LOG8371

Ingénierie de la qualité logicielle

Travail Pratique #2

Équipe: Sherbrooke

2002127 – Chebbi, Aymen
1621855 – Gafsi, Ahmed
2089012 – Nde Tsakou, Homer
2091172 – Potvin, David

Remis à : Patrick Loic Foalem

Hiver 2024

Table de matières

Régime d'Assurance Qualité

Introduction

Aujourd'hui, il n'est pas rare que plusieurs développeurs travaillent ensemble pour développer un logiciel et pour ce faire, de nombreux outils sont utilisés pour faciliter la gestion du code source. Deux de ces outils sont Gerrit et Stash. Bien que l'utilisation de ces logiciels permette aux développeurs de créer plus facilement des logiciels, il arrive que du code de mauvaise qualité ou du code contenant des défauts soit introduit sans le vouloir dans le logiciel. Dans ce contexte, le logiciel Sputnik est d'une grande utilité, car c'est un outil permettant d'effectuer une révision statique du code qui provient d'un ensemble de changements proposés pour Gerrit ou Stash. En effet, le logiciel analyse le code source à l'aide de divers outils d'analyse statique du code tel que des linters pour divers langages de programmation ou encore d'autres outils qui permettent de détecter des vulnérabilités et des bogues dans le code. Sputnik a été conçu pour être exécuté après la phase « build » du serveur d'intégration continue (IC) de Jenkins. Ce faisant, la revue du code est effectuée de manière automatisée.

Le logiciel Sputnik est utilisé pour améliorer la qualité du code et réduire la possibilité d'intrusion de défauts dans le code source, alors l'importance de la qualité du projet Sputnik est primordiale, car le logiciel doit pouvoir détecter les problèmes lorsqu'il y en a. Ainsi, il faut que le nombre de faux négatifs soit minimisé. De plus, le fait que l'outil peut être utilisé pour une grande variété de projets allant du développement de jeux vidéo aux développements d'applications aéronautiques où la qualité est cruciale renforce d'autant plus l'importance de la qualité.

Les parties prenantes du logiciel Sputnik sont les développeurs du logiciel (contributeurs du projet qui est sur GitHub), les développeurs qui utilisent Gerrit ou Stash en conjonction avec Sputnik, les ingénieurs en assurance qualité et le gestionnaire de projet.

Le critère de qualité visé par ce rapport est la performance.

La **performance** est un critère de qualité qui décrit la capacité d'un système à exécuter ses tâches ou ses fonctions dans un laps de temps acceptable et avec une efficacité maximale, en utilisant les ressources disponibles de manière optimale.

Critère de qualité

Tableau 1 : Objectif et méthodes de validation des sous-critères de qualité de performance

Critère	Sous critère	Objectifs	Mesures et méthode de validation

Performance	Comportement du temps	Temps de réponse moyen (ISO/IEC FDIS 25023:2015-12 PTb-1-G)** | Réduire le temps de réponse moyen de l'application	Temps de réponse moyen ≤ X secondes, calculé selon la formule : X = ∑ (Bi - Ai) / n, pour i = 1 à n, où Ai = Temps à la réception de l'intrant visé i ; Bi = Temps à la transmission de l'extrant associé i ; n = Nombre de temps de réponse mesurés.
		Convenance du temps de réponse (ISO/IEC FDIS 25023:2015-12 PTb-2-G)	Temps de réponse ≤ X secondes, calculé selon la formule : X = (∑ (Bi - Ai) / n) / C, pour i = 1 à n, où Ai = Temps à la réception de l'intrant visé i ; Bi = Temps à la transmission de l'extrant associé i ; n = Nombre de temps de réponse mesurés ; C = Temps de réponse maximum associé
		Convenance du temps de réponse (ISO/IEC FDIS 25023:2015-12 PTb-2-G)	Temps de réponse ≤ X secondes, calculé selon la formule : X = (∑ (Bi - Ai) / n) / C, pour i = 1 à n, où Ai = Temps à la réception de l'intrant visé i ; Bi = Temps à la transmission de l'extrant associé i ; n = Nombre de temps de réponse mesurés ; C = Temps de réponse maximum associé
	Utilisation des ressources	Utilisation moyenne de la mémoire volatile (ISO/IEC FDIS 25023:2015-12 PRu-2-G) **Minimiser l'utilisation de la mémoire volatile**	au maximum égal à …, calculé selon la formule suivante : X = ∑ (Ai / Bi) / n, où Ai = Espace de mémoire volatile utilisé par le système pour le traitement d'échantillons i ; Bi = Espace de mémoire volatile maximum, pour le traitement d'échantillons i, selon l'exigence suivante du présent document: ... [entrer l'identificateur de l'exigence] ; n = Nombre de traitements d'échantillons. **L'utilisation de la mémoire volatile doit être inférieur ou égal à 50%**
		Utilisation moyenne du CPU (ISO/IEC FDIS 25023:2015-12 PRu-1-G)	au maximum égal à …, calculé selon la formule suivante : X = ∑ (Ai / Bi) / n, où Ai = Utilisation réelle du CPU pour une

		Minimiser l'utilisation du CPU	durée d'observation i ; Bi = Durée d'observation i ; n = Nombre d'observations. **L'utilisation du CPU doit être inférieur ou égal à 50%**
		Utilisation moyenne de la mémoire morte (ISO/IEC FDIS 25023:2015-12 PRu-2-G) **Minimiser l'utilisation de la mémoire morte**	Au maximum égal à …, calculé selon la formule suivante : X = ∑ (Ai / Bi) / n, où Ai = Espace de mémoire morte utilisé par le système pour le traitement d'échantillons i ; Bi = Espace mémoire maximum, pour le traitement d'échantillons i, selon l'exigence suivante du présent document: ... [entrer l'identificateur de l'exigence] ; n = Nombre de traitements d'échantillons. **L'utilisation de la mémoire morte doit être inférieure ou égale à 50%**
	Capacité	Convenance de la capacité d'augmentation d'utilisateurs simultanés	Capacité égale à X, calculée selon la formule : X = (A / B) / C, où A = Nombre de transactions simultanées complétées dans une durée d'observation ; B = Durée d'observation ; C = Capacité minimum de transactions simultanées
		Convenance de la capacité de transactions simultanées	Capacité égale à X, calculée selon la formule : X = (A / B) / C, où A = Nombre de transactions simultanées complétées dans une durée d'observation ; B = Durée d'observation ; C = Capacité minimum de transactions simultanées
		Convenance de la capacité d'utilisateurs simultanés	Capacité égale à X, calculée selon la formule : X = (∑ (Ai / n)) / B, pour i = 1 à n, où A = Nombre d'utilisateurs qui peuvent accéder simultanément au système à l'observation i ; n = Nombre d'observations ; B = Capacité minimum d'utilisateurs simultanés

Stratégie de Validation

Afin de valider les objectifs sur le critère de qualité défini dans le plan de qualité et s'assurer du bon fonctionnement du logiciel dans des cas d'utilisations normales, il est important de tester le logiciel de façon rigoureuse avec des tests pertinents. C'est pourquoi l'élaboration d'une stratégie de tests / validation est une nécessité.

Tableau 2 : Stratégie de validation

Critère	Sous-Critère	Objectifs	Tests/Diagnostics	Outils/Utilitaires
Performance	Comportement du temps	Optimiser le temps de réponse de l'application	Profiling avec JProfiler	JProfiler
	Utilisation des ressources	Minimiser l'utilisation CPU et mémoire volatile	Analyse des ressources avec JProfiler	JProfiler
	Capacité	Évaluer la capacité du système à traiter des transactions simultanées	Tests de charge avec JProfiler	JProfiler, JMeter

Cas de tests

Le tableau ci-dessous représente donc les différents cas de tests pour les fonctionnalités clefs de sputnik lié à chacun des sous-critères de qualité définis plus haut.

