Bénard Julien FISE 3A ILSD*

UE CALC - Projet RabbitMQ

Sommaire

1. Introduction

Tout langage utilisant l'écriture latine a un profil de fréquence d'utilisation des lettres différent. Ainsi, si la langue française compte 7.6% de lettres **A** et 0.9% de lettres **B**, il en est autrement en est autrement en danois qui a respectivement des pourcentages de 6% et 2% pour ces lettres. 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.

L'objectif de ce projet est donc de proposer un programme rapide et efficace qui utilise ces propriétés pour détecter la langue d'un texte donné.

Le projet reposant sur des comptages de lettres et des analyses fréquentielles et statistiques, cela suppose que les textes entrés sont de grande taille. Cependant, comme on a pu l'annoncer plus tôt, il s'agit également de procéder rapidement. On cherche donc à paralléliser les opérations effectuées lors du traitement. Afin de parvenir à cette architecture, RabbitMQ est utilisé.

Ce projet se révèle donc comme une excellente occasion de présenter les fonctionnalités de RabbitMQ dans le contexte d'un programme.

2. Opérations réalisées

Le traitement à effectuer se décompose en deux opérations:

- La première est le comptage des lettres. Il s'agit de lire le fichier d'entrée progressivement et de compter et répertorier les apparitions des différentes lettres dans un dictionnaire associant les lettres à leurs nombres d'occurrences. Une fois le décompte effectué, les totaux sont divisés par le nombre de lettres reconnues afin d'obtenir des fréquences.

- La seconde consiste en la comparaison des statistiques obtenues avec les différents profils statistiques des langues en base de données. La comparaison se fait via le calcul d'un coefficient de détermination R^2 entre les statistiques obtenues sur le texte entré y et un profil de fréquences d'une langue donnée $\hat{y}$:

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

où $\bar{y}$ est la moyenne des statistiques obtenues lors du décompte.

3. Parallélisation

Les deux opérations présentées plus haut sont réalisées sur des textes assez longs, idéalement entre 50 000 caractères et 1 000 000 de caractères, afin de retourner des résultats précis. Par conséquent, il est nécessaire de paralléliser ces opérations, pour respecter la volonté initiale du projet qui était d'effectuer le traitement en un temps restreint.

Le texte peut être découpé en blocs, sur lesquels on effectue le comptage des apparitions des lettres en parallèle, avant d'agréger les résultats en un seul ensemble, et d'effectuer la division par le nombre total. La taille du découpage est de 16 lignes de texte par bloc. Ce choix est arbitraire mais permet de facilement découper des textes de tailles variées avec flexibilité, sans devoir garder en mémoire une partie trop importante du texte entre chaque envoi.

Les analyses fréquentielles peuvent être comparées (calcul du coefficient de détermination) simultanément avec chacun des différents profils, de manière parallèle, puis le résultat final rassemblé avant d'être affiché. Chaque serveur se charge alors de la comparaison des statistiques mesurées et de celles d'une ou plusieurs langues.

Par ailleurs, de nombreuses API proposent de réaliser ces deux opérations sur le web, ainsi on pourrait 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 procède en déployant un ensemble de serveurs, en local, sur les ports de la machine, chacun proposant les deux services. Ils serviront à simuler l'utilisation de véritables serveurs. La liste des adresses (ports) est fournie au programme au démarrage.

Ainsi, on espère pouvoir optimiser au mieux le programme.

4. RabbitMQ

L'architecture proposée vient créer plusieurs besoins:

- Distribuer les calculs des deux opérations (blocs de lignes de texte à compter, calcul de R2 pour chaque langage) entre les différents serveurs: il s'agit de ne pas les surcharger, ni de les laisser inactifs. Il est également important de répartir équitablement les opérations pour espérer pouvoir les finir le plus vite possible.

