LOG430 - Rapport du laboratoire 02

ÉTS - LOG430 - Architecture logicielle - Hiver 2026 - Groupe 1

Étudiant: Yanni Haddar Nom github: mapleduck repo github: github.com/mapleduck/log430-labo4

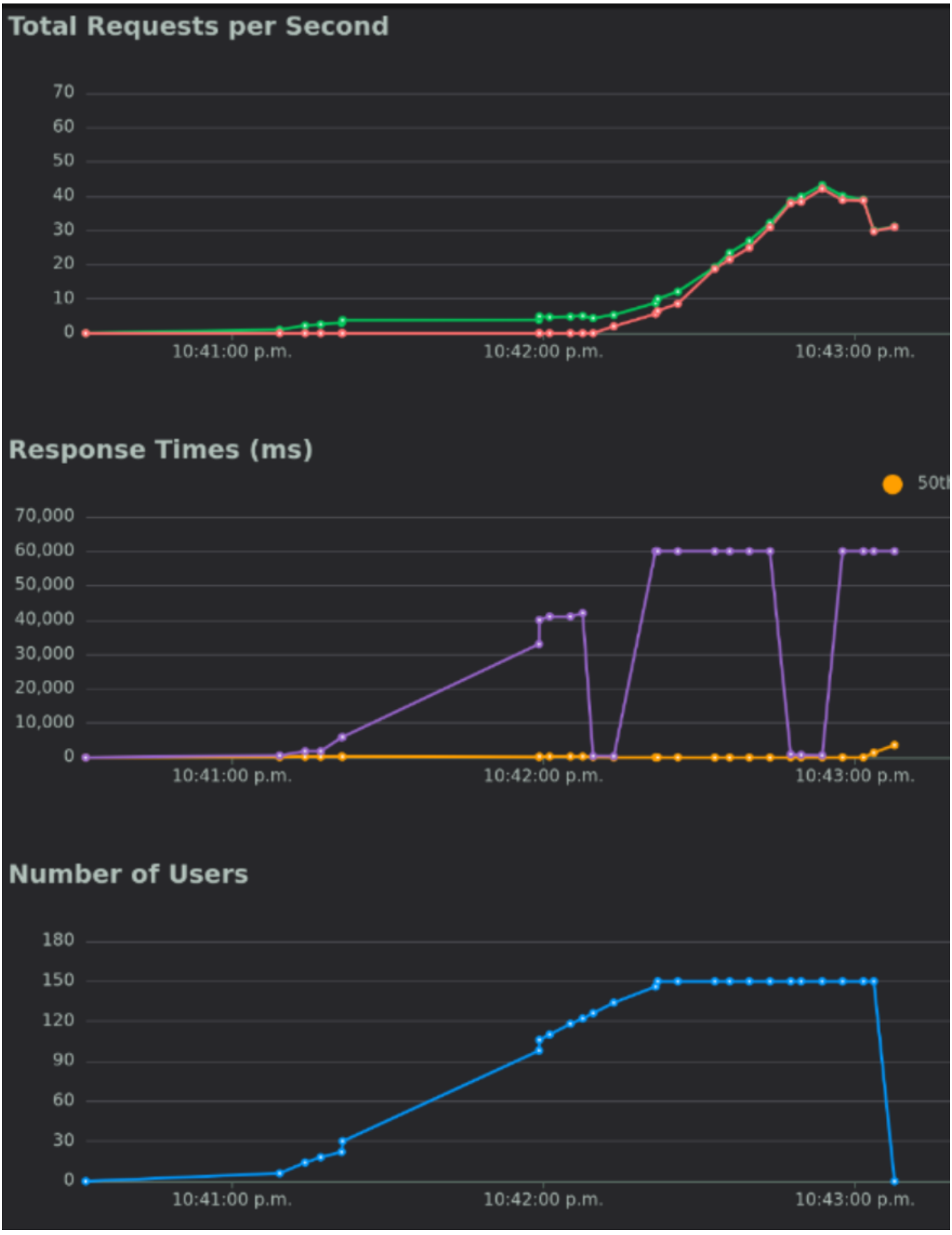
Questions

Note préalabe: des problèmes avec le VPN Cisco sur ma machine Ubuntu m'ont requis de faire ce lab sans accès à ma VM, donc store_manager et les autres containers furent uniquement roulé localement.