Tableau 3 : Information sur les cas de tests

Critère	**Sous-critère**	**Description des tests avec JProfiler**
Performance	Comportement du temps	Utiliser JProfiler pour mesurer et analyser le temps de réponse de l'application lors de l'exécution de diverses tâches. Identifier les sections de code qui causent des retards et optimiser ces zones pour améliorer les performances globales.

	Utilisation des ressources	Analyser avec JProfiler l'utilisation du CPU et de la mémoire volatile lors de l'exécution normale et lors de scénarios de charge élevée. Identifier les pics d'utilisation et les fuites de mémoire pour optimiser l'efficacité des ressources.
	Capacité	Effectuer des tests de charge en simulant un grand nombre de transactions simultanées et en observant l'impact sur les performances à l'aide de JProfiler. Utiliser également JMeter pour compléter cette analyse et tester la capacité sous des charges variables.

Rapports des tests

Pour l'ensemble des tests de profiling réalisés à l'aide de Jprofiler, il est important de savoir comment le configurer pour Sputnik.

Configuration de Jprofiler

Dans l'étape d'installation, appuyez sur « integrate » avec intellij IDEA.

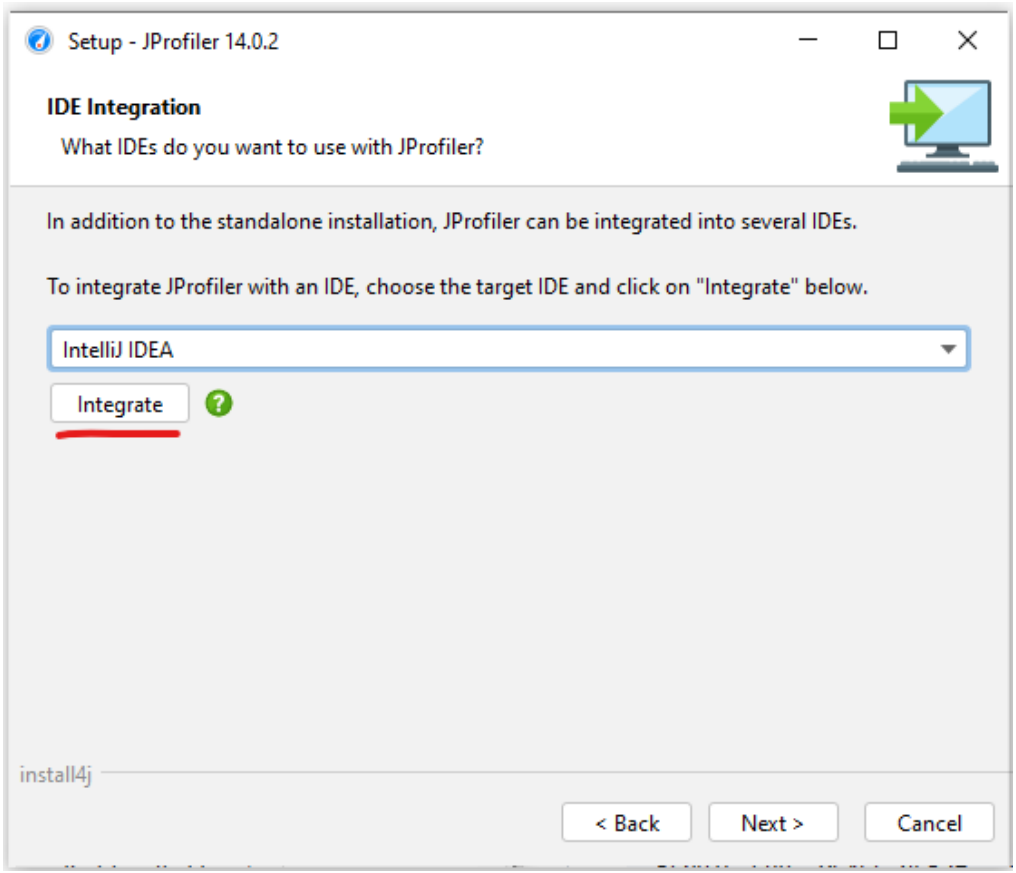

Figure 1. Intégration de Jprofiler avec Intellij IDEA

Suivez les étapes indiquées dans cette étape:

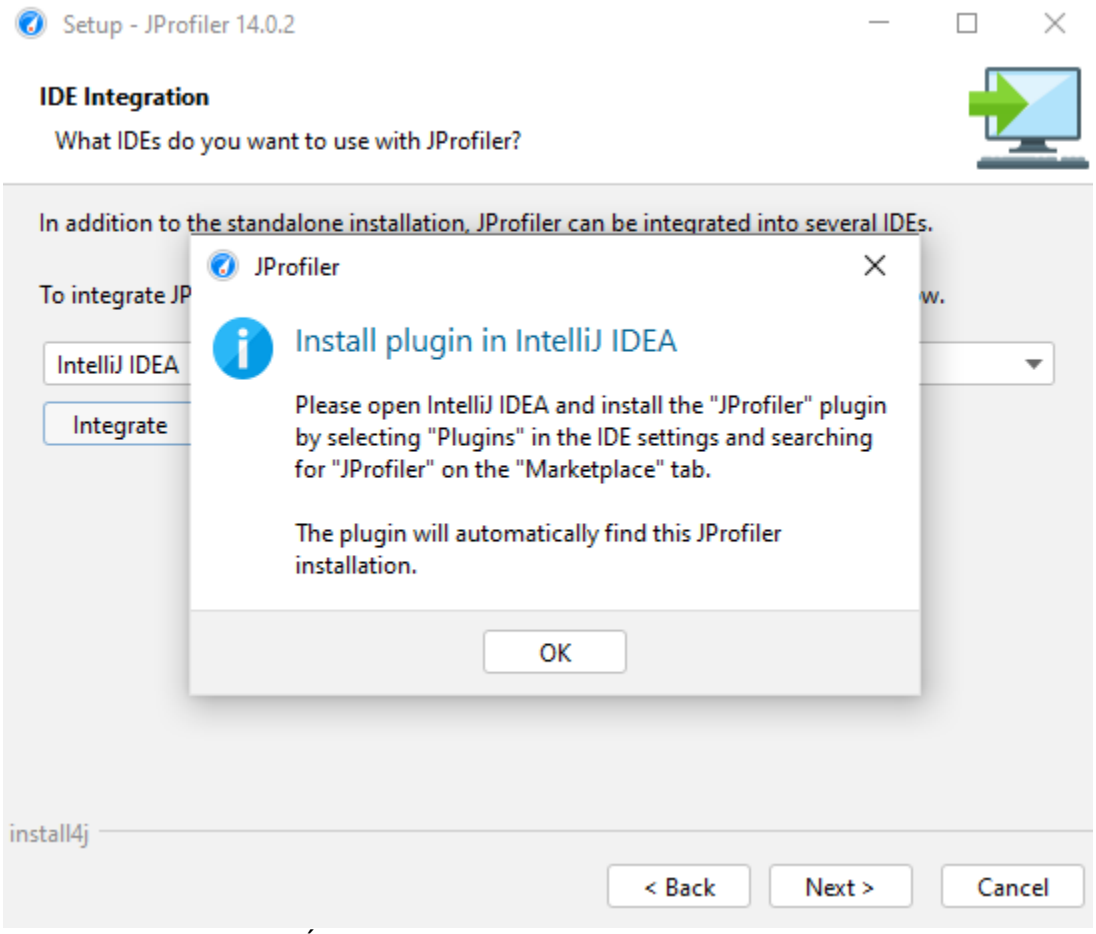

Figure 2. Étape d'intégration de Jprofiler avec Intellij IDEA

Allez dans « file », « settings », « plugins », et recherchez « Jprofiler ». Puis appuyez sur « install ».

