Exemple d’utilisation de RabbitMQ

Julien Bénard

https://github.com/Julien29121998/LangAnalyzer

Présentation du projet

Tout langage utilisant l'écriture latine a un profil de fréquence d'utilisation des lettres différent. Il est donc en théorie assez simple d'estimer en quel langage un texte assez long est écrit en comparant sa répartition de lettres avec les profils des langues.

Cela nécessite donc deux opérations: le comptage des lettres, et la comparaison des statistiques obtenues avec les différents profils. La comparaison se fait via le calcul d'un coefficient de détermination entre les statistiques obtenues et un profil :

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i-1}^{n}(y_i \quad \bar{y})^2}$$

Amélioration des performances

Les deux opérations sont réalisées sur des textes assez longs, idéalement entre 50 000 caractères et 1 000 000 de caractères. Par conséquent, on va paralléliser ces opérations

-le texte peut être découpé en blocs, sur lesquels on effectue le comptage en parallèle, avant d'agréger les résultats en un seul ensemble.

-les résultats peuvent être comparés (calcul du coefficient de détermination) simultanément avec chacun des différents profils, de manière parallèle, puis le résultat final rassemblé avant d'être affiché.

Par ailleurs, de nombreuses API proposent de réaliser ces opérations sur le web, ainsi on peut envisager de procéder en envoyant des requêtes à celles-ci, pour qu'un serveur les traite parallèlement. Toutefois, puisque ces services sont généralement payants, on procèdera en déployant un ensemble de serveurs, en local, sur les ports de la machine, chacun proposant les deux services.

Utilisation de RabbitMQ

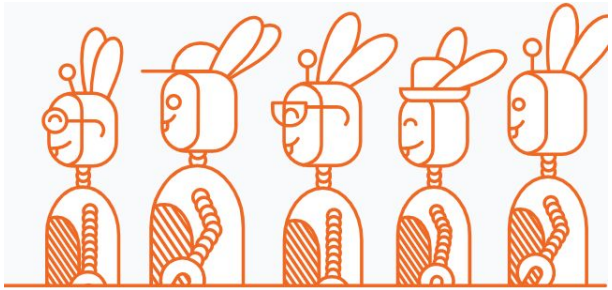
Besoins:

- distribuer les calculs des deux opérations (blocs de lignes de texte à compter, calcul de R2 pour chaque langage) entre les différents serveurs.
- encapsuler les appels aux serveurs derrière un appel de fonction (RPC).
- rendre l'ensemble scalable et facilement adaptable à un plus grand nombre de serveurs.

Solution:

Proposer un modèle hybride entre la *Worker Queue* et le modèle *RPC* des modèles de RabbitMQ en trois étapes

Solution avec RabbitMQ

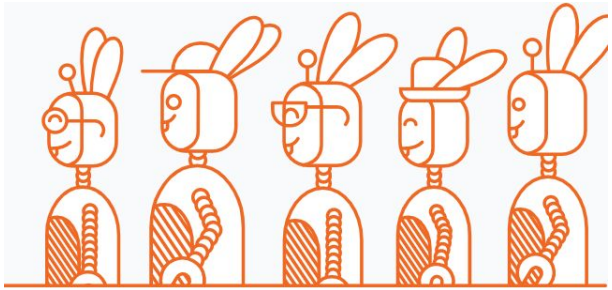


On découpe le processus en quatre étapes:

- Un contrôleur prend connaissance du nom du fichier à traiter, du nombre de serveurs disponibles et de leurs adresses (ports ici), et crée autant de processus *senders* que de serveurs. Il crée également deux queues (IN et OUT) rabbitmq, et trois sémaphores (verrous avec compteur).

- Le contrôleur lit le fichier, ligne par ligne, et envoie des paquets de 16 lignes dans la queue IN, qui se charge de les répartir entre les couples *sender-server* selon leur disponibilité. Chaque envoi fait décrémenter le premier sémaphore. Les senders contactent leurs serveurs respectifs pour obtenir des résultats et communiquent ceux ci au contrôleur via la queue OUT. à la réception, le contrôleur agrège les résultats et incrémente le premier sémaphore à chaque message reçu. Une fois celui-ci revenu à sa valeur initiale, tous les résultats ont été reçu et on peut passer à la phase suivante

Solution avec RabbitMQ



On découpe le processus en quatre étapes:

- Le contrôleur lit le fichier, ligne par ligne, et des couples [fréquences obtenues - profil d'une langue] dans la queue IN, qui se charge de les répartir entre les couples *sender-server* selon leur disponibilité. Chaque envoi fait décrémenter le deuxième sémaphore. Les senders contactent leurs serveurs respectifs pour obtenir des résultats et communiquent ceux ci au contrôleur via la queue OUT. à la réception, le contrôleur agrège les résultats et incrémente le deuxième sémaphore à chaque message reçu. Une fois celui-ci revenu à sa valeur initiale, tous les résultats ont été reçu et on peut passer à la phase suivante

- Le contrôleur va envoyer autant de messages "stop" qu'il y a de *senders*, en décrémentant le sémaphore 3. Ces messages vont stopper les *senders* et les faire quitter les queues. Ils répondent par un acquittement dans OUT qui vient incrémenter le sémaphore 3. Une fois celui ci revenu à sa valeur initiale, les résultats sont envoyés et le programme est stoppé.