- Encapsuler les appels aux serveurs derrière un appel de fonction, à la manière d'un appel *RPC*. De cette manière, l'utilisation est facilitée et rendue plus modulable.
- Rendre l'ensemble du programme scalable et facilement adaptable à un plus grand nombre de serveurs,

La solution principale qui se présente alors est RabbitMQ, un système de passage de messages asynchrones basé sur des queues. Il implémente le protocole AMQP, qui régit ces échanges. Il est écrit en Erlang et est actuellement sous licence MPL (licence libre de Mozilla).

RabbitMQ propose un système très développé pour l'échange de messages entre des programmes:

- Une connexion peut être établie avec un host donnée, puis proposer des channels, qui serviront à établir des queues.
- Les queues sont les supports servant à passer les messages, elles fonctionnent comme des FIFO unidirectionnelles, et peuvent être persistantes et survivre aux programmes les utilisant. Elles distribuent les messages issus de leurs producteurs entre les consommateurs placés en sortie, et renvoient en l'absence d'acquittement.
- Les échangeurs sont des nœuds permettant de trier les messages entre des queues sur la base de critères, ou de les distribuer à toutes.
- L'ensemble de l'utilisation de RabbitMQ, en particulier la charge sur les queues, peut être consulté et monitoré via le port 15672 qui offre un panneau de contrôle sur RabbitMQ.

On utilise par conséquent cet outil pour répondre à nos besoins. La solution proposée se base sur une architecture hybride inspirée par deux modèles présentés dans la documentation de RabbitMQ, les modèles de la *Worker Queue* et le modèle du *RPC.*

5. Architecture

On construit l'architecture du système de la manière suivante: un contrôleur est chargé de la construction et du dimensionnement du reste du système, ainsi que de la lecture des données et de l'ordonnancement général des taĉhes globales (lecture, statistiques, comparaison, affichage, arrêt). Ce contrôleur est relié à deux queues de RabbitMQ qui lui servent à communiquer avec les envoyeurs. La première, la queue d'entrée, se charge de répartir les sous-tâches de comptage et de comparaison entre les envoyeurs, tandis que la seconde, la queue de sortie, sert à rassembler les réponses des couples envoyeurs-serveurs pour les renvoyer au contrôleur.

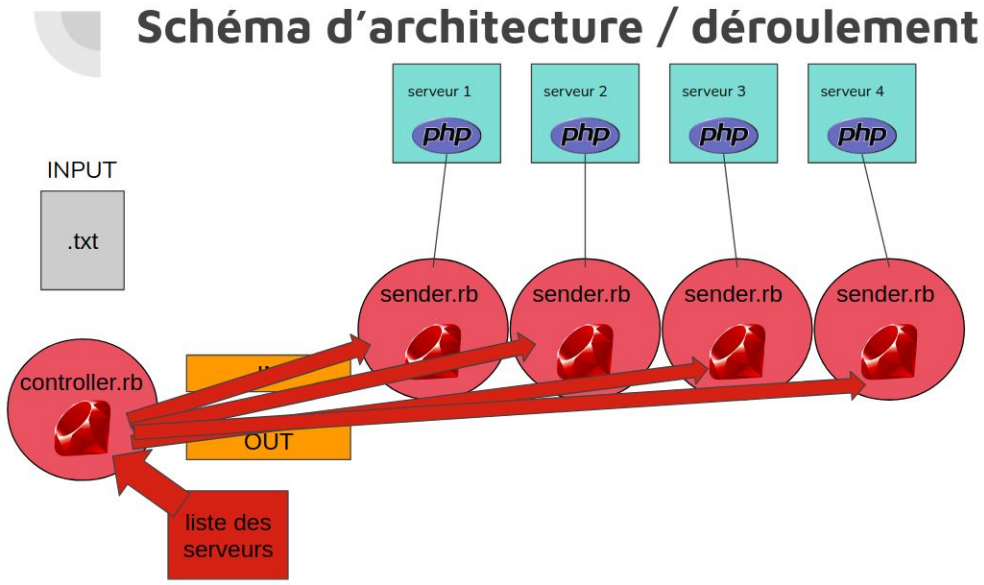
Les envoyeurs sont chargés de la communication avec les serveurs. Il y a autant d'envoyeurs que de serveurs et chaque envoyeur est associé à l'un d'eux. Ils reçoivent les données après répartition et communiquent avec leur serveur pour obtenir des résultats. Ils ont pour fonction de s'interfacer entre les queues permettant la distribution, et les serveurs chargés de traiter les données. Ils utilisent le protocole HTTP pour communiquer avec ceux-ci.