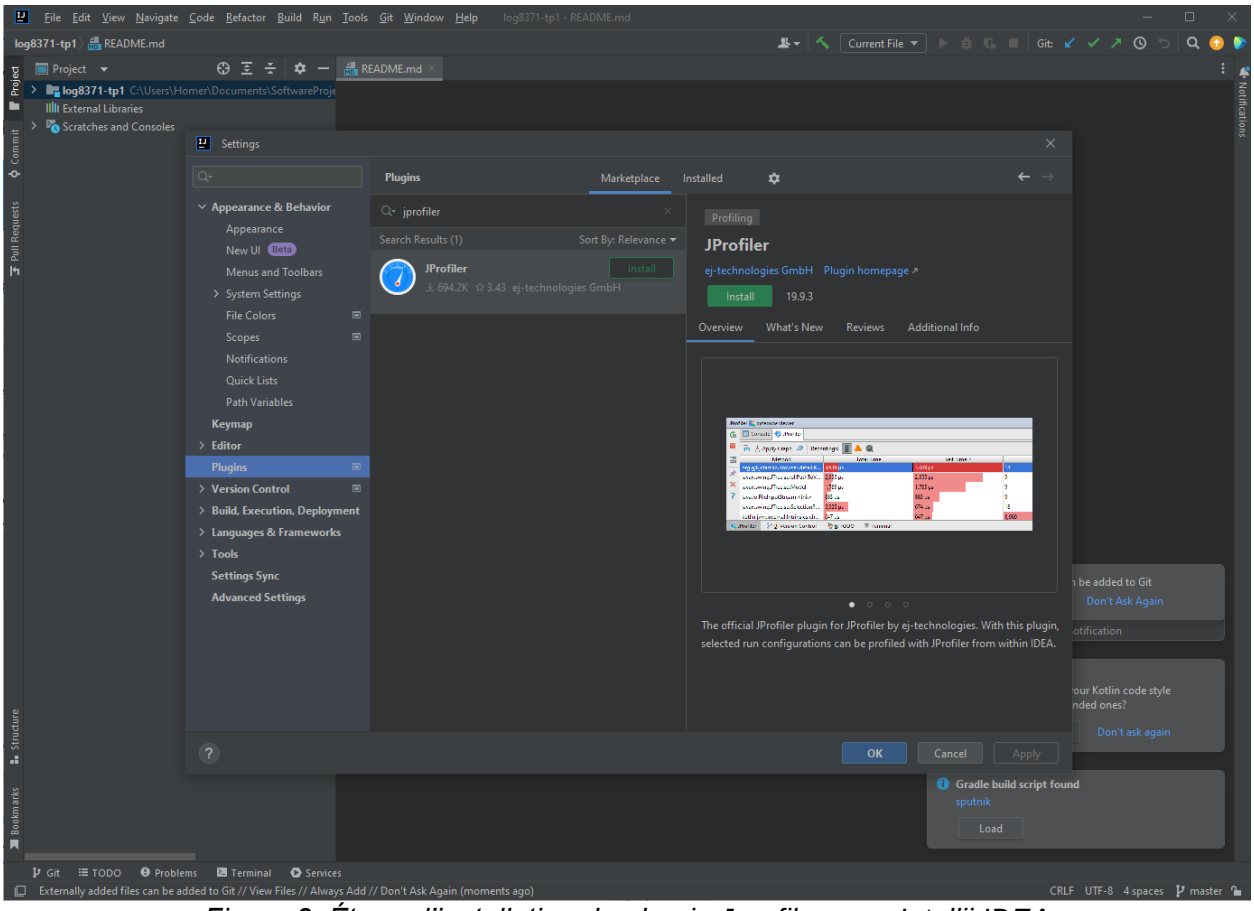

Figure 3. Étape d'installation du plug-in Jprofiler pour Intellij IDEA

À partir d'Intellij IDEA, une fois le plug-in installé, appuyez sur l'icône de Jprofiler pour démarrer le profiling de ce que vous voulez tester.

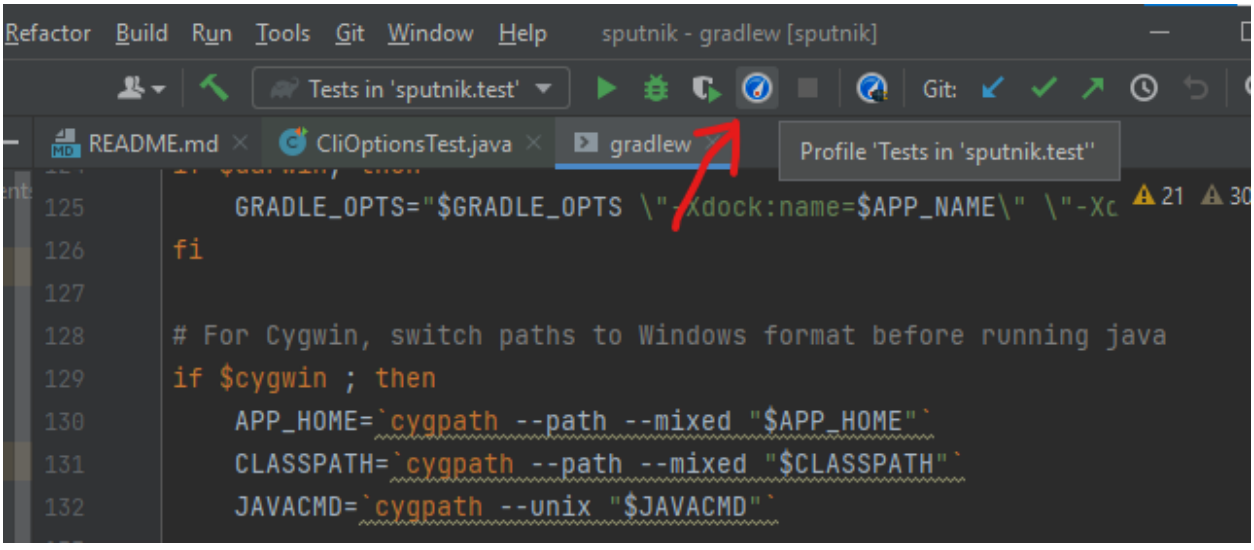

Figure 4. Exécution du profiling d'un projet

Lorsque Jprofiler s'ouvre lors de l'exécution du profiling, choisissez l'option « sampling ».