Schéma d'architecture / déroulement

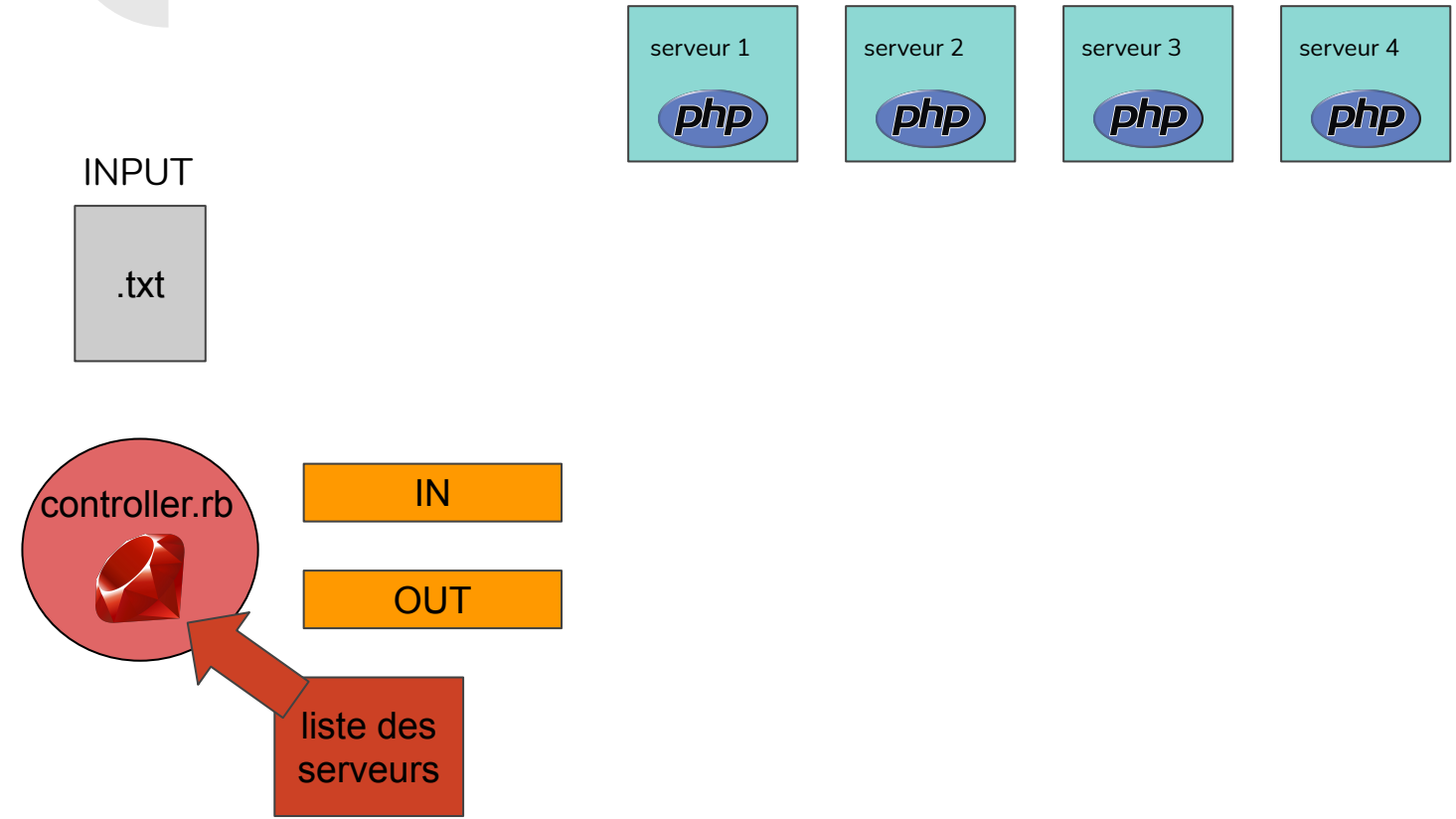


Schéma d'architecture / déroulement