Les serveurs sont de simples conteneurs de docker qui simulent des serveurs capables de répondre à des requêtes sur deux endpoints, l'un réservé au comptage, l'autre à la comparaison des fréquences.

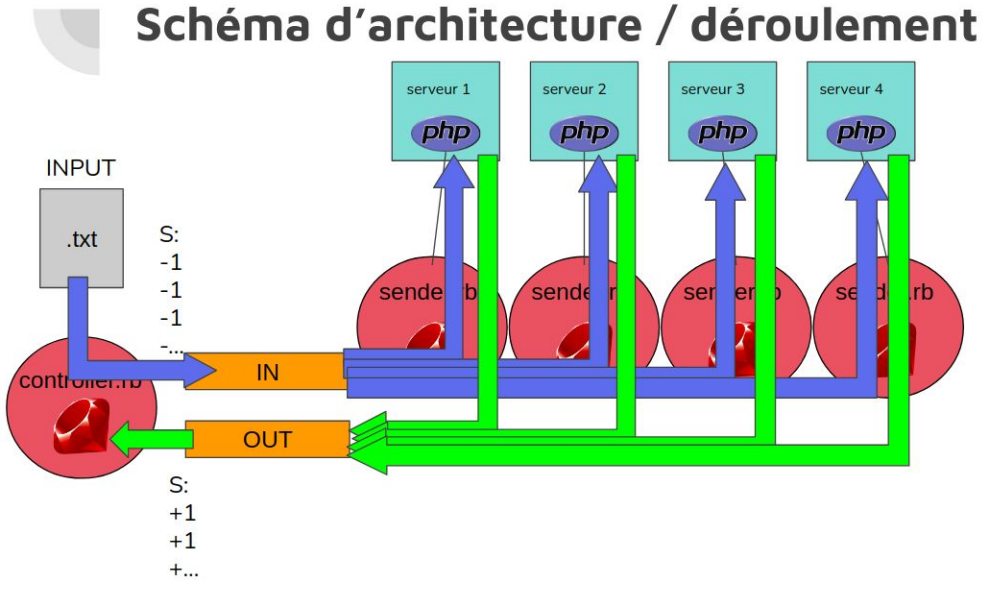
On peut alors découper la solution en quatre étapes:

- 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

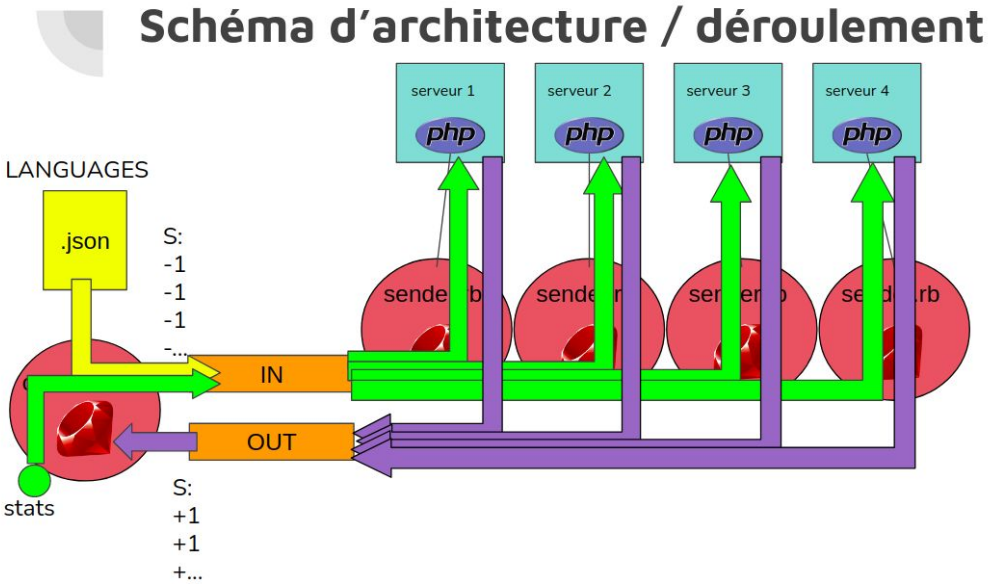
de serveurs. Il crée également deux queues (IN et OUT) rabbitmq, et trois sémaphores (verrous avec compteur).