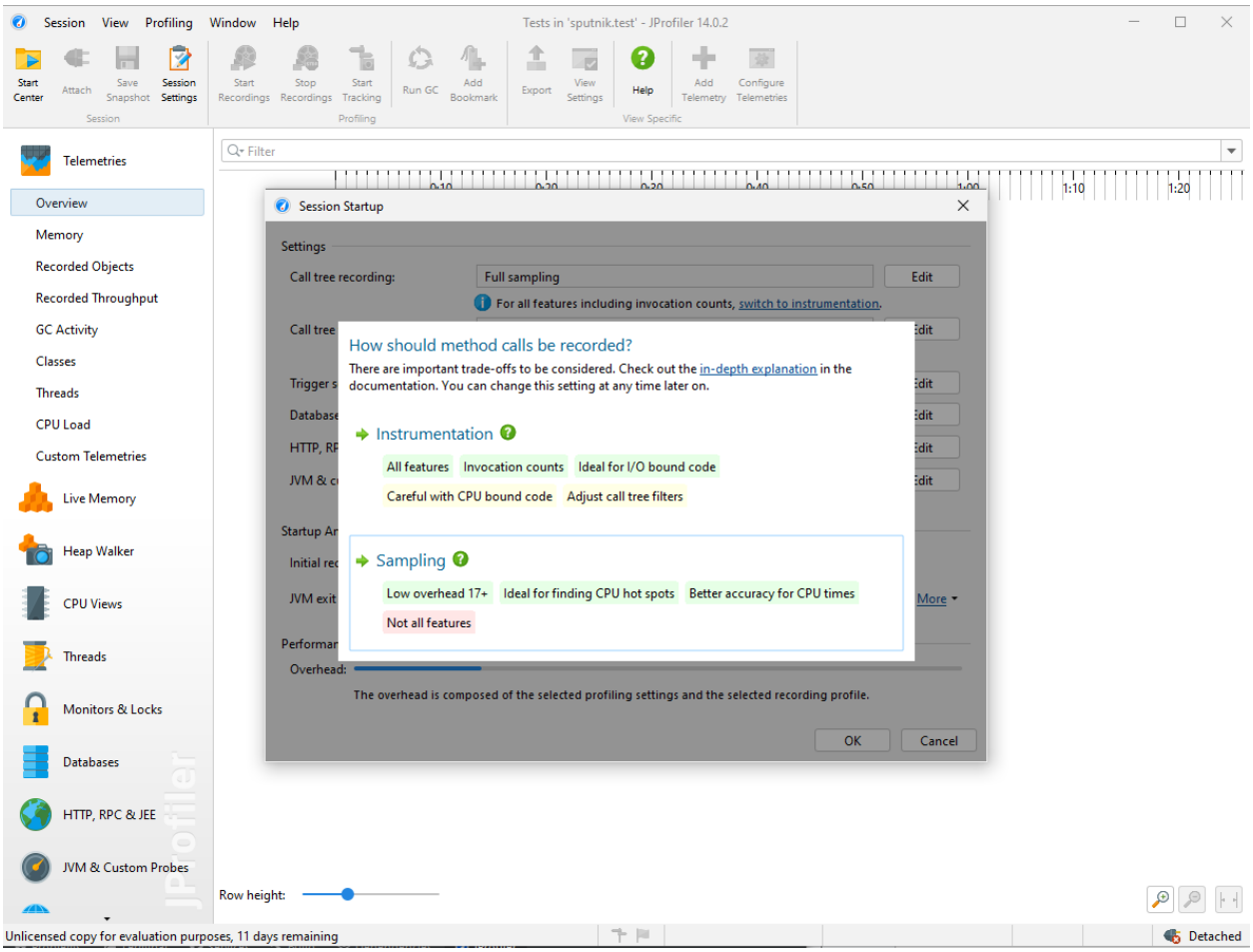

Figure 5. Option de profiling avec Jprofiler

Laissez les paramètres par défaut et le profiling pourra commencer.

Rapport du profiling

Résumé de la configuration

- Version de Java: 15
- Version de jacoco: 0.8.7
- Gradle: 6.7.1
- Outils utilisés: IntelliJ

Scénarios suivis

Notre stratégie pour les scénarios et de suivre dans un premier temps un scénario ayant pour but de réaliser le profiling de Sputnik en général: On test tous les modules en même temps avec Jprofiler pour avoir un snapshot général. Puis, un scénario où l'on teste chaque module séparément afin de comprendre son impact sur le profil général.

Profiling de toutes les fonctionnalités de Sputnik via les tests et Jprofiler

Pour ce scénario, nous avons capturé le profil qui suit en lançant tous les tests unitaires depuis IntelliJ avec JUnit.

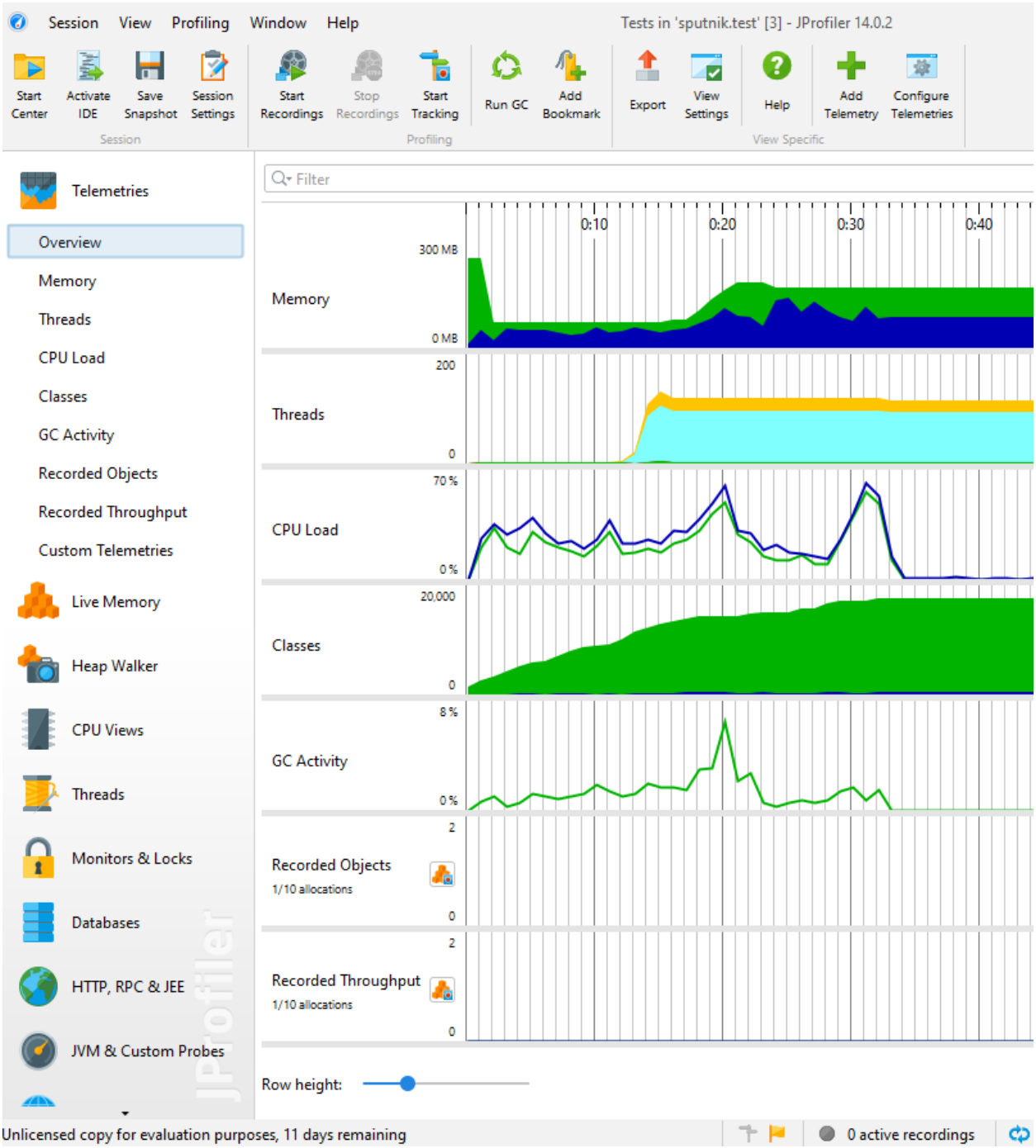

Figure 6. Résultat général

Nous observons à première vue une corrélation évidente entre certains pics d'activité du « Garbage Collector », d'usage mémoire et de « load CPU ». Depuis ce référentiel global, nous pouvons déjà souligner quelques éléments étranges:

- Le nombre de classes semble uniquement augmenter tout au long de l'exécution des tests, ce qui est corrélé avec l'utilisation de la mémoire qui ne diminue pas hormis après le premier pic.
- À partir de 0:15, le nombre de threads utilisé connait un grand pic puis ne semble presque plus évolué.

