

Pi-Injector

Exploitation des résultats

Table des matières

Introduction.....	2
Le problème vient-il du serveur ?.....	2
Réseau.....	2
Applicatif.....	2
Le serveur : unité mise en cause.....	3
Processeur.....	3
Entrées/sorties disque et mémoire RAM.....	4
Aperçu d'un résultat et analyse.....	4

Introduction

L'objectif du projet Pi-Injector est de pouvoir lancer des tests de charge configurables sur des serveurs afin de pouvoir évaluer leurs performances en utilisant comme injecteur des Raspberry Pi. A la fin de l'exécution d'un test, l'application va fournir différents graphiques qui vont permettre d'analyser et de détecter les problèmes existants. Pour cela, il est possible de fournir différents types de scénarios (suite d'instructions) pour pouvoir exécuter un test particulier.

Ce document va fournir de manière non exhaustive quelques explications pour analyser les résultats et quelques pistes pour améliorer les performances du serveur.

Le problème vient-il du serveur ?

C'est une question qu'il faut absolument se poser. Le problème de latence d'une application n'est pas seulement dû à de mauvaises performances du serveur mais peut aussi avoir d'autres sources.

Réseau

Imaginons que notre serveur reçoit une énorme quantité de requêtes (ce que va faire Pi-Injector à l'exécution d'un test). Toutes ces requêtes vont transiter via le réseau. Hors, le réseau a une vitesse limite (octet/seconde) et chaque carte réseau a une vitesse maximum de traitement. Donc si nous recevons un nombre de requêtes important, nous allons saturer le réseau et donc créer un goulot d'étranglement au niveau de la carte réseau. Pour remédier à cela, une solution serait d'ajouter une carte réseau au serveur ou bien de segmenter le réseau.

Applicatif

Les problèmes applicatifs vont concerner le côté affichage qui peut ne pas être optimisé et donc prendre un certain temps à s'afficher et aussi le côté couple serveur/SGBD. L'accès aux données représente un point majeur de l'utilisation des ressources du serveur. Il est possible que le serveur et le SGBD peuvent demander des ressources simultanément ceux qui peut créer des latences.

Une solution possible serait de mettre le serveur et le SGBD sur des machines différentes (avec cette solution, une latence peut être créée entre les deux entités). Du côté SGBD, il est aussi possible que pour chaque requête, un verrou soit créé sur les données, ce qui peut ralentir l'obtention de résultats.