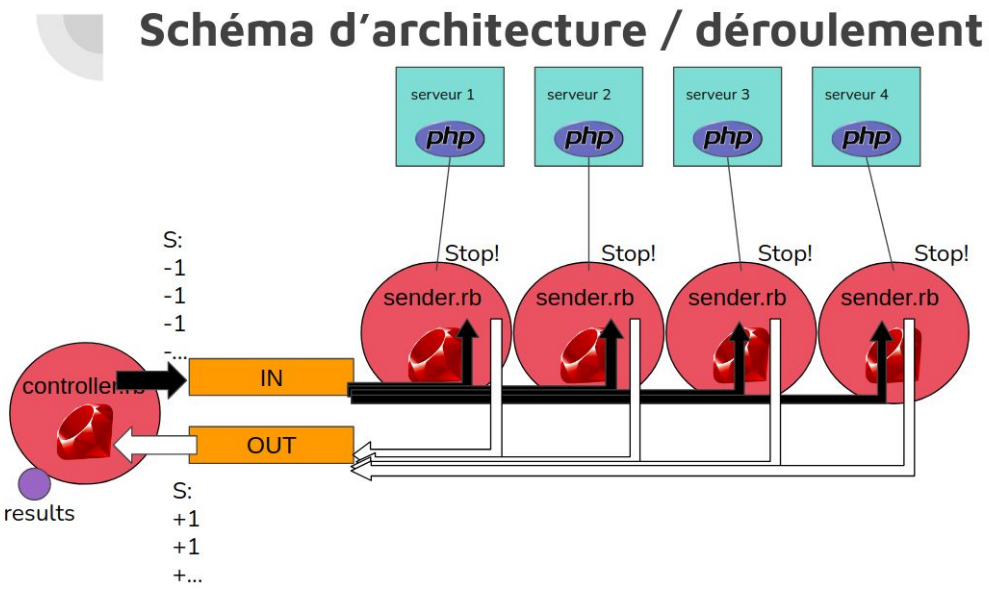
- Le contrôleur lit le fichier d'entrée, ligne par ligne, et envoie des paquets de 16 lignes dans la queue IN, qui se charge de les répartir entre les couples sender-server selon leur disponibilité. Chaque envoi fait requérir 1 au premier sémaphore. Les senders contactent leurs serveurs respectifs pour obtenir des résultats et communiquent ceux-ci au contrôleur via la queue OUT. à la réception, le contrôleur agrège les résultats et fait libérer 1 au premier sémaphore à chaque message reçu. Une fois celui-ci revenu à sa valeur initiale, cette valeur peut être requise, et alors tous les résultats ont été reçus et on peut passer à la phase suivante.



- Le contrôleur lit la base de données de langues, ligne par ligne, et envoie 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 requérir un au 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 fait libérer 1 au deuxième sémaphore à chaque message reçu. Une fois celui-ci revenu à sa valeur initiale, cette valeur peut être requise, et alors tous les résultats ont été reçus et on peut passer à la phase suivante.



