Université de Technologie
de Compiègne

TX P20 - NDN

Développement d'une stratégie de cache pour l'internet des objets (IdO) centré sur l'information et simulation avec NDN-sim

Étudiants :
HOANG Quoc Trung
LADEVE Yann

Encadrants :
JABER Ghada
LOUNIS Ahmed

Table des matières

Résumé

La démocratisation de l'accès à Internet (environ 2/3 de la population mondiale auront accès d'ici 2023) et l'explosion du nombre d'appareils connectés à un réseau IP(6) révèle progressivement les failles dans l'architecture Internet. Ce dernier étant fondé sur un modèle de communication d'hôte à hôte, se retrouve inadapté pour asservir des contenus comme des vidéos ou des images. Le paradigme ICN naquit alors dans le but d'établir un réseau qui réduit la latence d'accès aux divers contenus.

1 Named Data Networking

NDN est une implémentation des ICNs faisant partie de l'initiative Future Internet Architecture Program et qui s'inspire des travaux de recherches sur les CCNs (Content-Centric Networking)(7). Avec ce protocole, les utilisateurs envoient un **Interest Packet** afin de demander un contenu à un routeur et reçoivent des **Data Packets** correspondant. Ces noeuds/routeurs doivent donc maintenir trois tableaux :

- **Forwarding Information Base** : stocke le chemin d'accès associé à une donnée et forward les paquets d'intérêt vers l'interface correspondante.
- **Pending Interest Table** : maintient l'interface d'où provient le paquet d'intérêt.
- **Content Store** : cache pour les réponses aux paquets d'intérêts.

Le protocole (10) est défini par les algorithmes (2) pour la réception des Interest Packet et (1) pour la réception des Data Packet.

Algorithm 1: Algorithme de réception d'un Data Packet d'un noeud NDN

```
Input: if, l'interface sur laquelle le message a
       été reçue
       msg, le message reçu avec les
       attributs ;
       index : nom du contenu/chunk du
       contenu souhaité

if msg.type == DATA_Packet then
    cache(msg.data, CS);
    envoyer(msg.data, PIT[msg.index]);
    remove(PIT[msg.index], if);
end
```

Algorithm 2: Algorithme de réception d'un Interest Packet d'un noeud NDN

```
Input: if, l'interface sur laquelle le message a
       été reçue
       msg, le message reçu avec les
       attributs ;
       index : nom du contenu/chunk du
       contenu souhaité

if msg.type == INTEREST_Packet then
    data = CS[msg.index];

    if data != nil then
        envoyer(data, if);
    else
        if msg.index in PIT then
            PIT[msg.index].append(if)
        else
            interface_sortie = FIB[msg.index];
            envoyer(msg, interface_sortie);
        end
    end
end
```

2 Sécurité

La sécurité est un enjeu primordial pour tout développement. NDN est concerné par ce challenge et y répond en permettant de sécuriser directement la donnée. Le producteur de données doit effectuer une signature numérique pour chaque paquet. Cela permet de lier le nom de la donnée, son contenu et l'entité qui a générée cette information. De plus un chiffrement de la donnée peut être effectué. Ces outils d'authenticité et de confidentialité sont intrinsèque à la donnée. Cela implique que dans l'architecture NDN, la sécurité ne dépend ni du conteneur de l'information, ni du moyen de communication (2).

Cependant cette protection cryptographique demande un management des clés et des certificats. NDN permet pour ce faire de mettre un système de relation entre le nom de la donnée et celui du certificat (11).

3 Mécanisme de caching

Afin d'évaluer l'efficacité des mécanismes de caching, les variables et les critères suivants doivent être étudiés :

Hit ratio : fréquence à laquelle on trouve donnée dans le cache.

$$\tau = \frac{n_h}{n} = \frac{nombre\ de\ cache\ hit}{nombre\ d'accs\ total} \quad \tau \in [0,1] \qquad (1)$$

Temps d'accès mémoire et latence : durée nécessaire(s) pour chercher le contenu dans le cache. Notons δ_1 le temps que prends un cycle CPU, δ_{miss} et δ_{hit} le nombre de cycle respectivement pour un miss et un hit. On a donc :

$$\begin{cases} t_{miss} = \delta_{miss} \times \delta_1 \\ t_{hit} = \delta_{hit} \times \delta_1 \end{cases} \qquad (2)$$

D'où la latence totale :

$$\Delta = t_{miss} \times (1-\tau) + t_{hit} \times \tau + \Delta_{com} \qquad (3)$$

Cohérence du cache : critère qui s'assure qu'en présence de plusieurs caches, la donnée récupérée sur un cache est la même que la version la plus récente.

Coût énergétique : l'usage final étant destiné à l'IoT, le coût énergétique des algorithmes/implémentations doit être considéré malgré la difficulté d'évaluer la consommation.

3.1 Catégories de caching

Les mécanismes de caching peuvent être regroupés selon certain critères :

Caching homogène vs hétérogène : Dans les réseaux de noeuds homogènes, chaque noeud traversé par un Data Packet stocke uniformément celuici dans son cache qui est de capacité identique pour tous les noeuds. Pour les mécanismes de caches hétérogènes, tous les noeuds qui se trouvent sur le chemin du Data Packet ne le mettent pas forcément dans leur content store. De plus les noeuds n'ont pas forcément la même capacité de stockage. Supposons deux noeuds CR_1, CR_2, de capacité de stockage différente, se trouvant sur le chemin du data Packet qui est divisé en trois morceau C_1, C_2, C_3. Le routeur CR_1 va stocker dans son content store les deux premiers chunks tandis que le deuxième routeur va stocker seulement le troisième chunk.

Caching coopératif vs non coopératif : Dans les réseaux de noeuds non coopératif, chaque routeur prend de manière indépendante la décision de mettre en cache ou non le data Packet. De même, il ne prévient pas les autres noeuds des informations qu'il possède dans son content store. Au contraire dans un réseau coopératif les noeuds travaillent pour pouvoir stocker plus de données. On peut considérer deux types de coopérations. La **coopération implicte** et **coopération explicite**. La différence se fait en fonction de l'existence d'un mécanisme de diffusion des statuts de chaque routeur. Dans la coopération implicite, ce mécanisme n'existe pas. Les noeuds doivent donc faire des opérations supplémentaires pour définir son choix de mise en cache et/ou de transmission. Au contraire dans la coopération explicite, un noeud changeant d'état doit prévenir les autres routeurs de cette modification.

Caching On-path vs Off-path : Une méthode de cache On-path implique que seulement les noeuds sur le chemin du Data Packet peuvent le stocker dans leur content store. Au contraire pour la deuxième méthode des noeuds qui ne se trouvent pas sur le chemin du Data Packet vont eux aussi pouvoir le stocker.

3.2 Caching homogène et non coopératif

3.2.1 Caching Everything Everywhere

CEE est le mécanisme de cache par défaut dans le NDN. Il est développé pour minimiser la demande de bande passante et la latence. Dans ce mécanisme le Data Packet est stocké dans chaque noeud par lequel il traverse. Le fait qu'il soit non coopératif amène le problème de la redondance car chaque noeud va stocker la même information dans son content store.