Analyse de la consommation de la mémoire vive:

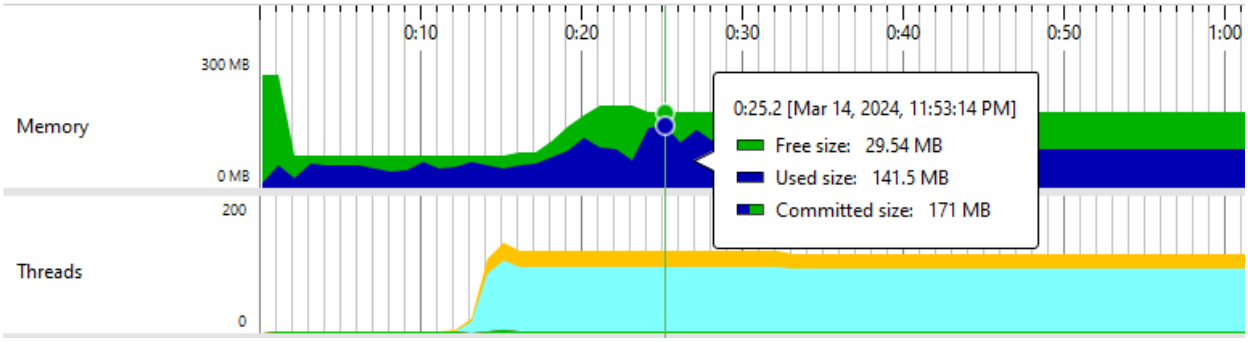

Figure 7. Graphique pour la mémoire vive

Quand nous examinons de manière attentive le graphique de l'usage de la mémoire vive, nous remarquons que l'on a tendance à réserver beaucoup plus de mémoire que nécessaire à certains moments:

- Au tout début de l'exécution
- À 0:20 secondes
- À partir de 0:30 secondes

Au tout début, le pic de mémoire réservée en trop est rapidement diminué une fois que les besoins réels en temps d'exécution sont connus.
Le deuxième pic de mémoire réservé est justifiable par le changement de tendance de la mémoire utilisée. En effet, cette dernière était en hausse constante pendant quelques secondes, puis a chuté brusquement.
Cependant, à partir de 0:30 le double de la mémoire vive utilisé est réservé tout au long du reste de l'exécution des tests malgré que la mémoire utilisée soit constante.
Cette gestion de la mémoire semble peu adéquate, il faudra analyser plus en détail l'impact individuel des modules sur cette dernière.

Analyse de l’usage des threads:

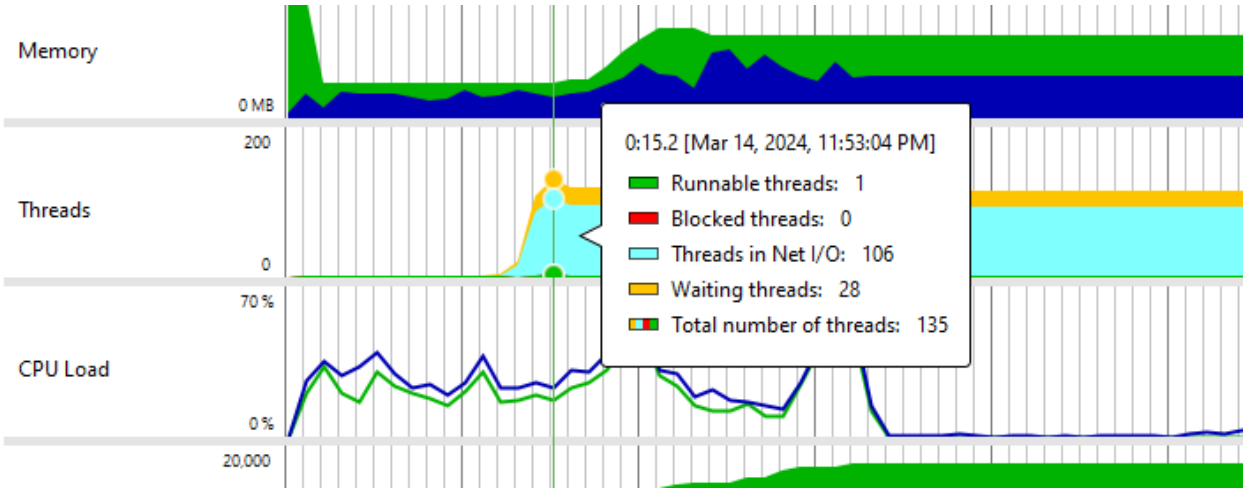

Figure 8. Graphique pour les threads

En ce qui concerne l'usage des threads, le nombre important de threads Net I/O utilisés dans Sputnik semble disproportionné, ce qui peut indiquer des problèmes de conception ou de performance. Ces threads sont dédiés à la gestion des opérations réseau telles que la réception et l'envoi de données. Pour remédier à cette situation, une analyse approfondie de l'architecture de l'application est nécessaire, suivie de possibles ajustements pour optimiser la gestion des threads et réduire la surutilisation des ressources système.
L’analyse des profils individuels des modules qui suivra s’intéressera particulièrement à cet aspect.

Analyse du CPU load:

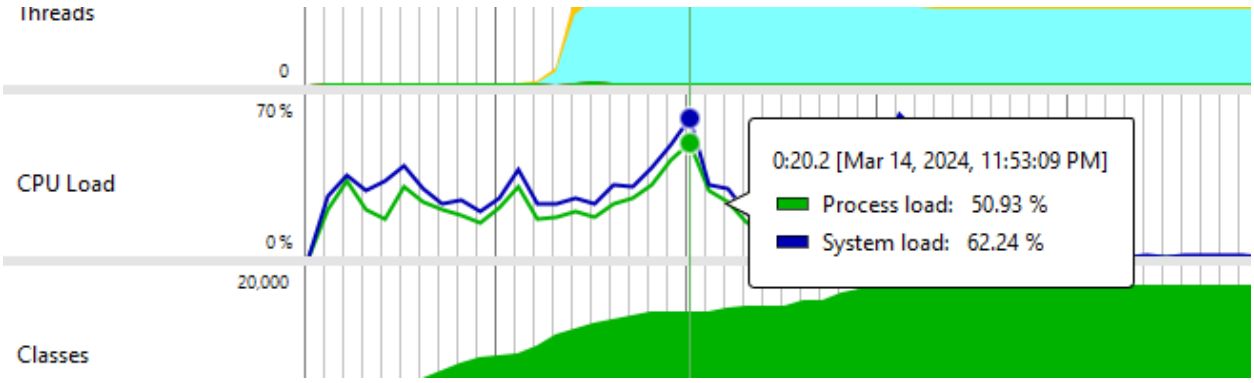

Figure 9. Graphique pour « CPU Load »

Pour le « CPU load », les pics à 0:30 secondes retient particulièrement notre attention, car il est le plus intense et souvient de façon soudaine contrairement à celui de 0:20 qui vient après une augmentation plus graduelle. Comme pour les points d’intérêts mentionnés précédemment, nous essayerons d’isoler les modules responsables pour ce pic afin de faciliter des analyses plus approfondies à l’avenir.