- Le contrôleur va envoyer autant de messages "stop" qu'il y a de senders, en faisant requérir 1 au sémaphore 3 à chaque fois. Ces messages vont stopper les senders et les faire quitter les queues. Ils répondent par un acquittement dans OUT qui vient faire libérer 1 au sémaphore 3 à chaque fois. Une fois celui-ci revenu à sa valeur initiale, cette valeur peut être requise, et alors tous les résultats ont été reçus et le programme peut être stoppé.



6. Choix d'implémentation

Le projet, dans sa partie client, a été codé en ruby, qui est un langage haut niveau permettant facilement de réaliser des programmes simples dans un laps de temps court. Ce choix est également soutenu par le fait qu'il existe des librairies permettant d'utiliser RabbitMQ avec de nombreux langages, dont Ruby (la gemme Bunny). Ruby présente également l'avantage, avec sa vision objet, de rendre le traitement des chaînes de caractères facilité. Enfin, ce langage gère le multithreading et l'appel de commandes dans le terminal, ce qui a aidé à la réalisation du projet.

Le côté serveur a été codé en Php, un langage adapté à ce genre d'architectures, et très largement utilisé dans le monde du web. Il rend la gestion des endpoints et des API REST simplifiée, ce qui a permis au projet d'être terminé à temps, sans avoir recours à des serveurs extérieurs payants déjà prêts.

De ces deux côtés, j'ai pu utiliser Docker, qui est un outil très puissant pour les questions de distribution et de compatibilité. En effet, il permet par exemple de faire fonctionner la partie client du projet, qui utilise RabbitMQ, sans autre installation préalable que celle de Docker. D'une manière similaire, les différents serveurs ne requièrent pas d'avoir PHP ou Apache server d'installés sur la machine. Par ailleurs, Docker sert à la distribution puisqu'il permet de répliquer des containers de serveur en autant d'exemplaires que l'on veut, et donc de simuler autant d'endpoints, disponibles pour recevoir des requêtes, que l'on souhaite.

Ces containers Docker sont des petites machines virtuelles linux qui peuvent recevoir des installations variées suivant un fichier d'instructions de configuration, le *dockerfile*. Ici, j'utilise deux images Docker: l'une d'elle, basée sur une image RabbitMQ, porte le côté client; l'autre, basée sur une image Apache php server, est répliquée autant de fois que nécessaire pour créer de multiples serveurs disponibles sur plusieurs ports de l'ordinateur.

Plusieurs choix ont également été faits au niveau du code même. Par exemple, lors de la lecture des fichiers, qu'il s'agit du fichier de langues ou du fichier d'entrée, le code en ruby procède à une lecture ligne par ligne afin d'éviter de charger brusquement l'ensemble du texte, de grande taille en mémoire.

Pour ce qui est du contrôle de l'ordre des différentes étapes du processus, vérifié par le contrôleur, on utilise un sémaphore, plutôt qu'un mutex comme dans l'exemple rpc du code donné sur la documentation. Cela est lié au fait qu'on souhaite attendre le retour de l'ensemble des tâches d'une étape avant de commencer la suivante: ainsi, avant de commencer à comparer les langues, on s'assure que tous les blocs du texte initial ont été comptés et que leurs résultats ont été reçus.

Le format des données envoyées, que ce soit à travers les queues, ou encore simplement dans les requêtes HTTP, est le format JSON. Ce format a été choisi car il répond au besoin de transformer les données objet en chaînes de caractères (Évidemment seuls des types primitifs sont admis par RabbitMQ et les messages HTTP) entre deux langages orientés objet.

Il permet alors de passer aisément de types objet complexes à des simples chaînes de caractères et inversement.

Néanmoins, il est important de noter que ces transferts d'informations requièrent certains types d'encodage, qui font perdre de nombreux caractères spéciaux tels que les **š**, les **ê**, ou les **ž**. Ainsi, cela, et la complexité d'étendre le comptage à l'ensemble des caractères pourvus de diacritiques, a influencé le choix de ne compter que les 26 lettres communes de l'alphabet latin. Ces lettres suffisent à identifier les différents langages dans une majorité des cas.