3.3 Caching hétérogène et non coopératif

3.3.1 Prob(p)

L'objectif de Prob(p) est de limiter la redondance. Dans ce mécanisme, chaque routeur à la probabilité p de mettre en cache le Data Packet et la probabilité 1-p de ne pas le stocker. Quand un routeur reçoit le Data Packet, il génère un nombre aléatoire x entre 0 et 1. Si $x \leq p$ alors le routeur le met en cache sinon il ne fait que le transmettre. Cette probabilité p est définie par l'administrateur réseau. Plus un Data Packet passe par un noeud plus il a de chance d'être mis en cache dans le content store. En effet cette loi de probabilité est une loi géométrique :

$P(X \geq 1) = \sum_{i=1}^{n} (1-p)^n p = 1 - (1-p)^n$
$\lim_{n\to\infty} 1 - (1-p)^n = 1$

Le Data Packet n'étant mis en cache à chaque

noeud, la redondance en est limitée. Ce mécanisme est non coopératif au vu du comportement indépendant de chaque noeud.

3.4 Caching homogène et coopératif

3.4.1 Breadcrumbs

Breadcrumbs est un mécanisme de caching homogène, en effet chaque noeud sur le chemin du Data Packet enregistre celui-ci dans son CS. De plus, lors de la réception du Data Packet une quatrième table est créée. Cette table contient le nom de la donnée, le noeud d'où vient le Data Packet (null si premier noeud), le prochain noeud où doit être envoyé le Data Packet, ainsi que le temps T_x que met le noeud pour transmettre le Packet. Breadcrumbs crée en amont un seuil de T_s qui va permettre au noeud de savoir ou envoyer l'Interest Packet. Lorsqu'un noeud reçoit le paquet en question, il regarde tout d'abord s'il détient l'information dans son CS.

Sinon, en supposant que chaque noeud utilise une LRU ou FIFO pour son paging algorithme, la probabilité que le contenu en question soit encore dans CS du prochain noeud est plus importante que dans le noeud parent. Dès lors, s'il y a eu un cache hit (ou refresh) depuis une période inférieure au seuil T_s, l'Interest Packet va être transféré à un noeud en aval et sinon vers un noeud en amont.

3.4.2 Intra-AS Co

Intra-AS Co a pour objectif de limiter la redondance en permettant aux routeurs de voir le contenu des routeurs voisins. Pour ce faire, chaque noeud possède deux autres tables : **Local Cache Summary Table** et **Exchange Cache Summary Table**. LCST enregistre les noms des données stockées en local, tandis que le ECST enregistre les noms des données stockées par ses voisins directs. Périodiquement, chaque noeud informe ses voisins du contenu de sa table LCST. Lors de la réception d'un Interest Packet, le routeur regarde s'il possède la donnée voulue dans son LCST. Sinon ce n'est pas le cas, il regarde ECST si l'un de ses voisins le possède. Si l'un de ses voisins possède la donnée en question, il transmet l'Interest Packet à ce voisin. Dans le cas contraire, il transfert la demande au prochain noeud selon ce qui est informé dans le FIB. Le Data Packet est ensuite mis en cache par tous les noeuds sur le chemin du retour.

3.5 Caching hétérogène et coopératif

3.5.1 Centrality-Based Caching

Dans le CBC, le Data Packet est mis en cache dans les noeuds ayant la proximité centrale la plus haute (betweenness centrality). Cette valeur qui est calculé pour chaque noeud permet de rendre compte de la fréquence où ce noeud se trouve sur le chemin le plus court entre deux autres routeurs sur le réseau. BC est calculé comme suit :

$$BC(R_y) = \sum_{\forall R_i R_j \in V \setminus R_y} \frac{n(R_i, R_j, R_y)}{n(R_i, R_j)} \qquad (4)$$

$n(R_i, R_j)$ et $n(R_i, R_j, R_y)$ représentent respectivement le nombre de plus court chemins entre le noeud i et j et le nombre de plus court chemins entre le noeud i et j et passant par y. CBC rajoute un champ dans l'Interest Packet qui prend la valeur BC initialement à 0. Lorsqu'un routeur reçoit l'Interest Packet, il compare sa BC avec celle enregistrée dans le Packet. Si la valeur est supérieure, elle écrase la BC présente. Lorsque l'Interest Packet arrive dans un noeud ayant la donnée dans son CS ou à la source de la donnée. Le Data Packet retourné récupère le champ BC qui était dans l'Interest Packet. Sur le chemin retour, chaque noeud regarde la valeur BC qui se trouve dans le Data Packet. Si elle égale à celle présente localement alors le noeud en question met en cache le Data Packet, sinon il renvoie le Packet en fonction de son PIT. Ce fonctionnement implique que les noeuds avec les plus grands BC sur le réseau auront à changer régulièrement leur cache.

3.5.2 Leave Copy Down

Dans LCD, un *caching suggestion flag* est ajouté dans le Data Packet. Lorsque la source ou un noeud possédant la donnée retourne le Data Packet, ce flag est initialisé à 1. Par la suite les noeuds recevant le Data Packet regarde le status de ce flag, s'il est à 1 le noeud courant enregistre dans son CS le Data Packet et change le status à 0. Une fois passé à 0 les autres routeurs ne font que transmettre le Data Packet au noeud se trouvant dans son PIT et ayant demandé cette donnée. Ce mécanisme permet aux requêtes fréquentes de se trouver aux extrémités du réseau tout en évitant que celles moins demandées ne viennent occuper de l'espace mémoire. Cependant une donnée populaire peut se retrouver à de nombreux noeuds et donc ne répond de manière efficace au problème de redondance.

3.5.3 Move Copy Down

MCD fonctionne de la même manière que LCD mais propose une solution lié au problème de redondance observé plutôt. Lorsqu'un noeud reçoit un Interest Packet dont il possède la donnée dans son CS, ce noeud retourne le Data Packet avec le flag initialisé à 1 et supprime cette donnée de son CS. De ce fait le contenu ne se trouvera que dans un seul noeud sur le chemin entre le requêteur et la source.

LCD comme MCD ont des mécanismes de caching coopératifs, hétérogènes et on-path. En effet seulement les noeuds sur le chemin peuvent mettre en cache le contenu. Avant ça il regarde l'information donnée par le flag pour savoir s'ils doivent faire cette mise en cache. Comme expliqué plus haut, seulement les noeuds qui ont le flag à 1 lorsqu'ils recoivent le Data Packet le mettent dans leur CS.