Le serveur : unité mise en cause

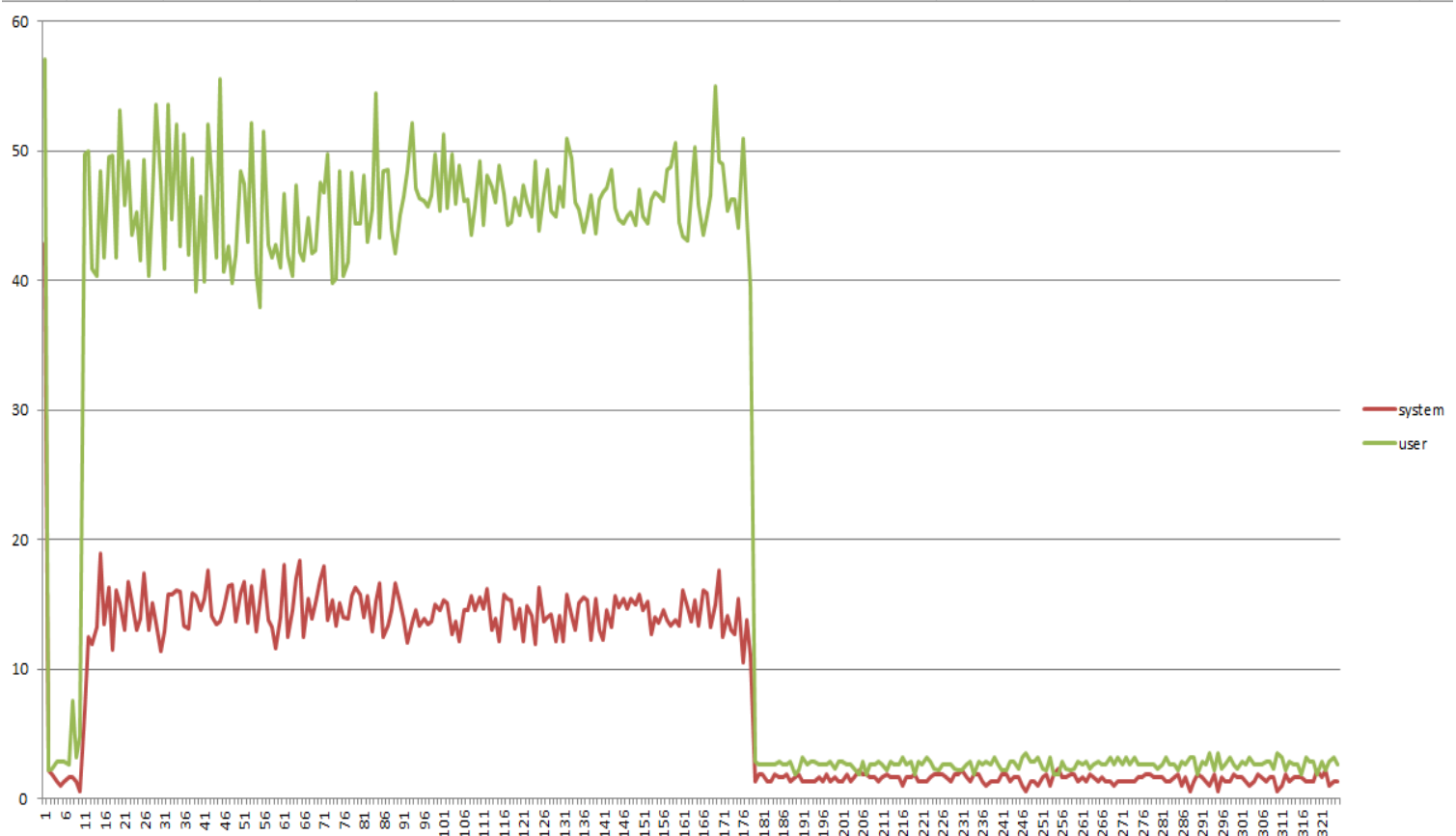
Les paramètres qui sont généralement mis en causes sur les performances du serveur sont le processeur, la mémoire et les entrées sorties (E/S) de disque (désigne le taux de requête sur disque physique pendant un intervalle de temps). Si l'une de ses unités n'est pas adaptées, l'application sera moins performante. Il faudra donc cibler via différent scénario l'unité que l'ont va vouloir tester.

Processeur

Pendant l'utilisation de l'application, le serveur va devoir traiter différentes informations. Pour cela, il va utiliser son processeur. On va pouvoir récupérer les mesures du pourcentage temps processeur grâce à différents outils. Dans notre cas, nous allons utiliser l'outil nommé Glances ([site](#)).

Pendant l'exécution de l'application, nous allons sauvegarder le temps processus pour le système et pour le user. Grâce à cela, nous allons savoir si notre test arrive à mettre le serveur cible au maximum de ses capacités au niveau du processeur.

Voici comme exemple un résultat obtenu :



On peut voir ici sur l'axe des ordonnées le pourcentage du processeur utilisé et sur l'axe des abscisses le temps. La courbe verte représente le pourcentage du processeur utilisé par le user et la courbe rouge représente le pourcentage du processeur utilisé par le système.

Ces courbes représentent le pourcentage de temps que le processeur passe en mode utilisateur et système. Si la valeur pour la courbe user est haute (alors que sans l'exécution de l'application, elle est basse), cela signifie que le serveur est occupé par l'application qui utilise trop les ressources du processeur.