Analyse du nombre de classes:

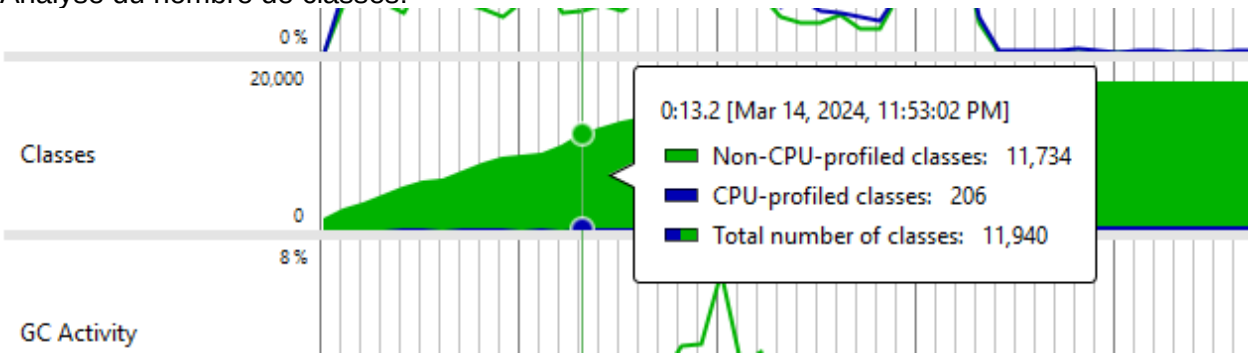

Figure 10. Graphique pour « Classes »

Le nombre de classes augmente de façon linéaire pendant une grande partie de l'exécution avant de plafonner. Ceci pourrait expliquer certains éléments sous-lignés précedemment comme l'usage inadéquat de la mémoire.

Analyse de l'activité du « Garbage Collector »

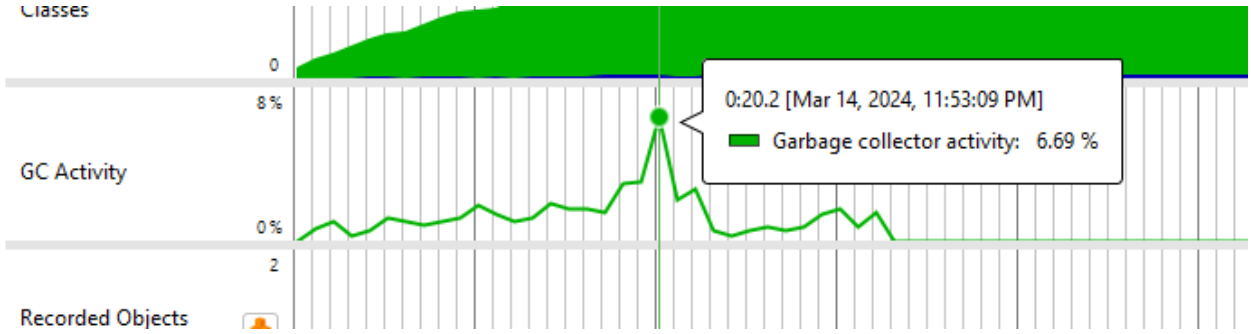

Figure 11. Graphique de « GC Activity » (« Garbage Collector »)

Nous remarquons que l'activité du « Garbage Collector » est assez minime tout au long de l'exécution, hormis un pic important à 0:20. Le nombre de classes actives qui ne diminue pas tout au long de l'exécution ainsi que la mémoire utilisée et réservée à partir de 0:30 nous amène à recommander une investigation approfondie de l'activité du « Garbage Collector » pour s'assurer que ce dernier est en mesure de libérer la mémoire réservée lorsqu'elle n'est plus nécessaire.

Conclusion du premier scénario:

Au long de cette analyse, nous avons souligné plusieurs éléments et enjeux importants. Selon notre analyse, nous recommandons une analyse approfondie des éléments suivants:

- Le pic de « CPU load » à 0:30 qui n'est corrélé à d'autres pics de ressources
- Le nombre important de threads Net I/O
- La gestion de la mémoire et des classes

Profiling de différents modules/fonctionnalités de Sputnik.

Le deuxième scénario aura pour vocation de fournir des analyses de l'impact individuel de chaque module ainsi que des recommandations pour les analyses approfondies à effectuer.

Les résultats de ce scénario seront dans un premier temps exposé individuellement avec quelques points importants explicités sous forme de liste à puce. Par la suite, une analyse sur les liens avec les observations du premier scénario sera réalisée, suivie de nos recommandations pour la suite.

Profiling du module « Configuration »

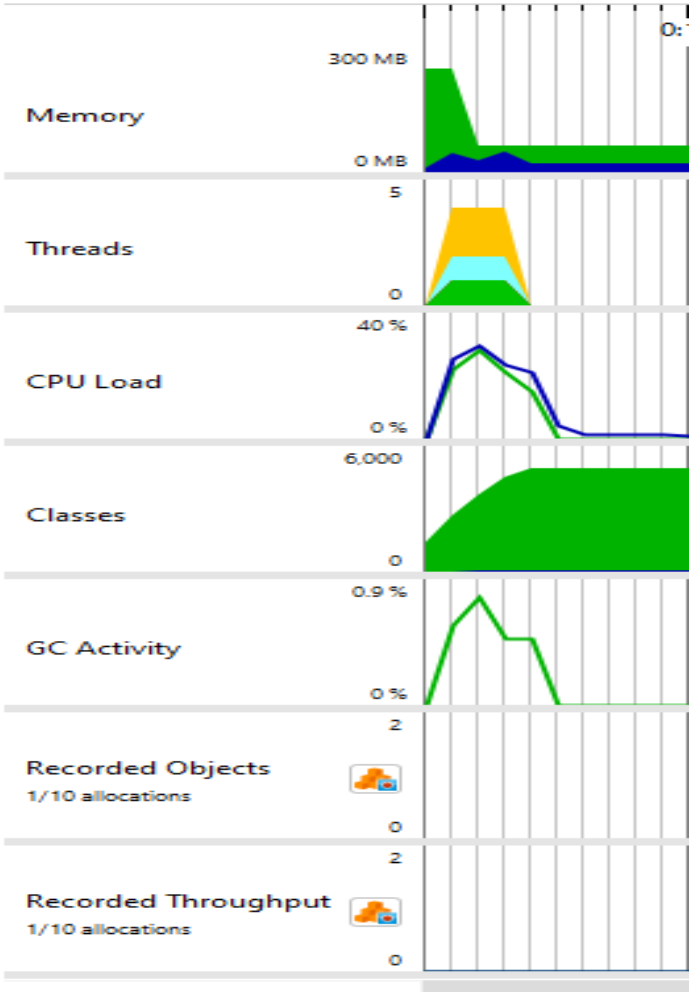

Figure 12. Profile général du module « Configuration »

Le nombre de threads utilisés et le « load CPU » semble raisonnable pour ce module. Nous soulignons néanmoins le nombre important de classes utilisées (environ 6000) qui contraste avec le peu de mémoire requis alors que beaucoup plus de mémoire est réservée que nécessaire.