3.5.4 Cooperative In-Network Caching

CINC est une méthode de caché orientée pour les multimédias vidéos (comme le streaming) qui impliquent un découpage de la donnée.
Introduisons les variables suivantes :

- L_r numéro d'un routeur dans le réseau
- q le nombre de routeurs sur le réseau
- S_d le numéro de série d'un Data Packet
- $z = S_d \mod q$
- D data divisé en 4 morceaux $D/C_1, D/C_2, D/C_3, D/C_4$

De plus deux autres tables sont maintenues en plus du CS, PIT et FIB. **Collaborative Router Table** et **Collaborative Content Store**. Le CRT enregistre le chemin d'accès aux autres noeuds du réseau ainsi que leur numéro L_r tandis que le CCS informe des Data Packets sauvegardés dans les CS des autres noeuds du réseau. Le CRT est renseigné préalablement. CINC fonctionne comme tel, lorsqu'un noeud reçoit un Data Packet (chunk), il regarde si la valeur z correspond à son L_r. Si c'est le cas, il met en cache le Data Packet. Sinon, il regarde sa table CRT et envoie le Data Packet au noeud ayant le bon numéro. De plus il rajoute dans sa table CCS l'information que ce noeud a dans son cache la donnée. Dans tous les cas à la fin de l'une de ces actions, il transfert le Data Packet comme indiqué dans son PIT. Lors de la réception d'un Interest Packet, le noeud regarde de manière séquentielle le CS, CCS, PIT et FIB. Le CINC est une méthode Off-path, en permettant à des noeuds qui ne sont pas sur le chemin de stocker des données dans leur CS.

4 Problème du paging

Puisque la mémoire est une ressource limitée, il est impossible de maintenir l'ensemble des données dans le CS et il est donc nécessaire d'appliquer un algorithme de cache permettant de supprimer certaines lignes et de respecter les contraintes de mémoire.

Vu les lacunes du modèle de complexité classique concernant la prise en compte de l'impact des données différentes, l'analyse compétitive permet d'évaluer la performance d'un algorithme online (non clairvoyant, incrémental avec un jeu de données quelconque) en le comparant à un algorithme offline (clairvoyant). Établir un benchmark des méthodes de caching revient donc à un problème de (h,k)-paging qui cherche à évaluer les performances d'un algorithme online de paging avec un cache de taille k par rapport à un algorithme offline ayant un cache de taille h *avec* $(h \leq k)$. Vu la complexité des démonstrations formelles dans cette approche, nous nous sommes fondés sur les travaux de recherches préexistantes dans ce domaine.

4.1 Algorithme de Bélády

Cet algorithme optimal (Cf. Algorithme 3) remplace la ligne de cache qui ne sera pas utilisée pour la plus grande période de temps (4). Cet algorithme offline doit donc connaître les accès futurs (d'où son nom d'algorithme clairvoyant) et ne sert que de référence pour la performance des autres algorithmes. En stockant les adresses de caches dans un tableau (array) de nom de contenu.

Le nombre de cache miss correspond donc à la somme de la capacité de la table de cache m et du nombre de mauvais choix de remplacement. Cet algorithme demeure tout de même optimal lorsque l'hypothèse de l'uniformité des contenus demandées puisqu'elle garantit le hit ratio optimal et les opérations peuvent se produire en complexité $O(1)$. Néanmoins, aucune implémentation préalable est disponible dans le projet NDN-sim.

4.2 Least Recently Used

L'algorithme LRU remplace la ligne utilisée la moins récemment et nécessite donc que tous les accès aux différentes lignes de cache sont enregistrés. La méthode naïve (Cf. Algorithme **??**) consiste en

Algorithm 3: Implémentation théorique de l'algorithme de Bélàdy

Input: x - lignes de cache demandées
Data: f - table de cache vide

```
hit = 0;
for i = 0; i < len(x); i = i + 1 do
    if not find(x[i], f) then
        hit = hit + 1 ;
    else
        if len(f) < capacity(f) then
            push_back(f, x[i])
        else
            j = furthestOccurrence(x, x[i+1])
            f[i] = j
        end
    end
end
```

l'utilisation d'une matrice $R \in M_{n,n}$ avec n la capacité du cache (en nombre d'entrées). L'étude (9) démontre dès lors que la famille des algorithmes LRU est $\frac{k}{k-h+1}$-competitive.

Néanmoins, cette solution n'est pas adaptée aux caches de grande taille puisque la complexité et la consommation en mémoire explosent selon la taille de la matrice. Les algorithmes d'approximation adressent alors ce constat en proposant des structures de données plus efficaces.

Algorithm 4: Algorithme naïve LRU

Input: x - ligne de cache demandée
Data: R - matrice LRU

```
R̄_x = 𝟙_n  with 𝟙_n = ( 1...1 );
R_x = 0_n  with 0_n = ( 0...0 )^T;

max = 0;
for i = 0; i < n; i = i + 1 do
    val = binary(R_i);
    if val > max then
        max = i;
    end
end
return max;
```

4.3 LRU approximation

4.3.1 Tree - PLRU

Cette approximation (Cf. Algorithme **??**) utilise un arbre décision binaire dont chaque feuille représente une ligne de cache i et chaque noeud correspond à un flag de un bit (0 représente l'instruction aller à droite et 1 à gauche). A chaque passage par un noeud, le flag est inversé, ce qui garantit que la prochaine donnée sera stockée dans une page non récente.

Algorithm 5: Algorithme Tree - PLRU

Input: x - ligne de cache demandée
Data: A - racine de l'arbre binaire de décision

```
CurrentNode = A;
while CurrentNode.Left != nil and
  CurrentNode.Right != nil do
    if CurrentNode.Flag == 1 then
        NextNode = CurrentNode.Left;
    else
        NextNode = CurrentNode.Right;
    end
    CurrentNode.Value = not
      CurrentNode.Flag;
    CurrentNode = NextNode;
end

store(x, CurrentNode.Page)
```

Par cette approximation, le résultat obtenu est sous-optimal en échange d'un gain en mémoire énorme (un arbre binaire de $nlog_2(n)$ noeuds et un tableau de cache de n lignes).

4.3.2 Bit - PLRU

De manière similaire, l'algorithme de Bit - PLRU stocke un bit intitulé MRU flag au niveau de chaque ligne de cache. Chaque accès met le bit de la ligne correspondante en 1 et lorsque le dernier bit 0 devient un 1, les autres sont tous réinitialisés à 0. A l'issu d'un cache miss, la ligne ayant un bit MRU égal à 0 et l'index le plus petit (dans l'ordre du nom des lignes de cache) est remplacé.

4.3.3 FIFO / LIFO / FILO

Le stockage et la suppression selon l'ordre d'accès dans le cache permet aussi d'approximer le LRU sans prendre en compte le nombre d'accès et ignore

complètement le passé au-delà de la capacité de la structure de donnée.

De même, des optimisations peuvent se produire au niveau de l'algorithme FIFO notamment au niveau de la structure de donnée. Le **Clock page replacement** introduit une file circulaire avec un pointeur vers la ligne de cache actuelle.

Algorithm 6: Algorithme Clock LRU

Input: x - ligne de cache demandée
Data: F - file circulaire de cache

```
if CacheMiss(x) and F.NumElements() ≥
 F.Capacity() then
    replaced = false;
    while not replaced do
        if F.Current.Flag == 0 then
            F.Current.Page = x;
            replaced = true;
        else
            F.Current.Flag = 0;
            F.Current = F.Current.Next;
        end
    end
end
```

4.4 Improved LRU

Les algorithmes d'approximation de LRU et de la famille LRU souffrent donc tous du cache poisoning, phénomène dans lequel les données qui sont accédées qu'une fois occupent une grande partie du cache.

