de payload virtuelle.



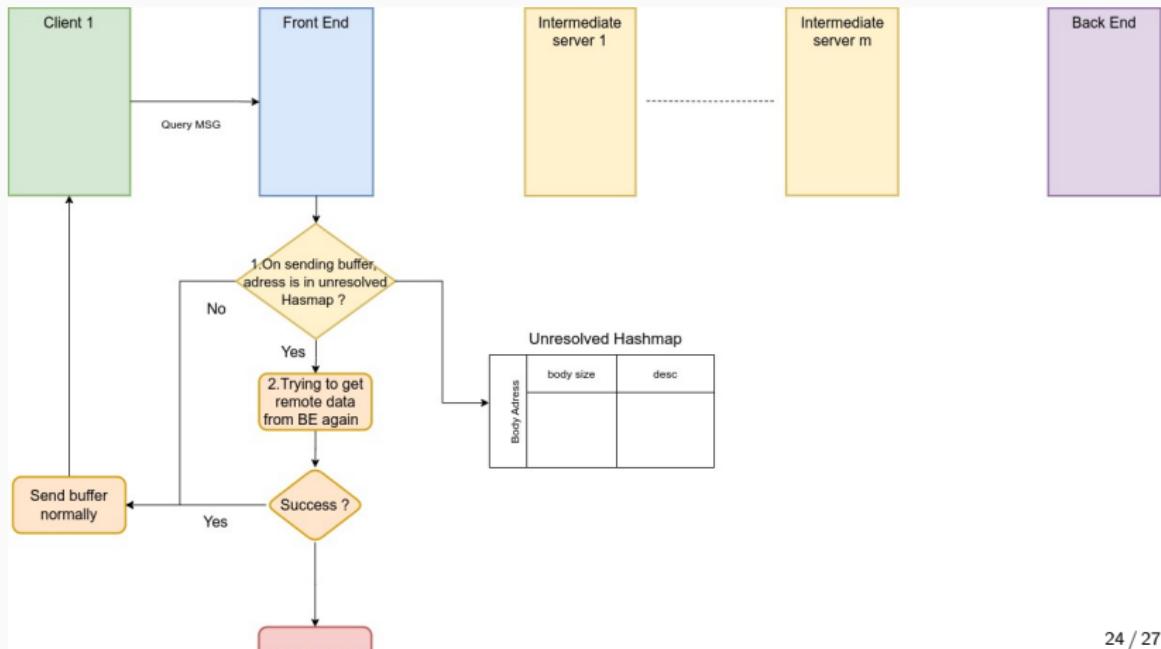
ODB vs Vanilla (non-aligné)

- Tendances similaires à l'aligné.
 - Écart CPU FE/BE plus important : overhead head/tail.
 - IS : réduction CPU pour grosses payloads.
 - Latence : ODB plus élevée, même tendance que pour aligné.



Problème : téléchargement indisponible

- Payload virtuelle : téléchargement depuis BE uniquement à la demande.
- Problème : BE inaccessible → SIGSEGV ou données corrompues.
- Solutions envisagées :
 - ① Arrêt du processus
 - ② Corruption contrôlée
 - ③ Envoi simulé
 - ④ Best-effort (retransmission si possible)



Problème : utilisation mémoire pour buffers virtuels

- Chaque payload virtuelle allouée en mémoire du BE.
- Plus de requêtes → plus de mémoire utilisée.
- Solution : ramasse-miettes :
 - Supprime payloads virtuelles utilisées
 - Thread dédié supprime payloads non utilisées après un délai configurable

Conclusion et perspectives (1/2)

- **ODB** : idée originale et prometteuse pour réduire l'envoi de données redondantes.
- Réduction potentielle de la charge CPU, consommation réseau et énergétique.
- Limites observées :
 - Surcoût CPU et latence liés à l'utilisation de *mprotect*.
 - Augmentation de la latence globale par rapport à vanilla, réduisant la QoS.
 - Charge transférée du serveur intermédiaire vers le Back-End et Front-End.
- Protocole pertinent pour un usage plus écologique et plus respectueux de l'environnement.

Conclusion et perspectives (2/2)

- **Pistes d'amélioration :**
 - Explorer l'utilisation de *userfaultfd* pour remplacer *mprotect*.
 - Étudier l'intégration de sockets non-bloquantes pour exploiter la boucle d'événements de nginx.
 - Développer une API dédiée pour limiter les détournements de fonctions *libc* et améliorer les performances.
- Ces axes pourraient réduire les impacts des limitations actuelles et améliorer la portabilité et l'efficacité d'ODB.

Repo git



Figure: Lien vers mon repo GitHub