Profiling du module « Connector »

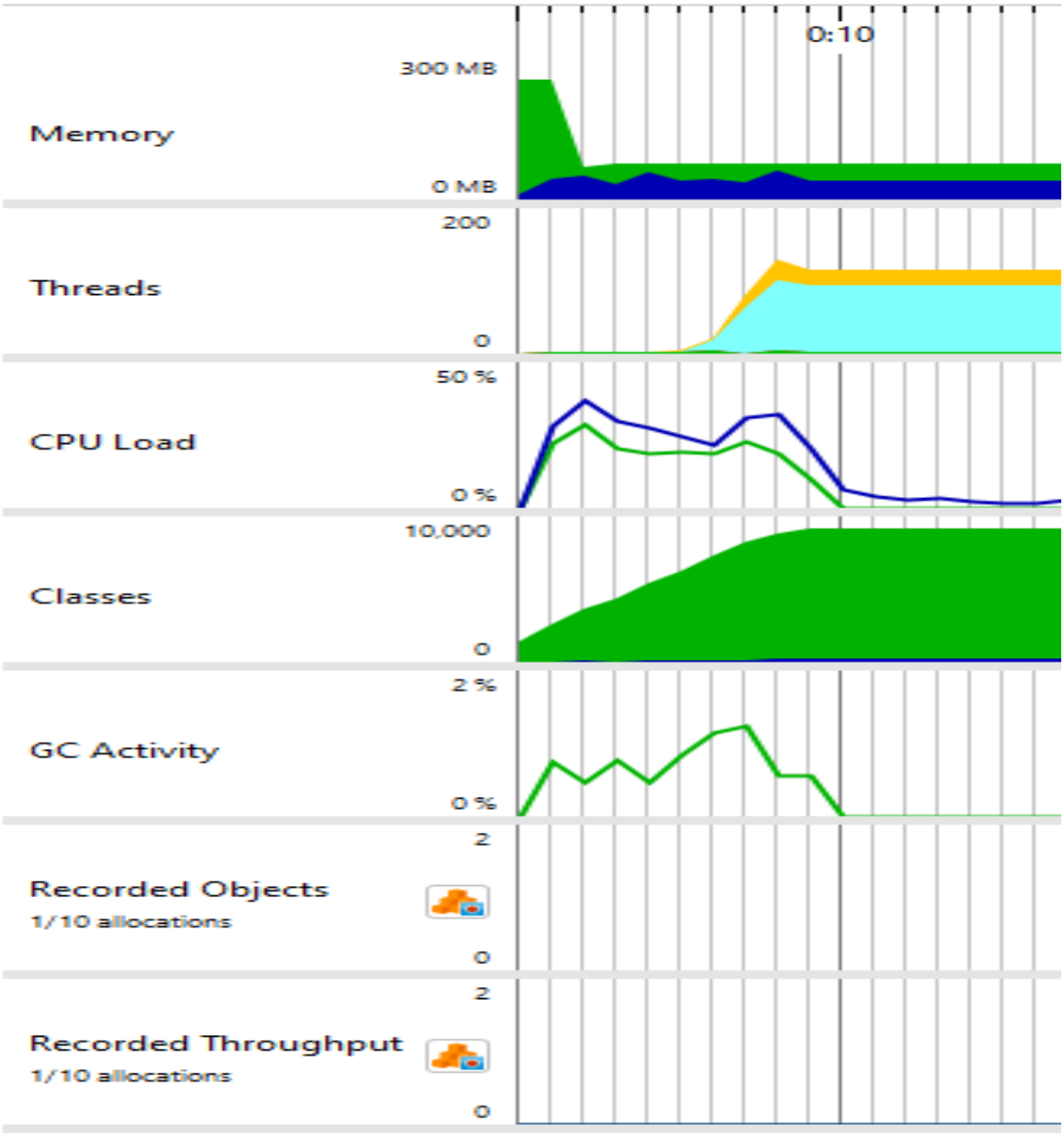

Figure 13. Profile général du module « Connector »

Nous remarquons ici un nombre très important de classes: 10 000 classes utilisées pour ce module uniquement sur les 20 000 utilisées lors du scénario général.
On remarque aussi un nombre important de threads Net I/O, mais cela n'est pas nécessairement aberrant puisqu'il s'agit du module « Connector ».

Profiling de « Engine »

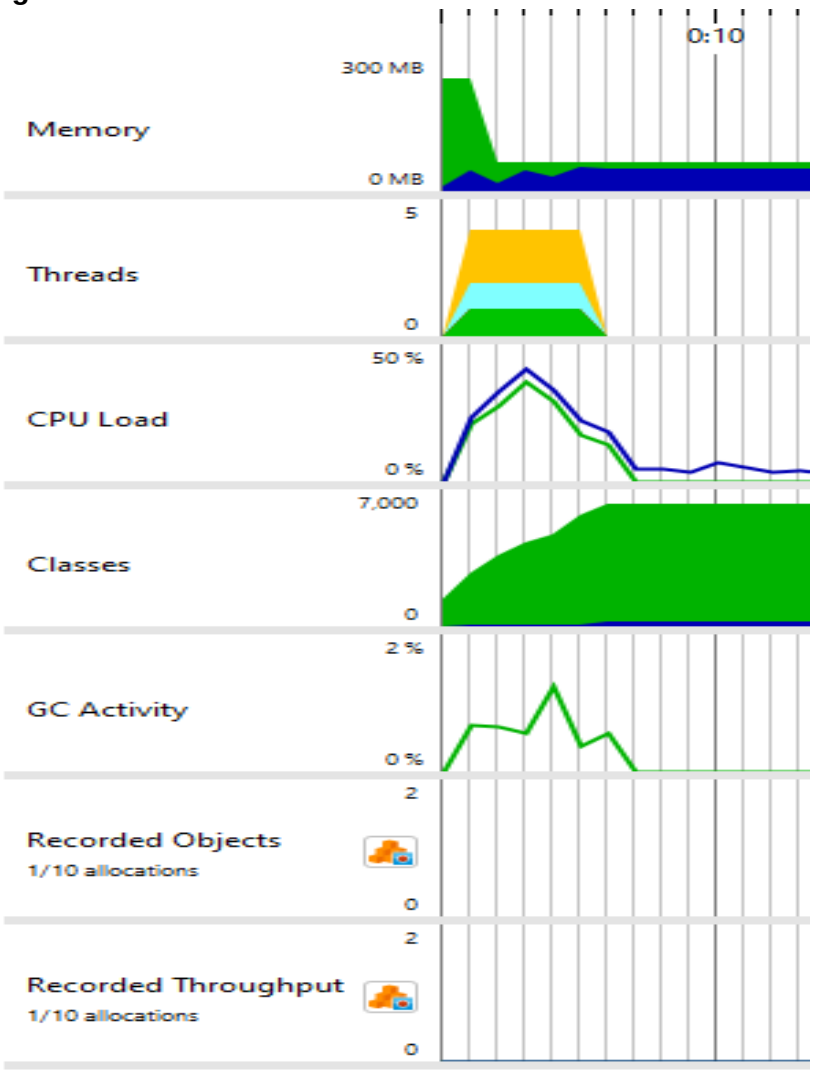

Figure 14. Profile général du module « Engine »

Nous remarquons que ce module a une très bonne gestion de la mémoire réservée qui suit de manière assez proche les besoins réels du module. Nous remarquons aussi l'absence de corrélation entre le nombre élevé de classes et le surplus de mémoire réservée cette fois-ci.

Profiling du module « Processor »