4.4.1 LRU-K

L'algorithme (1) découpe la mémoire cache en K groupes selon le nombre d'accès et maintient un historique des K derniers accès. A l'issu d'un cache miss, la ligne ayant la donnée qui n'a pas été accédée récemment et dont le K-ième accès est le LRU de la liste K correspondante est remplacée. Cette algorithme adresse le problème du cache poisoning au dépit d'une consommation beaucoup plus importante.

4.5 Least Frequently Used

Contrairement aux algorithmes de la famille LRU, les algorithmes LFU stockent le nombre d'accès d'une ligne de cache et non pas le dernier accès.

Algorithm 7: Algorithme LFU

Input: x - ligne de cache demandée
Data: H - hash map du cache

```
if x in H then
    H[x].Count++;
else
    if len(H) ≥ capacity(H) then
        remove(min(H.Count));
    end
    insert(H,x);
    H[x].Count = 1;
end
```

Cette famille d'algorithme (Cf. Algorithme (7)) souffre tout de même d'un problème de cache poisoning dans lequel un accès qui s'est produit à plusieurs reprises en instant t et plus du tout après reste toujours dans le cache. De plus, que la dimension temporelle est absente, les éléments nouveaux ont tendances de disparaître plus rapidement. Ceci peut engendrer un état de miss rate à 100% si on rencontre une succession de 0101010, deux données non présentes dans le cache auparavant.

4.6 Hybrid LFU LRU

4.6.1 Least Frequently Recently Used

L'algorithme LFRU (5) combine les approches LFU et LRU et divise le cache en deux partitions ; une partition privilégiée qui stocke les accès populaires (qui utilise un LRU en tant qu'algorithme de paging) et une partition non-privilégiée (qui utilise une PLFU). Les contenus les moins utilisés perdent leur statut privilégié (et rentre dans la partition non privilégié) et sont évincés lorsqu'ils ne sont pas récemment utilisés (LRU).

L'inclusion de la dimension temporelle permet dès lors de négliger les effets du cache poisoning sans pour autant introduire une structure de donnée lourde ou non intuitive. De manière similaire, l' algorithme **LFUDA** (LFU with Dynamic Aging), ajoute l'âge de cache au niveau de chaque ligne de cache pour prendre en compte cette dimension.

4.6.2 Adaptive Replacement Cache

L'ARC maintient deux structures "ghost" qui contiennent des méta-données sur les données évin-

cées de la partition privilégiée et non-privilégiée, en plus de ses partitions. En notant, P_1 la partition non-privilégiée P_2 la privilégiée et G_1, G_2 leur ghost respectives, on a donc le schéma suivant :

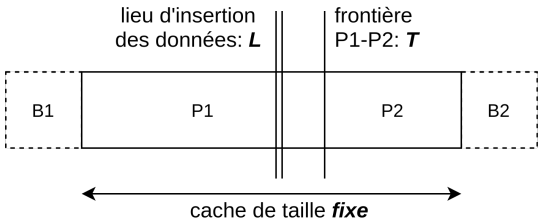


FIGURE 1 – Fonctionnement d'un cache ARC

Les nouvelles données sont alors insérées à gauche de L dans P_1 et poussent les anciens vers la gauche. Celles qui sont évincées de P_1 rentrent alors dans G_1 (gardant seulement leur méta-données) et progressivement poussées vers la gauche jusqu'à la suppression. Les données dans G_1 ou P_1 qui sont référencées encore une autre fois passent dans P_2, insérées à droite de L et procèdent de manière similaire dans le sens inverse.

Les deux partitions ne sont pas fixes (la frontière T varie selon le jeu de données) et remplissent dynamiquement le cache de taille fixe.

4.7 Stochastic Cache

La famille de caching stochastique, elle, choisit aléatoirement une ligne de cache pour être remplacée une fois la capacité limite du cache atteinte. Ces algorithmes nécessitent aucune connaissances des accès futurs ou antérieurs et demeurent donc les plus simples à implémenter et les moins consommatrices en terme de ressources.

4.8 ML Caching

4.8.1 Adaptative Caching using Multiple Experts

La stratégie d'ACME (Ari et al.) pioche la décision optimale (qui correspond au mieux à la charge actuelle sur le réseau) parmi un pool d'algorithmes de remplacement de cache statique.

A chaque remplacement, chaque stratégie vote, en utilisant ses propres critères sur l'objet à remplacer. L'algorithme ACME utilise alors le retour du cache sur les hit/miss pour varier le poids des votes. Cette méthode est optimale si l'augmentation du coût d'un accès dû à l'abstraction du cache est compensée par la croissance du taux de cache hit.

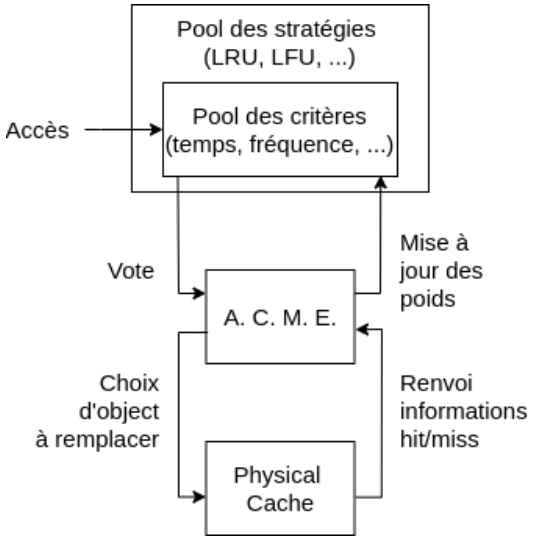


FIGURE 2 – Fonctionnement d'un cache ACME

5 Fog Computing

5.1 Cloud Computing

Le Cloud Computing est un modèle mis en place depuis plusieurs années qui permet aux entreprises de déléguer les contraintes de management d'entrepôts de données ainsi que le coût de process lié à ces données. Ce modèle est centralisé, cela veut dire que toutes les données et les requêtes passent par un point central. Cette disposition implique qu'un temps de transfert a lieu entre le moment où l'utilisateur effectue une action et le moment où il reçoit une réponse.

L'émergence de l'Internet of Things a vu l'apparition de données très nombreuses et variées apportant de nouvelles contraintes comme des réponses en temps réels, une gestion de l'hétérogénéité des données, un réseau très dispersé géographiquement ou encore la mobilité des appareils. Ces nouveaux challenges ne peuvent être répondus avec le seul modèle du Cloud Computing.

Le Fog Computing apparaît donc comme une réponse à ces contraintes.

5.2 Architecture Fog Computing

Le modèle Fog Computing vient compléter le modèle du Cloud Computing en rajoutant une couche entre le cloud et les appareils terminaux. Cette couche se trouve aux extrémités du réseau internet à proximité des appareils.

