Consommation énergétique des frameworks de développement dans la vraie vie

AUBERTIN Philippe
philippe.aubertin@axopen.com

COMBE Arthur
arthur.combe@axopen.com

MITTELETTE Nathan
nathan.mittelette@axopen.com

CHARMEAU Antoine
antoine.charmeau@axopen.com

CHABOUD Thomas
thomas.chaboud@axopen.com

VIDAL Romain
romain.vidal@axopen.com

JOURNET Alexandre
alexandre.journet@axopen.com

February 28, 2023

Abstract

L'objectif de cette étude est d'analyser l'impact énergétique relatif aux usages des frameworks dans le cadre de développement d'applications web.

Le choix d'un framework a-t-il un impact significatif sur la consommation énergétique ?

Cet usage peut-il être mesuré dans la pratique, autrement dit, dans la "vraie vie" ?

1 Qui sommes-nous ?

Créée en 2007 à Lyon, AXOPEN est spécialisée dans les développements de projets informatiques métiers, techniques et complexes. Composée de plus de 40 passionnés (développeurs, chefs de projet et experts) qui partagent des valeurs et des convictions communes autour de l'informatique.

Avec pour moteur la performance et la qualité du SI, nos équipes travaillent en collaboration avec des PME et grands comptes de tous secteurs d'activités, de l'assurance à l'énergie en passant par l'industrie, pour répondre à des enjeux métiers spécifiques.

2 Introduction

Nous savons aujourd'hui que l'usage que nous faisons de la technologie au sens large est très consommateur de ressources, en particulier dans les data centers au sein desquels nous hébergeons nos applications.

Dans la littérature scientifique, de nombreuses analyses portant sur la consommation des serveurs au sein des data centers existent. Cependant, il est toujours exposé dans ces dernières des moyennes mesurées sur un ensemble de serveurs, et ce, indépendamment de l'usage qui en est fait. Or, en tant que développeurs, il nous parait essentiel justement de nous questionner sur nos usages réels et leurs impacts.

Partant de ces constats, nous avons choisi à travers cette étude de relever le pari suivant : mesurer et comprendre l'impact de nos usages des frameworks de développement dans "la vraie vie".

Pour répondre à cette problématique, nous avons pris le parti d'analyser les pratiques de développement pour voir si cela avait un impact mesurable et quantifiable.

L'objectif est le suivant : savoir si nous reproduisons les résultats de l'article "Energy Efficiency across Programming Languages" [**Energy**] non pas au niveau des algorithmes standards, mais au niveau du framework, sur une utilisation la plus proche de la réalité possible, c'est-à-dire, reproduisant le plus fidèlement possible la manière réelle dont sont codées les applications.

Notre questionnement a pour but d'apporter un éclairage quant aux choix des technologies, et ce, dans une perspective de réduction de nos usages énergétiques.

2.0.1 Précaution de lecture

En aucun cas, nous ne souhaitons démontrer la supériorité d'un langage ou d'un framework sur un autre. En tant qu'acteurs du monde du développement, nous sommes conscients que chaque technologie possède ses qualités propres.

L'objectif ici est seulement de fournir un éclairage sur la consommation énergétique.

Évidemment, chaque usage peut apporter une consommation différente, et nous ne prétendons pas non plus que l'usage que nous avons choisi comme étant la tâche de référence soit pertinent dans tous les cas d'usage. L'attention du lecteur est donc requise dans la lecture et l'analyse de ces résultats.

3 Périmètre de notre analyse

Nous avons pris le parti d'analyser uniquement des frameworks, il s'agit de l'usage majoritaire des serveurs actuellement[1].

Pour cet usage, nous avons choisi les frameworks classiques du marché afin de les comparer sur un pied d'égalité. [2]

Voici les configurations que nous avons choisi de tester. Afin d'être le plus juste possible dans nos comparaisons, nous ne mettons aucun paramétrage spécifique sur les serveurs applicatifs autres que les paramètres par défaut. Cela implique que nous ne réalisons pas de tuning, que ce soit de la RAM, des pools de connexion ou autre, nous utilisons directement la configuration par défaut.

Nous sommes conscients du biais de cette approche, mais nous souhaitons analyser l'existant pour un développement standard et non optimisé. Notre expérience d'équipe de développeurs nous montre que la majorité des applications ne sont pas optimisées, nous voulions donc reproduire cette réalité.