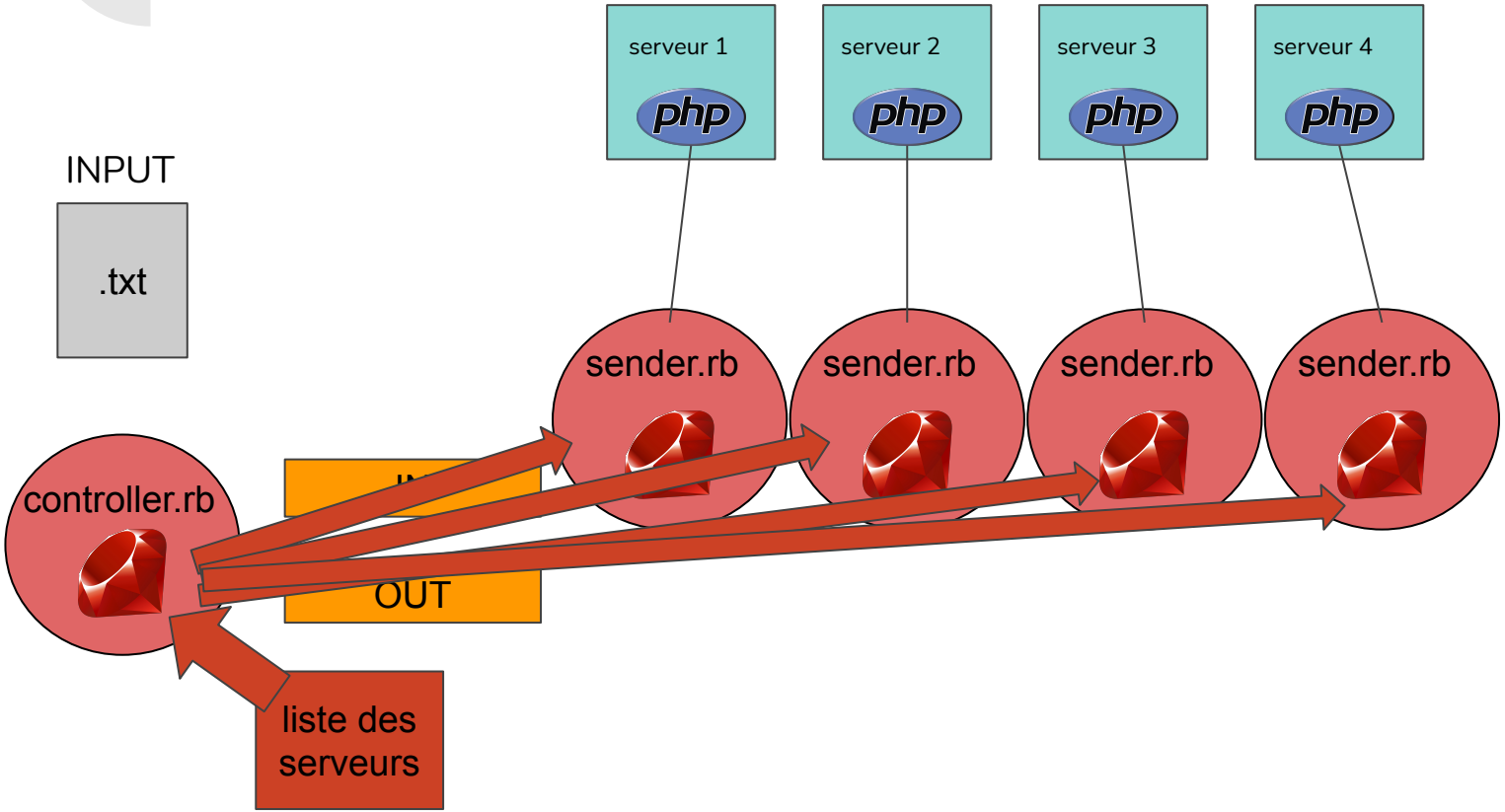


Schéma d'architecture / déroulement