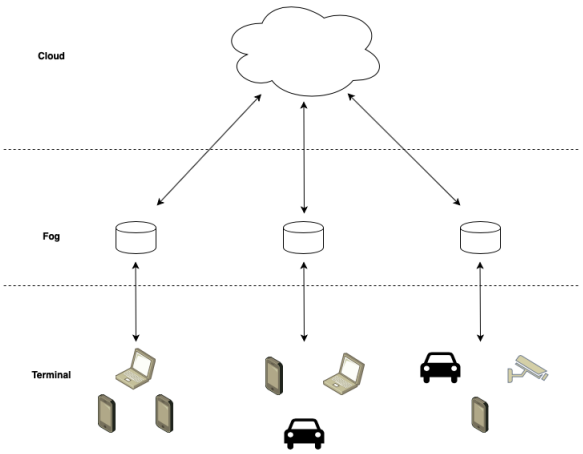


FIGURE 3 – Fog Computing

Elle est composé de noeuds qui peuvent être des routeurs, serveurs, ou encore des passerelles. Ces noeuds peuvent être fixe ou en mouvement. Les appareils ayant besoin d'effectuer des opérations peuvent s'y connecter pour utiliser les services de transport, de process ou de mise en cache notamment. Le fog est aussi relié au cloud pour pouvoir utiliser tous les avantages de celui-ci (capacités de calcul et de stockage plus importantes).

Le positionnement des noeuds en périphérie du réseau permet une proximité avec les appareils. Cela réduit le temps de transport des données et permet donc de répondre aux IoT ayant besoin de retour en temps réel. La plupart du temps les données générées n'ont pas besoin d'aller dans le Cloud pour être traitées. Le Fog prenant ce rôle, cela permet donc de diminuer la quantité de transmission des données et donc d'améliorer la bande passante.

Au vu des challenges apportés par le Fog, NDN se place comme candidat majeur dans l'implémentation de cette architecture. En effet, les noeuds présent sur le réseau peuvent mettre en cache des informations ainsi que supporter des tâches.

6 Simulation

6.1 Architecture NDN-sim

6.1.1 NS-3

L'architecture de NDN-SIM se fonde sur NS-3, un simulateur de réseaux à évènement discret, en construisant sur les abstractions :

1. **Application** : programme à simuler
2. **Channel** : voix de communication entre différentes noeuds (canal)
3. **Net Device** : composant simulant les pilotes physiques et logicielles permettant à un noeud de communiquer via un canal à d'autre noeuds.
4. **Node** : entité de calcul de base (noeud) capable d'héberger un ou plusieurs Application et Net Device indépendants.
5. **Topology Helper** : constructeur de topologie du réseau qui peuple un ensemble de noeuds une grille et les relient.

6.1.2 ndn-cxx

La librairie Ndn-cxx correspond à une librairie bas niveau en C++14 qui introduit des primitives utilisées pour les applications de plus haut niveau. Cette dernière propose notamment des fonctionnalités d'encodage et de décodage (`ndn-cxx::encoding`), de sécurité et de signature (`ndn-cxx::security`) ainsi qu'une implémentation partielle d'une interface (`ndn-cxx::face`) de communication de bas niveau (permettant d'exprimer un Interest Packet et la publication des données).

6.1.3 Named Data Networking Forwarder

Le composant Named Data Networking Forwarder (NFD) implémente le modèle d'acheminement des paquets (Cf. Section (1)). Concrètement, il défini les tables (PIT, CS, FIB) et certaines stratégies de cache (`nfd::cs::lru`, etc..) et d'acheminement. Ce dernier expose l'API face (`nfd::face`), une abstraction pour envoyer et recevoir des paquets NDN.

6.1.4 NDN-Sim

La couche de simulation (NDN-Sim) se trouve au dessus de la couche de prototype précédente et est constituée de trois composants :

1. **Core** : implémentation du protocole NDN.

2. **Utilities** : constructeurs de topologies, Packet tracers (au niveau de l'application, des canaux ou du réseau).
3. **Helpers** : configuration et installation du protocole stack NDN

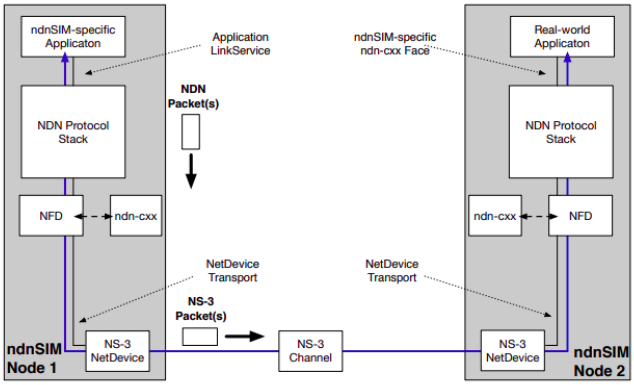


FIGURE 4 – NDN packet exchange process between two simulated nodes(8)

6.1.5 Simulation des applications

Afin de faciliter les simulations, ndn-SIM proposent deux familles implémentations d'Applications de base :

1. Les **Consommateurs** : des applications qui simulent un noeud qui demande à une fréquence aléatoire des contenus en émettant des interest packet sur le réseau. Par défaut, la librairie propose trois consommateurs :
   (a) **ConsummerCbr** : qui envoie des requêtes à des intervalles de temps défini selon une loi de probabilité donnée.
   (b) **ConsumerBatches** : qui envoie des groupes de requêtes à des intervalles de temps défini selon une loi de probabilité donnée.
   (c) **ConsumerWindow** : qui envoie des requêtes selon le principe des fenêtres glissantes. Un groupe de requêtes est initialement envoyé l'un suivant l'autre sans attente d'une réponse. Au fur et à mesure des réponses, les requêtes qui suivent dans le queue sont envoyées tout en respectant la taille maximale de la fenêtre.
2. Les **Producteurs** : des applications qui produisent des contenus de taille variable à intervalle régulier.

6.2 Construction des scénarios

6.3 Architecture

L'objectif étant de simuler un réseau de capteurs possédant des contraintes sur la taille du cache et communiquant par wifi, deux modèles ont été mis en place.

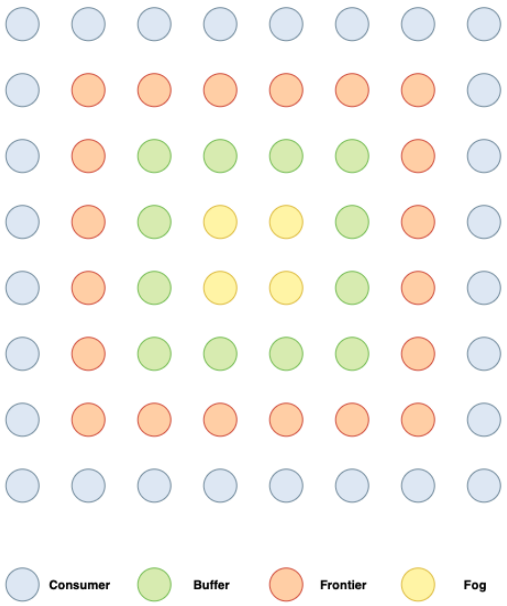


FIGURE 5 – Réseaux de capteurs avec trois niveaux

Chaque modèle possède quatre types d'individus. Les noeuds nommés *fog* correspondent aux producteurs de données, ils se trouvent au centre du réseau. Ensuite plusieurs couches de capteurs se trouvent autour des noeuds fogs, ce sont les noeuds nommés *buffer* et *frontier*. Pour le deuxième modèle la couche buffer est composée de deux niveaux. Les noeuds frontiers correspondent au capteur les plus éloignés du centre du réseau et donc les plus proches des générateurs d'interest packet. Finalement nous trouvons sur la dernière couche les noeuds *consumer*.