Soit mon ordi est moins puissant qu'une des VMs (ce qui serait surprenant selon ses specs), soit il y avait un problème de conflit de ressources, mais à noter qu'il y a un surprenant haut taux d'échec dans les requests.

> 💡 Question 1 : Combien d'utilisateurs faut-il pour que le Store Manager commence à échouer dans votre environnement de test ? Pour répondre à cette question, comparez la ligne Failures et la ligne Users dans les graphiques.

Vers 122 users:



> 💡 Question 2 : Sur l'onglet Statistics, comparez la différence entre les requêtes et les échecs pour tous les endpoints. Combien d'entre eux échouent plus de 50 % du temps ?

Toutes les requêtes échouent bien au dessus de 50%. 81% des requêtes overall échouent:

Type	Name	# Requests	# Fails
POST	/orders	683	661
GET	/orders/reports/best-sellers	590	420
GET	/orders/reports/highest-spenders	585	426
	Aggregated	1858	1507

> 💡 Question 3 : Affichez quelques exemples des messages d'erreur affichés dans l'onglet Failures. Ces messages indiquent une défaillance dans quelle(s) partie(s) du Store Manager ? Par exemple, est-ce que le problème vient du service Python / MySQL / Redis / autre ?

La plupart des éches viennent de Flask qui est overloaded (uniquement les get), mais une bonne patie vient aussi du serveur SQL qui ne peux pas handle tout les write (car les posts sont beaucoup plus couteux que les gets).

# Failures	Method	Name	Message
250	POST	/orders	CatchResponseError('Erreur : 500 - (mysql.connector.errors.OperationalError) 1040 (08004): Too many connections\n(Background on this error at: https://sqlalche.me/e/20/e3q8)')
183	POST	/orders	CatchResponseError('Erreur : 500 - (mysql.connector.errors.DatabaseError) 1040 (HY000): Too many connections\n(Background on this error at: https://sqlalche.me/e/20/4xp6)')
114	POST	/orders	Exception('Aucune réponse (erreur ou timeout)')
114	POST	/orders	RetriesExceeded('http://store_manager:5000/orders', 0, original=timed out)
420	GET	/orders/reports/best-sellers	CatchResponseError('Erreur : 500 - Aucun JSON dans la réponse. Message : <!doctype html>\n<html lang=en>\n<title>500 Internal Server Error</title>\n Internal Server Error \n The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application. \n')
426	GET	/orders/reports/highest-spenders	CatchResponseError('Erreur : 500 - Aucun JSON dans la réponse. Message : <!doctype html>\n<html lang=en>\n<title>500 Internal Server Error</title>\n Internal Server Error \n The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application. \n')

> 💡 Question 4 : Sur l'onglet Statistics, comparez les résultats actuels avec les résultats du test de charge précédent. Est-ce que vous voyez quelques différences dans les métriques pour l'endpoint POST /orders ?

Oui. Pour commencer (pas visible dans le tableau), le système a handle beaucoup plus de requests, passant de 31 à 58 RPS. Les requêtes sont répondues beaucoup, beaucoup plus rapidement (voir toute les stats au

milieu du tableau). Le taux d'échec, lui, n'a pas bougé vraiment, restant à 80%. Mais cela reste une amélioration nette.

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/orders	767	741	20	60000	60000	9335.84	0	60004	103.2	22.6	22.6
GET	/orders/reports/best-sellers	685	481	35	320	440	108.23	6	480	1342.18	17.8	17.4
GET	/orders/reports/highest-spenders	707	510	35	140	220	55.69	8	365	307.87	18.2	18
	Aggregated	2159	1732	29	60000	60000	3369.2	0	60004	563.32	58.6	58

> 💡 Question 5 : Si nous avions plus d'articles dans notre base de données (par exemple, 1 million), ou simplement plus d'articles par commande en moyenne, le temps de réponse de l'endpoint POST /orders augmenterait-il, diminuerait-il ou resterait-il identique ?