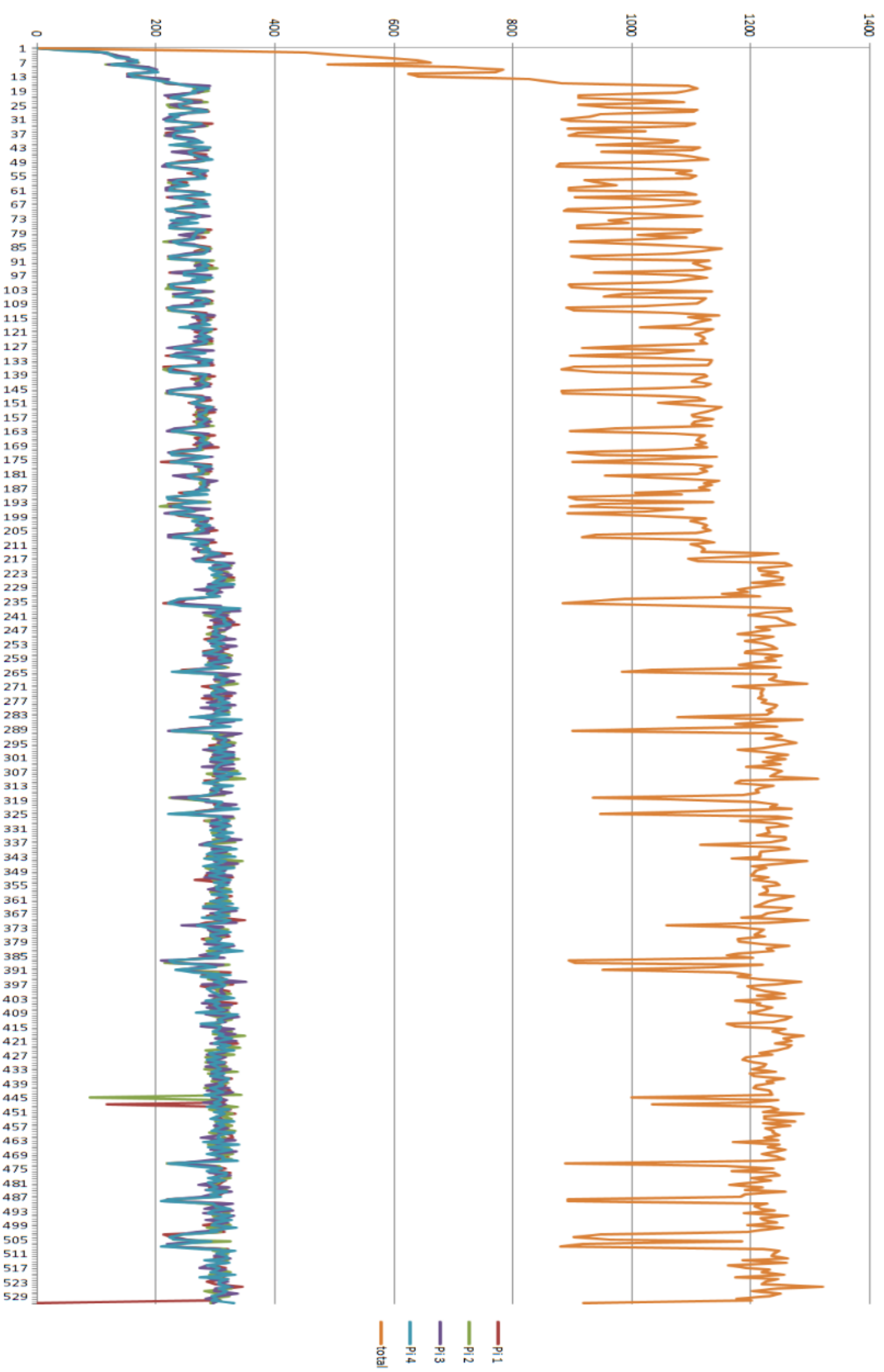
On peut voir ici que pendant l'exécution du test (environ entre 5 et 180 secondes), le processeur augmente sa charge. Si la somme des pourcentages du processeur utilisé par le user et le système est supérieur à 85%, on peut supposer que cela signifie que le processeur est surchargé et que le serveur peut nécessiter un nouveau processeur pour améliorer ses performances.