La connection entre ces noeuds se fait via un réseau sans fil (comme défini par l'IEEE dans le protocole 802.11a) adhoc. Puisqu'on simule un réseau de capteur statique, les noeuds sont immobiles et la capacité de stockage en cache des machines sont restreinte à 15 paquets.

Pour chacun de ces modèles différent scénarios ont été mis en place. Pour avoir un modèle de référence nous avons tout d'abord simuler un réseau où aucun noeud ne cache les packets qu'il reçoit. Puisqu'une

simulation de 60s d'un scénario particulier nécessite environ 9h d'exécution, les différents méthodes de caching sur le réseau que nous avons testé sont le *Caching Everything Everywhere*, *Caching only on frontier nodes* et *Caching only on buffer nodes*. Pour chaque méthode de caching, les méthodes de remplacement de cache locale suivante ont été testées : LFU, LRU, Fifo et random.

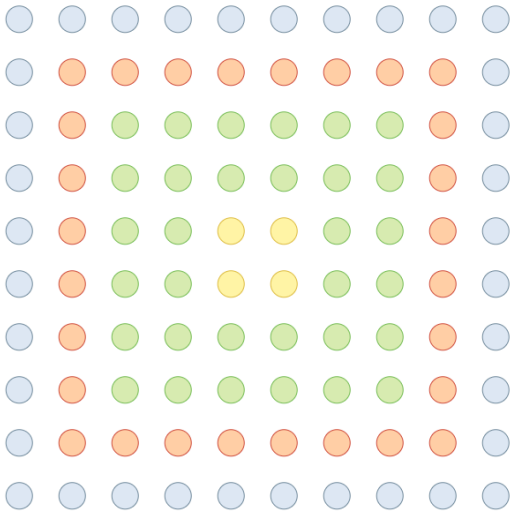


FIGURE 6 – Réseaux de capteurs avec quatre niveaux

6.4 Évènements

On a étudié deux cas de figures :

— Les noeuds *consummer* vont chacun demander 100 contenus par seconde sans suivre une distribution aléatoire particulière à un préfixe commun */root*. La simulation dure alors 60s et chaque le temps moyen d'exécution d'un scénario est de 9h.
— Les noeuds *consummer* vont chacun demander 30 contenus par seconde sans suivre une distribution aléatoire particulière à un préfixe commun */root*. La simulation dure alors 15s et chaque le temps moyen d'exécution d'un scénario est de 5 minutes.

Dans les deux cas, les démarrages des consommateurs sont décalés de 1 seconde afin d'augmenter la variance dans l'initialisation du réseau et obtenir des résultats qui reflètent plus fidèlement la réalité. Les noeuds *producers*, eux, produisent des contenus de taille moyenne 1024 bytes sur le même préfixe.

6.5 Évaluation des performances

L'évaluation des performances se fait à travers deux métriques ;

— La **latence** au niveau des applications consommatrices qu'on obtient grâce à `ndn::AppDelayTracer` en interprétant l'attribut *FullDelay* (temps entre l'envoi du premier Interest Packet et la réception du dernier chunk du contenu demandé).
L'attribut *LastDelay* n'est pas employé puisqu'elle sur-estime la performance du réseau en ne prenant en compte que le délai entre l'envoi du dernier Interest Packet et la réception du dernier chunk du contenu demandé.
— Le **hit ratio** au niveau des caches qu'on obtient grâce à `ndn::CsTracer` en calculant pour un instant t donné le quotient :

$$\frac{CacheHits}{CacheMisses + CacheHits} \tag{5}$$

Initialement, ces métriques sont évaluées par les Tracers à des instants t fixes (pour le cache, toute les 1 secondes pour chaque noeuds et pour la latence, à chaque réception finale du contenu). Tout de même, cette méthode d'évaluation introduit une variance élevée et réduit fortement l'interprétabilité des données. Il faut donc discrétiser l'axe du temps en des intervalles réguliers de 5 secondes puis calculer la moyenne et l'écart-type empirique sur ces intervalles pour en déduire la métrique.

6.5.1 Modèles de références

Par contrainte de temps, on a choisi un modèle de référence où on cache nulle part sur le réseau. Il serait tout de même possible d'établir deux modèles de références :

— **No cache** qui constitue la borne inférieure de l'estimation de la performance de l'architecture.
— **Optimal cache** qui utilise l'algorithme de Bélády (Cf. Section (4.1)) pour le remplacement de cache et une stratégie de caching coopérative Cache Everything Everywhere (3.2.1). Ce modèle purement théorique qui nécessite en même temps la connaissance à priori des données et une capacité de stockage homogène importante sur le réseau entier constitue alors la borne supérieure de la performance.

Dans ce cas de figure, l'évaluation des performances reviendra donc à calculer pour chaque mé-

trique les ratios :

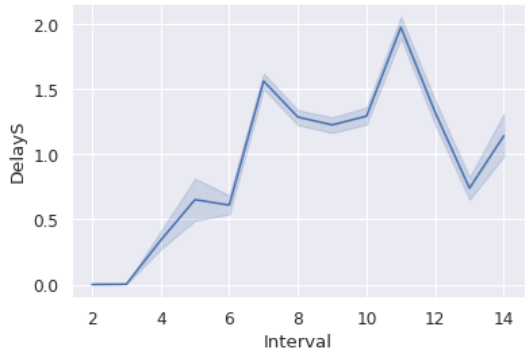
$$metric = \frac{metric_{observed} - metric_{no\ cache}}{metric_{optimal\ cache}} \quad (6)$$

6.5.2 Observations

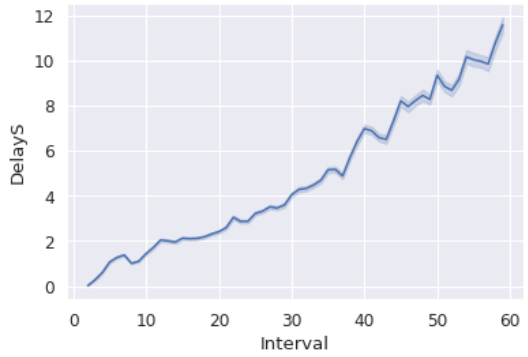
En regroupant l'ensemble des scénarios, on observe qu'au cours du temps le réseau se sature progressivement avec des requêtes. Cette saturation se produit cycliquement en plusieurs phases ;

— une première phase où les consommateurs envoient leurs intérêts suivie de sa propagation. Les différentes messages se propagent alors dans le réseau jusqu'à ce qu'un noeud ait le contenu dans son content store et innondent alors le réseau.

— dans un deuxième temps, les données sont retournées au consommateur et la latence moyenne diminue avant une deuxième vague d'intérêts qu'on intitule la **détente**



(a) durée de simulation : 15s



(b) durée de simulation : 60s

FIGURE 7 – Evolution des latences de l'application au cours du temps et de la durée de la simulation avec des intervalles communes à 1s

Tout de même, même si la détente du réseau se produit cycliquement, le réseau a tendance de se saturer plus rapidement que la détente (Cf. Figure (7)). La durée de simulation influence donc fortement la performance du réseau dans notre scénario : plus un réseau est "vieux" plus sa performance s'appauvrit.