Numéro	Langage	Framework
1	Java	Spring Boot
2	C sharp	Core framework
3	PHP	Symfony
4	Rust	Rocket
5	JavaScript	Express

3.1 Méthodologie

Nous définissons arbitrairement une tâche comme étant le traitement de x requêtes. Chaque requête au serveur consiste en un ensemble de traitements :

- désérialisation d'une requête HTTP (données en JSON)
- insertion dans une base de données du contenu de la requête
- lecture d'une information dans la base de données (entrainant plusieurs requêtes de lecture) avec un ORM [3]
- calcul sur cette donnée
- formatage de la donnée en JSON
- retour de la donnée dans la requête HTTP

[1] Dans l'usage de création d'applications web ou/et mobile

[2] Évidemment, il nous est matériellement impossible de tester toutes les technologies et combinaisons entre elles, nous nous sommes donc limités à ce qui nous paraissait couvrir une large part de marché.

[3] Hibernate (java), Doctrine (php), Diesel (rust), Sequilize (js), Entity Framework (c)

Il s'agit du traitement classique d'une API.

Nous avons pris le parti de ne pas faire de calcul complexe, car l'usage majoritaire des APIs est la lecture et l'écriture en base de données.

La base de données est la même dans tous les tests, et elle est hébergée sur le serveur. Ceci est discutable d'un point de vue architectural, mais nous souhaitions éviter des aléas de réseau entre le serveur et la base de données.

3.1.1 Préchauffage

Notre analyse ne prend pas en compte le temps de démarrage des serveurs applicatifs, ni le temps de rodage. Avant chacun de nos tirs de performances, nous avons préchauffé les applications avec des tirs à blanc. Ce protocole exclut donc d'en tirer des analyses pour un contexte d'exécution serverless.[4]

3.1.2 Protocole de mesure

Nous avons décidé de mesurer la consommation énergétique mesurée en watt directement sur la prise de courant du serveur. La référence du wattmètre : [**plug**].

Notre objectif est de limiter au maximum les aléas de mesure "logiciel". Il nous a donc paru pertinent d'effectuer nos mesures de la manière la plus "pure".

Lors de nos tests, nous notons la consommation en watt pendant toute la durée du test, ce qui nous permet d'obtenir des Wh pour le traitement d'une tâche.

Nous lançons la mesure au début de la tâche jusqu'à la fin, ce qui nous donne la durée de la tâche et sa consommation.

- Soit *ta* la tâche à effectuer.

- Soit *t* le temps pris au serveur pour répondre à l'intégralité de la tâche en seconde.

- Soit *cb* la consommation de base en w du serveur.

- Soit *c* la consommation à l'instant t du serveur.

- Soit *ctt* la consommation en watt * seconde consommée pour effectuer la tâche.

- Soit *cbt* la consommation de base en watt * seconde consommée par le serveur sur la durée de la tâche.

On a de suite : [ctt = t * c].

[4]Serverless : dans un contexte d'exécution sans serveur

Dans un premier temps, nous allons étudier *ctt* en fonction des différentes technologies. On pourrait se demander s'il est pertinent de soustraire la consommation de base *cb* du serveur, car celle-ci est supposée être la même pour chaque technologie. Nous avons décidé en première analyse de le garder, afin de pouvoir voir la consommation *ctt* avec la *cbt* et de voir si l'impact des technologies est faible et non significatif avec la consommation de base *cb*.

3.1.3 Injecteur

Afin d'injecter facilement des requêtes, nous utilisons l'outil JMeter [**jmeter**]. Le script de test est paramétrable pour jouer sur le ratio de requête de modification et de requête de lecture. Afin de minimiser l'effet d'impact de l'injecteur sur les mesures, l'injecteur possède sa propre machine et celle-ci est directement branchée en Ethernet sur le serveur.

3.1.4 Mesure CPU et RAM

Les mesures sont effectuées sur les serveurs avec le logiciel Glances. [**glance**]

3.1.5 Serveur

Nous avons utilisé un serveur lame classique pour effectuer nos tests. Nous n'avons volontairement pas choisi les dernières versions afin d'être au plus proche du parc machine actuel. Le système d'exploitation du serveur est un Debian 11 Bullseye. [**debian**]

3.1.6 Origine de la consommation énergétique

Le but de notre article n'est pas de chercher l'explication de la consommation énergétique. Néanmoins, nous avons pris le soin de mesurer deux indicateurs directement sur la machine. Le taux d'usage du CPU et le taux d'usage de la RAM. L'idée est de voir si une "corrélation" existe entre l'usage du CPU, celui de la RAM et la consommation énergétique. Ces mesures sont réalisées avec l'outil. [**glance**]

3.1.7 Description de la tâche à effectuer

La tâche est un ensemble de x requêtes lancées par y threads en parallèle. Afin de simuler un usage plus réaliste, nous avons défini un ratio de requêtes effectuant des écritures en BDD et un ratio pour des requêtes de lecture simple.

Nous avons imaginé une base de données un minimum complexe, afin de se rapprocher le plus possible des cas que l'on peut croiser dans un projet. La base a également été remplie avec des données générées aléatoirement, ce qui nous permet au final d'avoir un modèle de données réaliste, avec un nombre de données qui l'est tout autant.