Le temps de réponse resterait relativement identique. Même avec 1 million de produits, la recherche d'articles par product_id reste très performante car elle utilise la clé primaire de la table Product, qui est très efficace selon mes recherches. Et grâce à l'optimisation n+1 rajoutée, une requête récupère tout les prix d'un coup.

> 💡 Question 6 : Sur l'onglet Statistics, comparez les résultats actuels avec les résultats du test de charge précédent. Est-ce que vous voyez quelques différences significatives dans les métriques pour les endpoints POST /orders, GET /orders/reports/highest-spenders et GET /orders/reports/best-sellers ? Dans quelle mesure la performance s'est-elle améliorée ou détériorée (par exemple, en pourcentage)?

Énorme amélioration pour les GET (0% failure rate) et temps de réponse divisé par 5. Mais pour les POSTS seulement, aucune amélioration notable.

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/orders	842	830	22	60000	60000	8221.25	0	60006	108.82	27.4	27.4
GET	/orders/reports/best-sellers	762	0	34	64	93	35.9	2	174	387	18.5	0
GET	/orders/reports/highest-spenders	762	0	35	63	90	36.42	3	277	417	19	0
	Aggregated	2366	830	31	770	60000	2949.03	0	60006	297.66	64.9	27.4

> 💡 Question 7 : La génération de rapports repose désormais entièrement sur des requêtes adressées à Redis, ce qui réduit la charge pesant sur MySQL. Cependant, le point de terminaison POST /orders reste à la traîne par rapport aux autres en termes de performances dans notre scénario de test. Alors, qu'est-ce qui limite les performances de l'endpoint POST /orders ?

La performance de POST /orders est limitée par les opérations d'écriture MySQL (il n'est pas sur REDIS) nécessaires pour garantir la persistance des données et la gestion des stocks. Contrairement aux rapports qui lisent un cache pre-calculated dans Redis, chaque commande doit valider et enregistrer plusieurs entries dnas l BD, etant limité par les ressources disques et le serveur MySQL.

> 💡 Question 8 : Sur l'onglet Statistics, comparez les résultats actuels avec les résultats du test de charge précédent. Est-ce que vous voyez quelques différences significatives dans les métriques pour les endpoints POST /orders, GET /orders/reports/highest-spenders et GET /orders/reports/best-sellers ? Dans quelle mesure la performance s'est-elle améliorée ou détériorée (par exemple, en pourcentage) ? La réponse dépendra de votre environnement d'exécution (par exemple, vous obtiendrez de meilleures performances en exécutant 2 instances de Store Manager sur 2 machines virtuelles plutôt que sur une seule).

Il y a une nette dégradation des performanes par rapport au test précédent, sauf dans une métrique: la rapidité de réponse des GET. Le taux d'échec, qui était à 35%, est passé à 59%. Les RPS sont passées de 64 à 53.

Ma théorie est que, en étant sur une seule machine, l'effet de balance est contre-productif, car le nombre de coeurs sont limités et les requêtes se partagent toutes les mêmes ressources pour leur exécution. Il n'y a pas de réel load balancing car tout roule sur la même machine. Il y a juste un risque augmenté de collisions.

Je n'ai aucun doute qu'en ayant deux (ou même un cluster) de machines sur lesquelles il est réellement possible de faire du load balancing, même si ces machines étaient significativement plus faibles que mon laptop, les résultats seraient notablement meilleurs, car nginx est optimisée pour cela, par pour tout rouler sur une seule machine.

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/orders	785	722	9	60000	60000	8172.34	0	60004	135.38	19	19
GET	/orders/reports/best-sellers	808	350	4	21	35	7.07	0	137	288.5	17.1	17.1
GET	/orders/reports/highest-spenders	809	338	5	23	35	7.48	0	140	310.7	17.3	17.3
	Aggregated	2402	1410	5	19000	60000	2675.71	0	60004	245.94	53.4	53.4

> 💡 Question 9 : Dans le fichier nginx.conf, il existe un attribut qui configure l'équilibrage de charge. Quelle politique d'équilibrage de charge utilisons-nous actuellement ? Consultez la documentation officielle de Nginx si vous avez des questions.

`least_conn` dans Upstream est le paramètre utilisé. Selon la doc, cette politique distribue de manière intelligente en envoyant les requêtes au serveur qui a le moins de connexions en cours à cet instant.