6.5.3 Cache Everything Everywhere

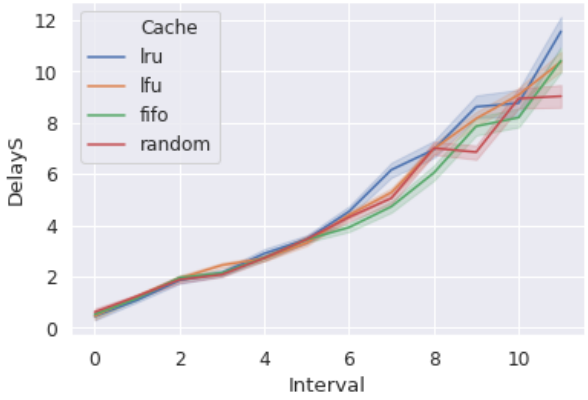


FIGURE 8 – Latence moyenne des différentes stratégies de cache local à intervalle ($I = 5s$) pour une simulation de 60s

En appliquant les différentes stratégies de cache (Cf. Figure (8)) dans une stratégie de cache coopérative CEE, on observe que les périodes de détente des remplacements de cache ne sont pas les mêmes. Par exemple, lors que le scénario avec lru est en détente, le scénario avec un remplacement aléatoire (random) atteint son pic.

De plus, la disparité en terme de latence entre les stratégies de remplacement devient plus marqué lorsque la durée de la simulation se prolonge. Au bout d'un certain temps, la stratégie du choix aléatoire, étant la moins coûteuse, n'est plus fiable.

L'algorithme FIFO donne logiquement une performance plus faible que la LRU puisqu'il s'agit de son approximation. Tout de même, il est surprenant que sa latence soit comparable au LFU. Ceci peut provenir de la durée pas assez importante de la simulation qui ne permet pas de les différencier.

En ce qui concerne la latence, le LRU semble bien complémenter le CEE ; lorsqu'un noeud reçoit un data packet il va forcément le stocker et remplacer le paquet le moins récemment utilisé.

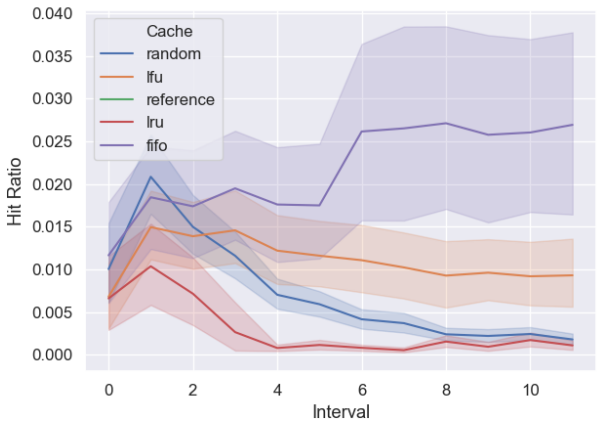


FIGURE 9 – Hit Ratio moyens des différentes stratégies de cache local à intervalle ($I = 5s$) pour une simulation de 6s

Les hits ratios sont globalement tous très faible, le meilleur étant avec fifo mais ne dépassant pas les 3%. Ces résultats s'expliquent principalement par la taille mémoire de chaque noeud. En effet cette capacité étant très réduite, les noeuds ne doivent pas pouvoir garder de nombreux contenus différents. De ce fait, les caches sont renouvelés très régulièrement et empêchent les noeuds de pouvoir répondre positivement à un interest packet. Il est également possible que le temps de simulation ainsi que le nombre de contenus demandé par seconde ne soient pas adaptés pour ces scénarios.

6.5.4 Cache only in buffer nodes

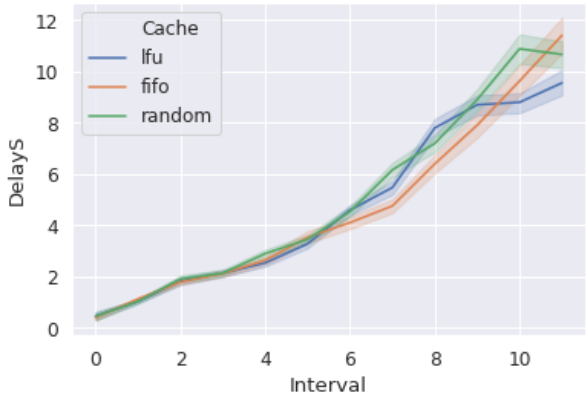


FIGURE 10 – Latence moyenne des différentes stratégies de cache local à intervalle ($I = 1s$) pour une simulation de 15s (lru n'est pas inclue à cause d'une erreur dans la simulation)

L'évolution de la latence moyenne (Cf. Figure (10)) suggère que la stratégie de restreindre le cache au noeud buffer est plus sensible au cache poisoning à l'échelle local dont la performance pauvre des remplacements de cache, relativement proche d'un cache aléatoire en témoigne.

6.5.5 Cache only in frontier nodes

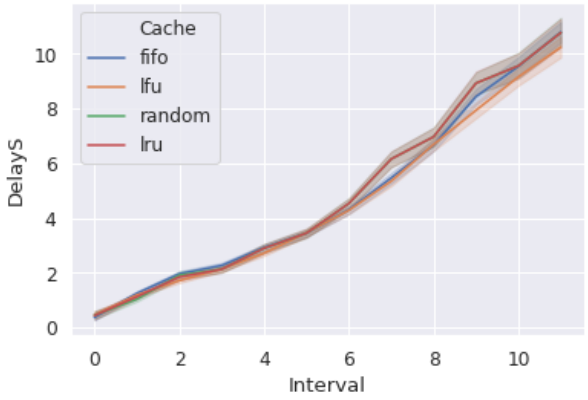


FIGURE 11 – Latence moyenne des différentes stratégies de cache local à intervalle ($I = 1s$) pour une simulation de 15s

Sur ces deux simulations de 15s, le hit ratio reste nul tout au long de la simulation. Comme expliqué plus haut les paramètres de scénarios ne sont peut être pas adaptés à l'architecture de nos réseaux.

6.5.6 Etude de l'évolutivité

Afin de prouver l'évolutivité de l'architecture, on a étudié l'influence du nombre de couche intermédiaire entre les noeuds frontaliers et les noeuds fogs (Cf. Figure (13). Toutes les scénarios de cache ont été prises en compte afin d'augmenter le nombre de scénarios différentes et donc de réduire la variance du résultat.

On observe alors qu'au cours du temps que la latence moyenne ne varie pas trop entre les deux modèles tandis que son écart inter-quartile a légèrement augmenté. Tout de même, puisque la différence entre l'écart-type est inférieur à 0.01%, on peut en conclure que l'architecture peut être élargie sans trop faire souffrir la performance.

De plus, l'architecture est évolutive par rapport aux nombres de noeuds ainsi que la taille du cache (Cf. Figure (13)) ; l'augmentation de la taille de cache améliore en générale la performance.