Le modèle utilisé est le suivant : des chantiers, qui ont des journaux, qui eux-mêmes ont des dépenses, qui ont chacune un ouvrier. Les chantiers ont également des ouvriers, qui ont chacun un user. Lors de la génération de données, nous avons généré de telle sorte qu'on ait 1000 chantiers, avec chacun 10 journaux, et chaque journal ait 10 dépenses. Chaque chantier a également 10 ouvriers. Pour résumer, il y a donc dans la base 1000 chantiers, 10 000 journaux, 100 000 dépenses, 10 000 ouvriers et 1 000 users.

Concernant les requêtes utilisées dans la tâche, il y en a 2 différentes : une de lecture et une d'écriture. La requête de lecture va simplement récupérer un chantier de manière aléatoire, avec toutes ses relations en cascade. Chaque chantier récupère donc également 10 journaux, 100 dépenses, 10 ouvriers et 10 users. Compte tenu de la consistance des données préalablement générées, on arrive à environ 18 Ko de données par requête. La requête d'écriture est plus simple, on envoie simplement une requête SQL d'update sur la table chantier, afin de modifier tous les champs simples et non relationnels.

4 Résultats

4.1 Résultats bruts

Voici les résultats de chacun des tests effectués sur le serveur. Description des colonnes :

- La colonne Temps mesure le temps en seconde pris par le test pour s'exécuter jusqu'à la fin de la tâche.
- La colonne Total (Wh) est le nombre de Wh mesuré par la sonde pendant toute la durée de l'exécution de la tâche.
- La colonne Traitement (Wh) est le nombre de Wh applicables pour le traitement en soustrayant les Wh théoriques du serveur pendant la durée d'exécution de la tâche. Il permet de se rendre compte de la consommation réellement imputable de la tâche en enlevant le bruit de fond de la consommation serveur.

- La colonne % plus consommateur permet de se rendre compte en comparant les résultats entre eux de savoir l'écart en % de performance. Le plus efficient est donc toujours marqué 100.

Table 1: 100 concurrents 100 itérations = 10 000 requêtes, ratio lecture/écriture = 2

Technologie	Temps (s)	Total (wh)	Traitement (wh)	% plus consommateur
JS/Express	325	3.06	1.98	3478
JAVA/SpringBoot	37	0.34	0.22	382
C/DotnetCore	51	0.44	0.27	478
PHP/Symfony	107	0.98	0.65	1099
RUST/Rocket	10	0.09	0.06	100

Table 2: 100 concurrents 500 itérations = 50 000 requêtes, ratio lecture/écriture = 2

Technologie	Temps (s)	Total (wh)	Traitement (wh)	% plus consommateur
JS/Express	1608	15.12	10.19	3179
JAVA/SpringBoot	190	1.74	1.11	361
C/DotnetCore	236	2.04	1.25	411
PHP/Symfony	530	4.95	3.32	1037
RUST/Rocket	52	0.48	0.32	100

Table 3: 200 concurrents 100 itérations = 20 000 requêtes, ratio lecture/écriture = 2

Technologie	Temps (s)	Total (wh)	Traitement (wh)	% plus consommateur
JS/Express	1059	7.39	3.86	3298
JAVA/SpringBoot	75	0.7	0.45	374
C/DotnetCore	91	0.79	0.49	407
PHP/Symfony	213	1.97	1.32	1048
RUST/Rocket	21	0.19	0.13	100

4.2 Analyse des résultats

On constate de prime abord que tous les frameworks n'ont pas mis le même temps pour traiter la tâche. Sur la tâche la plus longue, on constate même

des écarts assez importants. On constate aussi des écarts de consommation énergétique pour le traitement de la même charge.

4.2.1 Analyses 100 concurrents 100 itérations

Par exemple, pour traiter la tâche de 100 requêtes par 100 utilisateurs, on obtient en mesure 0.97 Wh en JS/Express, 0.44 Wh en C/.NET Core, 0.34 Wh en Java/Spring Boot, 0.98 en PHP/Symfony et seulement 0.09 en Rust/Rocket !

Ce qui est tout de suite intéressant est que l'on peut mesurer quelque chose de significatif entre deux technologies en termes d'énergie. Si on soustrait la consommation de base du serveur sur la durée de traitement de la tâche, on obtient toujours un écart significatif. Cette "surconsommation" est la part qui revient de droit à l'usage de la technologie qui en est faite.

L'autre information intéressante est de constater un corrélation presque parfaite entre la consommation et le temps de traitement de la tâche. Ceci peut s'expliquer par l'usage du CPU à 100

4.2.2 Analyses 200 concurrents 100 itérations et 100 concurrents 500 itérations

L'analyse de ces résultats montre sensiblement les mêmes conclusions. Nous avons toujours un très net avantage pour la solution Rust/Rocket puis pour le groupe Java /Spring Boot et C/.NET Core puis enfin pour PHP/Symfony et en dernier JS/Express.