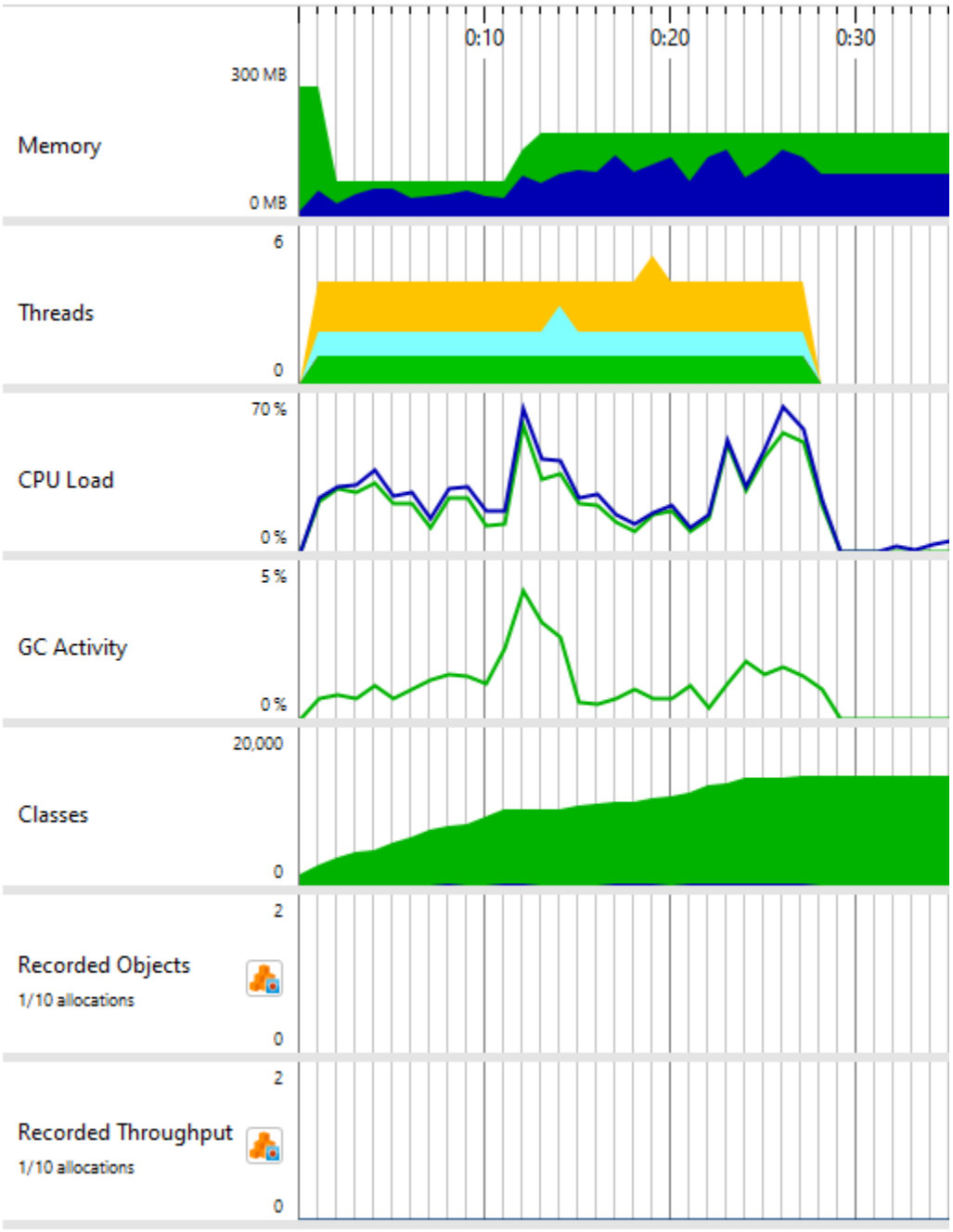

Figure 15. Profile général du module « Processor »

Nous souhaitons souligner que ce module semble être le principal responsable du pic de « load CPU » à 0:30 vu que le pic est présent dans son profil, mais absent de celui des autres modules.
On remarque aussi un usage très élevé de classes (plus de 10 000).

Profiling du module « Review »

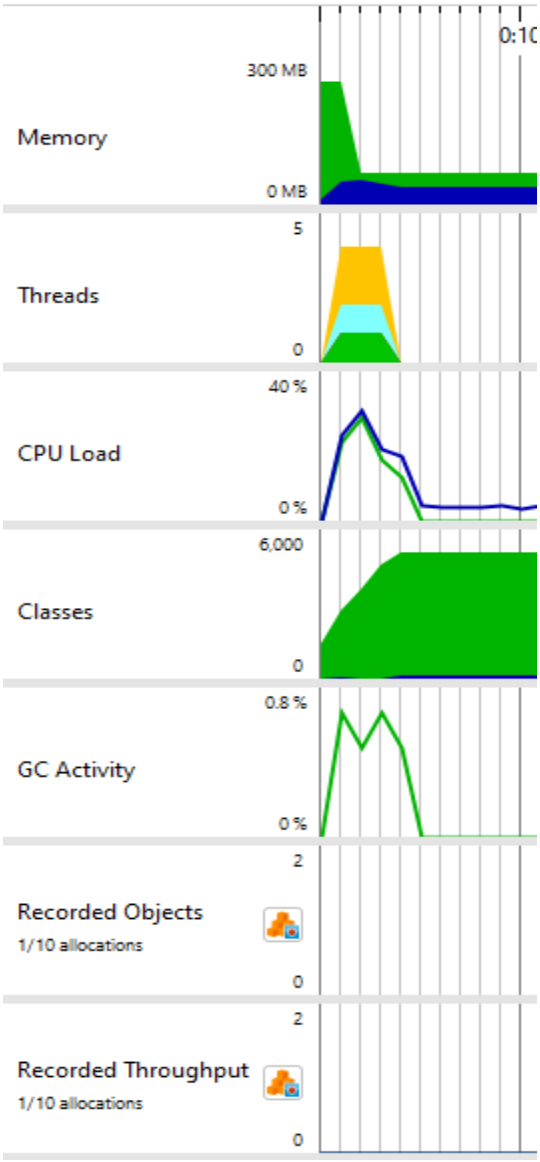

Figure 16. Profile général du module Review

Ce module a une bonne gestion de la mémoire réservée, des threads et du « load CPU ». On encore la présence d'au moins 6000 classes utilisées.

Retour sur les enjeux discutés et recommandations:

Le tableau qui suit expose notre analyse des observations effectuées pour le deuxième scénario ainsi que notre recommandation pour chaque enjeu.

Tableau 4 : Récapitulatif des observations et recommandations

Enjeux	Analyse	Recommendation	Priorité
Mémoire réservée	Lors du deuxième scénario, nous avons observé un contraste entre certains modules qui ne requiert pas plus de mémoire vive que leur usage réel, et d'autres qui requièrent souvent le double ou plus de mémoire réservée que leur usage réel. De plus, nous avons remarqué l'absence de corrélation certaine entre le nombre de classes utilisées et le surplus de mémoire réservée, mais une corrélation significative entre l'activité du « Garbage Collector » et l'accumulation de mémoire vive réservée en trop. Le surplus de mémoire vive réservée le plus significatif a été observé lors du profiling du module Configuration où bien plus que le double nécessaire était alloué.	Effectuer une analyse approfondie pour comparer la cause du surplus de mémoire vive réservée par les modules suivants: « Processor », « Connector » et « Configuration »,	Élevée
« Load CPU »	Le module « Processor » semble responsable du pic important de « load CPU » observé pendant le premier scénario. Ce pic est corrélé avec la libération des threads lors du profiling du module individuellement.	Effectuer une analyse approfondie sur les causes du pic de « load CPU » en lien avec l'usage de la classe « Processor ».	Élevée
Thread Net I/O	L'usage des threads de ce type est majoritairement dû au module « Connector », ce qui a du sens sémantiquement, car ce dernier est responsable des connexions.	Effectuer une analyse des impacts des threads sur la performance.	Faible
Nombre de classes	Lors du deuxième scénario, nous avons observé que tous les modules faisaient l'usage d'au moins environ 6 000 classes.	Effectuer une analyse de l'impact du nombre de classes supplémentaires utilisées par les modules « Processo » et « Engine » sur les	Modérée

	Nous concluons donc que parmi ces dernières il y a majoritairement des classes communes à tous les modules. Cependant, les modules « Processor » et « Engine » utilisent un nombre de classes supérieures à 10 000. Ces derniers sont des modules importants avec des responsabilités claires, donc l'usage de plus de classes uniques n'est pas aberrant, mais mérite quand même d'être étudié.	performances et la mémoire vive	

Plan d'action suggéré

Nous recommandons de prioriser les analyses axées sur le surplus de mémoire réservée et le pic de « load CPU ». Ces derniers éléments ont un impact élevé sur la performance et pourraient s'avérer critiques sur des systèmes moins puissants ou quand les ressources sont en demande forte.