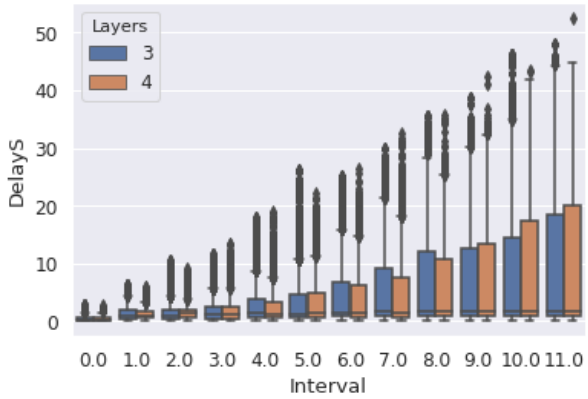


FIGURE 12 – Evolution de la latence moyenne selon le nombre de couche de l'architecture à intervalle ($I = 5s$) pour une simulation de 60s

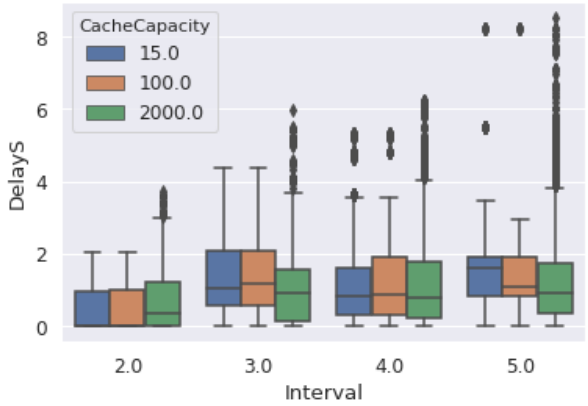


FIGURE 13 – Evolution de la latence moyenne selon la taille du cache à intervalle ($I = 2s$) pour une simulation de 15s

7 Conclusion

Pour rappel l'intitulé de cette TX est le suivant :

Développement d'une stratégie de cache pour l'internet des objets (IdO) centré sur l'information et simulation avec NDN-sim

Pour pouvoir essayer de mettre en place cette stratégie, il nous a fallut préalablement passer par différentes étapes. Tout d'abord une étape de recherche et d'apprentissage sur les thèmes clés de la TX, notamment le paradigme NDN modèle centré sur le contenu plutôt que sa localisation. Nous avons pu comprendre les différentes stratégies de cache locales et globales qui peuvent être mises en place dans un tel réseaux. La mise en place d'un tel réseau s'est fait dans le cadre de l'Internet des Objets. De ce fait, des contraintes comme le type de communication ou la capacité de stockage ont du être considérées. Une fois cette phase théorique effectuée, l'objectif a été de mettre en application toutes ces connaissances grâce à NDN-sim. Ce simulateur permettant d'implémenter des réseaux NDN a été compliqué à prendre en main, n'étant développé que dans un cadre expérimental. Après avoir compris les blocs clés permettant de simuler un réseau NDN complet, nous avons mis en place différents scénarios.

Dans le cadre de cette TX, uniquement des réseaux en grille avec différentes profondeurs ont été considérés. Dans ce cas figure, les combinaisons suivantes des stratégies de caches sont optimales :

Cache coopérative	**Remplacement de cache**
Cache Everything Everywhere	LRU
Buffer Nodes Only	FIFO/ Random
Frontier Nodes Only	FIFO/ LRU

7.1 Difficultés et méthodologie

Afin de généraliser l'approche pour étudier les performances, il est nécessaire, pour chaque architecture, d'étudier des scénarios qui correspondent à une combinaison de différentes critères :

1. la **stratégie de cache locale** (lfu, lru, fifo, random)
2. la **stratégie de cache coopérative**
3. la **capacité de stockage** (taille des caches en nombre de paquets des noeuds)
4. le **temps de simulation**
5. l'enchaînement des **évènements** qui surviennent dans le réseau (fréquence et loi de consommation des contenus sur le réseau, des coupures, la taille des contenus produits, etc...)
6. le **modèle de mobilité des noeuds** (distance entre les noeuds, manets ou vanets, etc...)

Ceci engendre une explosion combinatoire des scénarios à étudier qui est magnifiée par le fait que les noeuds puissent être hétérogènes (et donc avoir des combinaisons de critères différentes).

De plus, le temps d'exécution des scénarios varient fortement (de l'ordre de quelques minutes à plusieurs

heures) selon la durée de simulation, le nombre de noeuds et la fréquence de consommation des contenus. Or comme observé dans la Section (6.5.2), la durée de la simulation influence de manière non négligeable la performance du réseau.

L'approche par optimisation des différents critères n'est donc pas envisageable pour développer une stratégie de cache *généraliste* pour les réseaux ndn. L'approche à prendre serait plutôt d'effectuer des recherches sur une sous-partie de l'espace des critères puis de raffiner selon les résultats obtenus. Il est aussi nécessaire de discrétiser la dimension temporelle pour réduire les variances des performances et obtenir des résultats plus interprétables.

7.2 Travaux futurs

Dans l'étude de l'architecture, il serait intéressant de prendre en compte le coût en terme de consommation mémoire, CPUs et énergétique. Ces métriques sont obstruées par ndnSIM mais disponible dans ns-3 (certes, sans documentation). On peut alors adapter la procédure de test des scénarios développée pour d'autres architectures.

8 Code

Vous trouverez l'ensemble du code effectué lors de cette TX ici

Références

[1] Acharya, S., Alonso, R., Franklin, M., and Zdonik, S. (1995). Broadcast disks : data management for asymmetric communication environments. In *Mobile Computing*, pages 331–361. Springer.

[2] Afanasyev, A., Burke, J., Refaei, T., Wang, L., Zhang, B., and Zhang, L. (2018). A brief introduction to named data networking. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 1–6. IEEE.

[Ari et al.] Ari, I., Amer, A., Gramacy, R. B., Miller, E. L., Brandt, S. A., and Long, D. D. Acme : Adaptive caching using multiple experts.

[4] Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2) :78–101.

[5] Bilal, M. and Kang, S.-G. (2017). A cache management scheme for efficient content eviction and replication in cache networks. *IEEE Access*, PP.

[6] et al., A. (2020). Cisco annual internet report (2018–2023). Technical report.

[7] Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., and Braynard, R. L. (2009). Networking named content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, page 1–12, New York, NY, USA. Association for Computing Machinery.

[8] Mastorakis, S., Afanasyev, A., and Zhang, L. (2017). On the evolution of ndnsim : An open-source simulator for ndn experimentation. *ACM SIGCOMM Computer Communication Review*, 47(3) :19–33.

[9] Sleator, D. D. and Tarjan, R. E. (1985). Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2) :202–208.

[10] Zhang, M., Luo, H., and Zhang, H. (2015). A survey of caching mechanisms in information-centric networking. *IEEE Communications Surveys & Tutorials*, 17(3) :1473–1499.

[11] Zhang, Z., Yu, Y., Zhang, H., Newberry, E., Mastorakis, S., Li, Y., Afanasyev, A., and Zhang, L. (2018). An overview of security support in named data networking. *IEEE Communications Magazine*, 56(11) :62–68.