4.3 Discussion

4.3.1 Rapidité et consommation

On peut légitimement se demander si finalement, la consommation ne suit pas le temps de traitement de la tâche. Plus une tâche prend du temps à s'exécuter, plus la consommation de cette tâche augmente. C'est évidement un facteur prépondérant dans la consommation. Mais si on regarde en détail indépendamment du temps de traitement global, on peut remarquer des différences "sensibles". Par exemple, pour une seconde de traitement, Java/Spring Boot va consommer 0,0093 Wh alors que C/.NET Core 0,0086 Wh et JS/Express s'approche de 0,0094 Wh. On en tire donc la conclusion suivante, indépendamment du temps de traitement de la tâche, la manière dont les langages utilisent la machine induit une consommation différente.

Sur la figure 1, on peut clairement voir que l'usage du CPU n'est pas exactement le même pour chacun des langages. Il serait intéressant d'analyser plus en détail ce qui a le plus d'impact sur la consommation énergétique entre les différents composants de la machine et l'usage qui en est fait. (CPU, RAM, disque, réseau...).

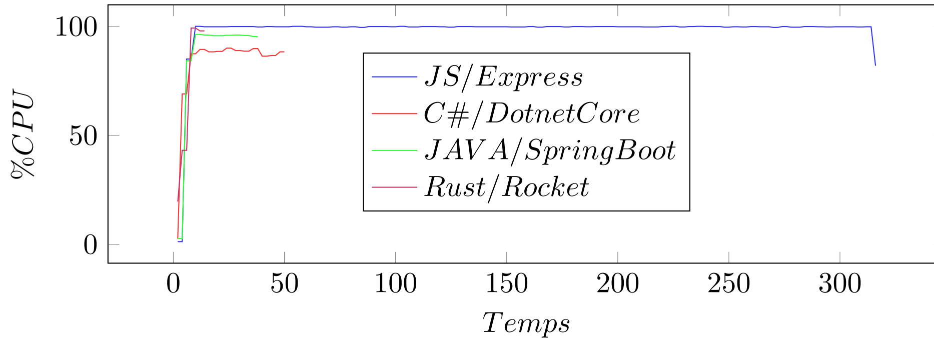

Figure 1: 100c 100i CPU Usage

5 Conclusion

Plusieurs enseignements sont à tirer de cette expérience.

- Nous retrouvons les mêmes classements de technologies que dans l'article de 2017 [**Energy**]. L'ajout d'un framework ne semble pas bouleverser le classement.
- Le temps de traitement semble primordial dans la consommation : plus un traitement est court, moins il consomme.
- Le choix de la technologie est le facteur numéro 1 de la consommation énergétique de l'application.

De ces trois enseignements, nous pouvons tirer la conclusion suivante : Il est primordial de choisir un framework et un langage de technologie si

1) on souhaite obtenir de bonnes performances

2) des consommations énergétiques faibles.

En tant qu'acteurs de développement, nous ne pouvons pas faire l'impasse sur la prise en compte de ces considérations lors du choix des technologies que nous utilisons. Nous sommes les principaux responsables de l'usage qui est fait des serveurs et autres infrastructures qui sont à la base de la consommation des data centers.

6 Travaux futurs discussion

6.0.1 Reproductibilité sur d'autres machines

Il serait intéressant de reproduire ce test avec le même protocole, sur d'autres machines physiques et/ou virtuelles, afin d'observer si les différentes architectures physiques procurent des résultats similaires ou non.

6.0.2 Temps de développement versus temps d'exécution

Face à ces résultats, on pourrait se demander pourquoi les développeurs n'utilisent pas davantage les technologies les plus performantes, aussi bien d'un point de vue de rapidité d'exécution, que de celui de la consommation énergétique. On peut bien sûr imaginer qu'une de ces explications demeure dans la simplicité d'accès des langages. En effet, il est clairement plus facile de démarrer un développement en JavaScript qu'en Rust. Nous pensons également que les communautés établies ont un impact non négligeable dans le choix d'une technologie.

Aujourd'hui, le facteur le plus important du prix d'un développement est, peu ou prou, le coût du nombre de jours nécessaires pour le réaliser. Le temps d'exécution est rarement pris en charge. Quant au coût énergétique, il n'est souvent même pas évoqué. Au vu des enjeux actuels de transition énergétique, il nous paraît primordial de remettre au centre de la table le choix des technologies ainsi que leurs usages.

6.0.3 Perspective serverless

Le serverless est un usage qui tend à se démocratiser. Dans ce cas d'utilisation, le temps d'exécution va certainement prendre un ascendant très fort car il va être le critère de facturation. Et cette perspective peut clairement changer la donne dans l'approche que nous avons dans le choix des technologies.