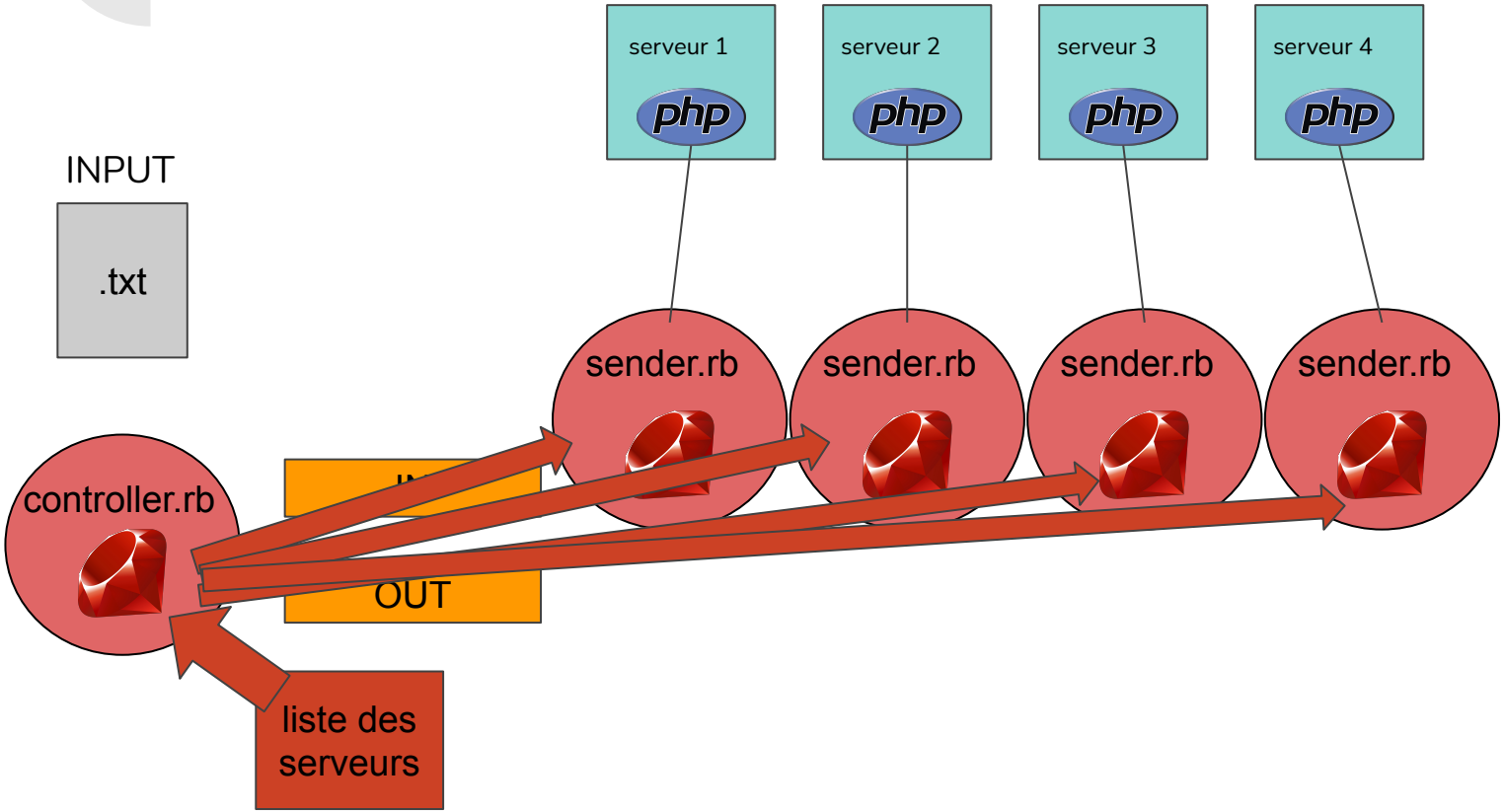


Schéma d'architecture / déroulement

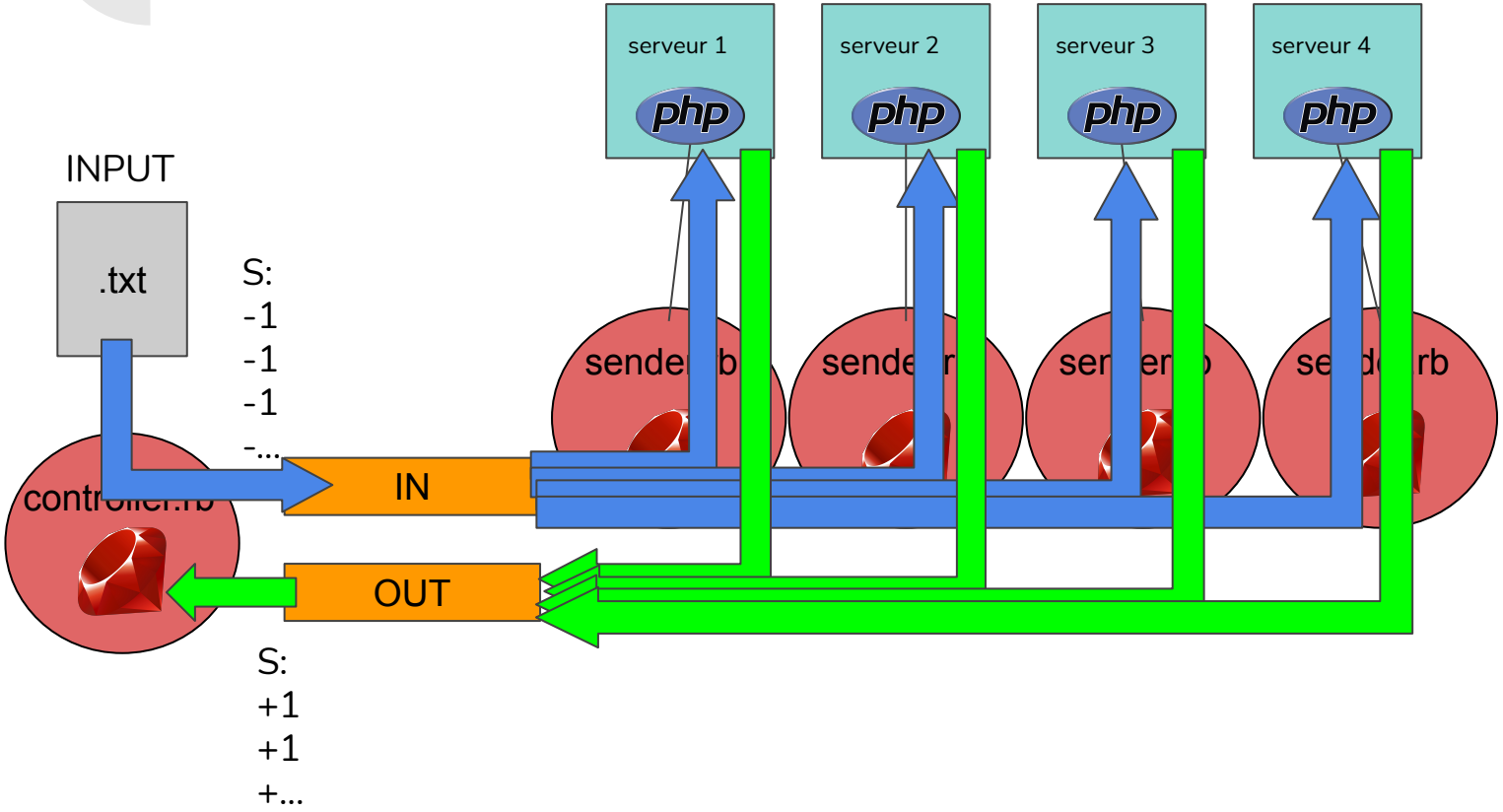