Entrées/sorties disque et mémoire RAM

Si notre application est sujette à réaliser de nombreuses E/S disque, cela peut créer un goulot d'étranglement car le couplage disque dur/OS a une vitesse limitée. Pour remédier à cela, nous pouvons changer les disques dur utilisés en les remplaçant par des plus performants, mettre en place un RAID ou bien augmenter la mémoire RAM. Niveau RAM, on peut simplement en ajouter pour augmenter le cache.

Aperçu d'un résultat et analyse

Nous venons de voir quelques unités qui peuvent être la cause d'une perte de performance du serveur. Pour pouvoir évaluer leurs performances, nous allons utiliser le projet Pi-Injector. Voici un exemple de résultat possible (lancé pour tester un serveur LDAP avec 4 Raspberry Pi) retourné par l'application.



On peut voir ici sur l'axe des ordonnées le nombre de requêtes par seconde et en abscisse le temps. Les courbes Pi 1, Pi 2, Pi 3 et Pi 4 représentent le nombre de requêtes effectuées par chaque Raspberry Pi et la courbe nommée « total » représente le total du nombre de requêtes pour tous les Raspberry Pi. Sur ce graphique, nous pouvons diviser la courbe en trois phases (que nous analyserons à partir de la courbe « total »).

La première phase se situe entre 1 et 7 secondes et elle représente le moment où les Raspberry Pi commencent à envoyer des requêtes sur le serveur cible. Il ne faut donc pas prendre spécialement en compte cette partie dans l'analyse.

La deuxième phase se situe entre 7 et 217 secondes. On peut voir que le nombre de requêtes stagne à environ 1000 requêtes par seconde. Cette partie représente une notion appelée « temps de chauffe ». Au démarrage d'une application Java, l'intégralité du bytecode est interprété par la JVM. La JVM entre alors dans une phase de chauffe durant laquelle elle optimise les fragments de code utilisés jusqu'à ce qu'elle atteigne un point d'équilibre permettant à l'application de fonctionner à vitesse maximale. La JVM peut compiler des fragments de code utilisés le plus fréquemment (nommés « hotspot »). Ce code compilé est alors stocké dans le cache et les pointeurs de code sont réécrits pour viser ce code optimisé. Comme pour la partie précédente, il ne faut donc pas prendre spécialement en compte cette partie.

La troisième phase commence à partir de 217 secondes et dure jusqu'à la fin de l'exécution simultanée des Raspberry Pi. C'est donc cette partie qu'il va falloir analyser.

Comme dit plus haut, ce graphique ne fait que représenter le nombre de requêtes par seconde. Au moment de la troisième partie, nous sommes arrivés à une stagnation du nombre de requêtes par seconde. Il va donc falloir analyser à ce moment précis toutes les unités qui peuvent réduire les performances du serveur pour en conclure si une de ses unités est au maximum de ses capacités.

Une utilisation possible de l'application est d'augmenter au fur et à mesure des tests le nombre d'injecteurs (sur un même scénario). Imaginons que sur le premier test lancé, avec un seul Raspberry Pi, le processeur du serveur est à 20% d'utilisation alors que sans exécution du test, il est à 10%. Sur le second test, il est à 30% et sur le 4ème test à 35%. On voit donc qu'entre le 3ème et le 4ème test, le gain n'est plus proportionnel. On peut donc supposer qu'il y a un des composants qui n'est pas performant.