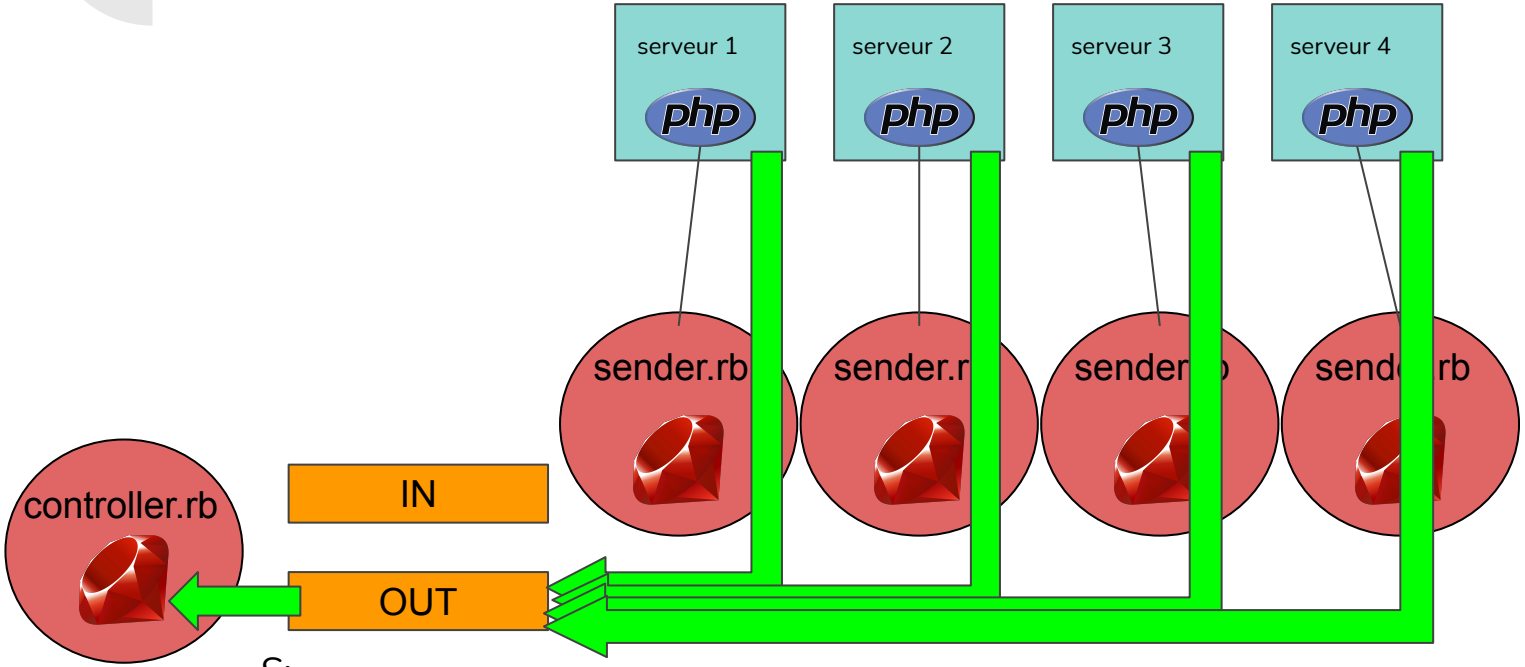


Schéma d'architecture / déroulement



S:
+1
+1
+...

Schéma d'architecture / déroulement

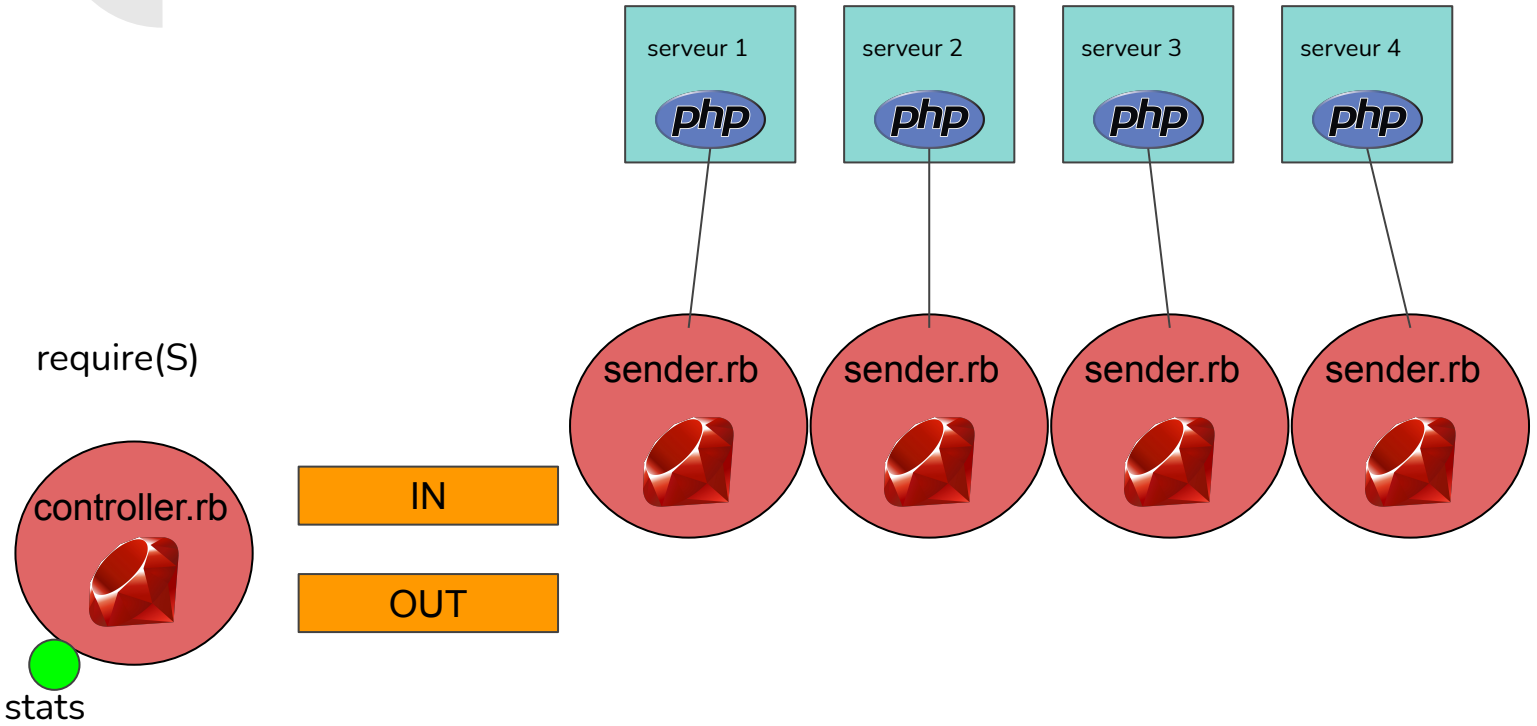


Schéma d'architecture / déroulement

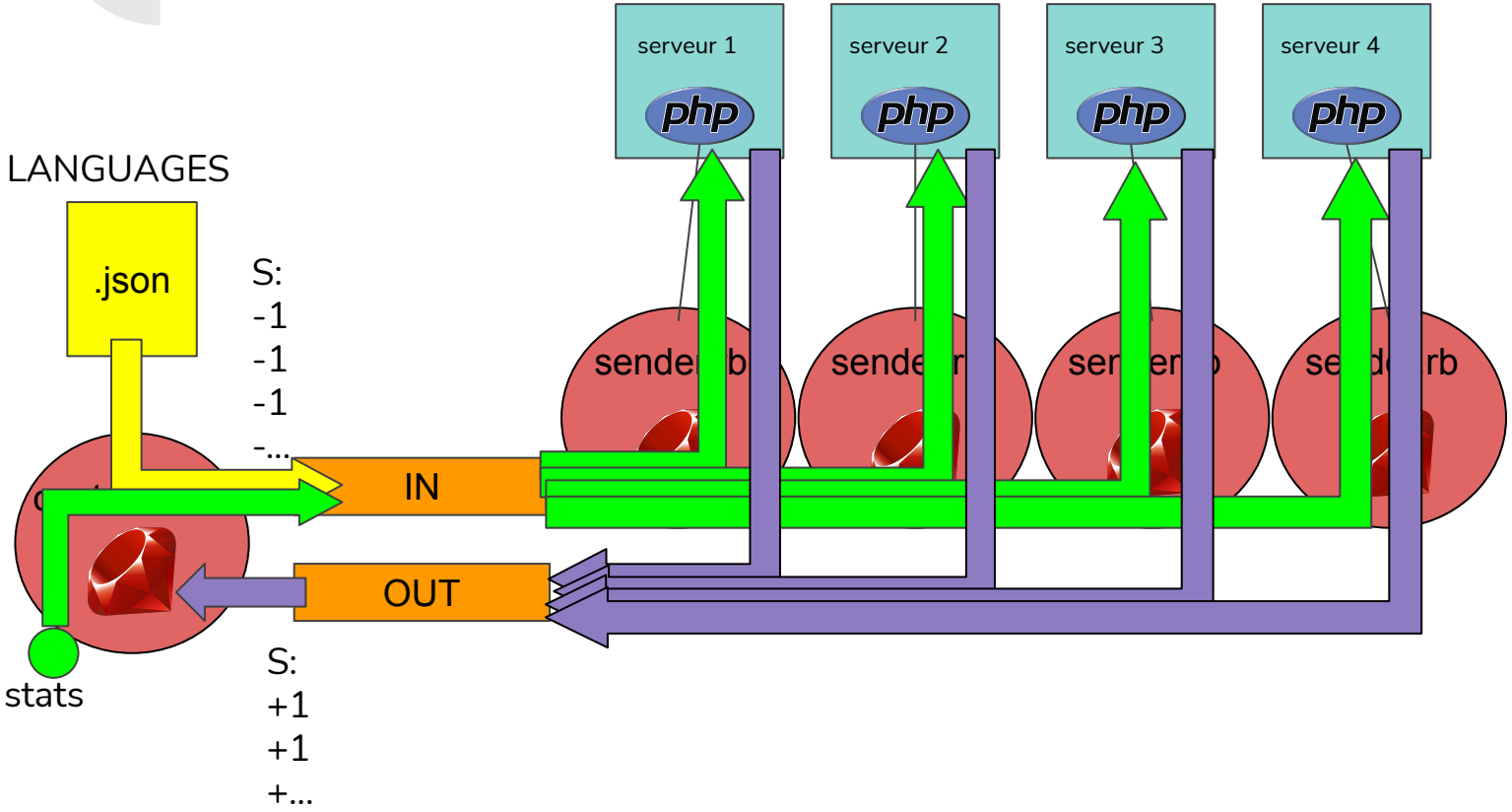


Schéma d'architecture / déroulement

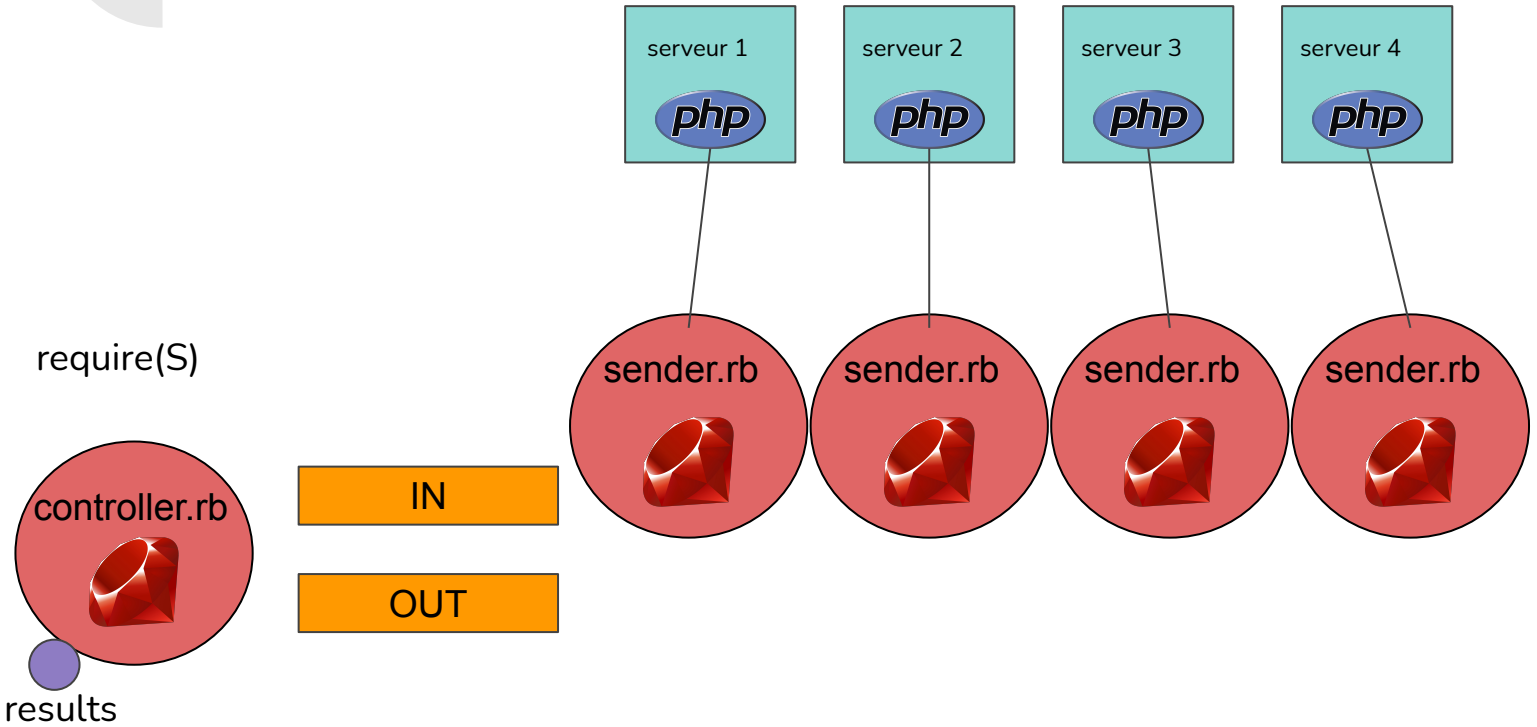


Schéma d'architecture / déroulement

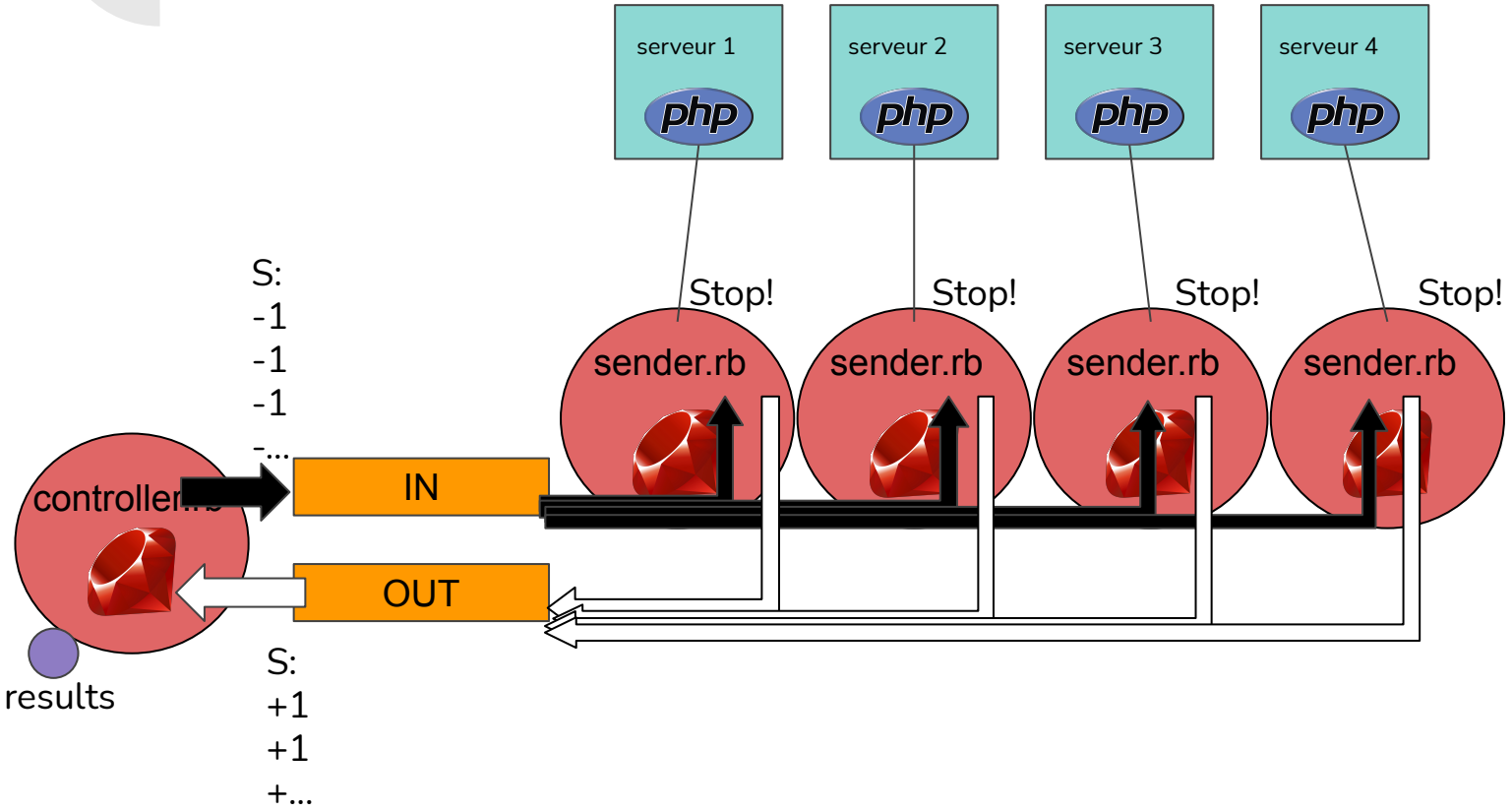
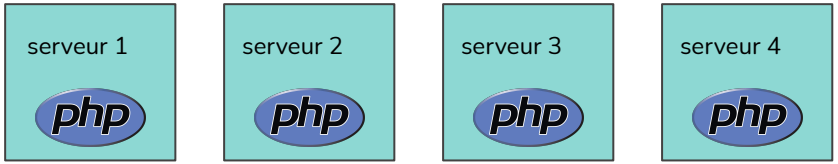
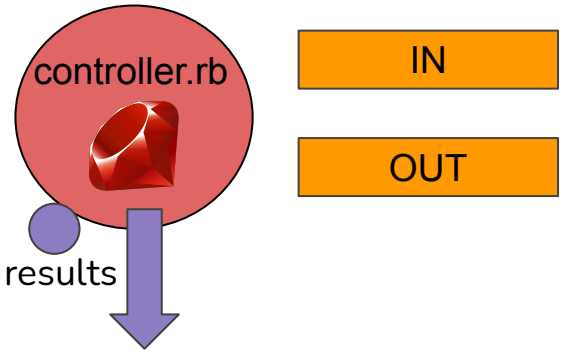


Schéma d'architecture / déroulement



require(S)



Intérêt de RabbitMQ

- Permet de gérer la distribution des tâches de manière équitable entre les serveurs
- Permet de constituer une queue dans laquelle stocker toutes les tâches à réaliser sans risquer d'en perdre ou d'en voir non-réalisées
- Permet de constituer une queue de résultats fournissant les mêmes avantages
- Il n'y a pas besoin de faire de changements si on veut rajouter des serveurs, la queue s'occupera de distribuer les tâches entre les senders
- Les messages envoyés dans les queues peuvent être monitorés via le port 15672
- facile à encapsuler dans le reste du programme