7. Utilisation

Faire fonctionner le programme requiert Docker sur une distribution Linux. Si vous n'avez pas Linux, mais Docker malgré tout, vous pouvez faire fonctionner le programme en adaptant le contenu des makefiles aux commandes associées à votre système d'exploitation, ou en étudiant le *README.html* présent dans le projet.

Pour mettre en place l'ensemble des containers, il faut utiliser *make docker.* Cette instruction va successivement construire les images docker des serveurs et du client, et les démarrer ensuite.

Pour lancer le programme, il y a deux possibilités:

- utiliser *make example* pour relancer le programme sur le dernier texte étudié par le programme (ou l'exemple initial, l'intégralité du texte de *Ocean, mare* d'Alessandro Baricco en version originale).
- utiliser *make run_on_input* après avoir complété *input.txt* avec un nouveau texte à considérer.
- Il est également possible, si l'utilisateur a une bonne compréhension de la manière dont le programme fonctionne, de personnaliser les instructions fournies au programme, pour des résultats spécifiques.

Une fois le programme lancé, sur le texte par défaut ou n'importe quel texte, les étapes décrites plus tôt s'enchaînent, jusqu'à l'arrêt des senders, notifié dans la console. Ensuite s'affichent des premiers résultats: l'analyse fréquentielle du texte, c'est-à-dire les pourcentages d'occurrences pour les lettres de **A** à **Z** dans l'ordre (ainsi par exemple, la 5e

valeur est le pourcentage de présence de la lettre **E** dans le texte). Ensuite s'affichent les ressemblance avec les différents langages en base de données, dans l'ordre, où la valeur indiquée correspond à R^2. Enfin s'affiche le langage le plus probable, correspondant au premier élément de la liste précédente. Il correspond à la décision finale du programme, le langage dans lequel le texte a la plus forte probabilité d'être écrit.

8. Exemple et Résultats

On observe les résultats de l'exemple fourni, un texte long en italien:
On peut voir que les différents décomptes ont fonctionné, et que le programme a effectivement détecté la langue avec une ressemblance de 99%. Les langues ayant eu des coefficients proches sont celles qui sont apparentées, les langues romanes en particulier.
En testant avec d'autres exemples, de longueurs et d'origines variables, on remarque que la détection est généralement juste, même si le texte ne fait que quelques lignes.

Toutefois les résultats ne sont pas aussi bons, pour le tchèque par exemple, langue avec de nombreux caractères spéciaux, il subsiste beaucoup de faux négatifs.

```
cp input.txt ./client/sample.txt && cd client && make run
make[1] : on entre dans le répertoire « /home/administrateur/Bureau/TPCALC/client »
echo "run test"
run test
docker exec -it calc_client ruby controller.rb "sample.txt" 8470 8971 8972 8974
Task finished, worker dies.
Task finished, worker dies.
Task finished, worker dies.
Task finished, worker dies.
FREQUENCY ANALYSIS

{"freq":[0.11777675398183493,0.012180559995486 94,0.04420918031553809,0.03496681020703661,0.11862765377310593,0.009585550687301379,0.015725192275146203,0.012298087590966358,0.0978957859305365,1.880441527670697e-05,6.581545346847439e-05,0.0639350119408037,0.030387935087158466,0.072589744071908 09,0.093095958931157 03,0.02632618138738976,0.007234998777713007,0.06434870907689125,0.05568927584196769,0.05814325203557796,0.0343415633990861,0.022057579119577276,0.0005406269392053254,0.000164538633671186,0.00033847947498072547,0.007455950657214314],"total":212716}

POSSIBLE LANGUAGES

{"Italian":0.9931971495032902,"Spanish":0.9157496558003538,"Portuguese":0.84759111342877,"Esperanto":0.8431091911900772,"French":0.8189519254478089,"Swedish":0.786899679346034,"English":0.7584937097524724,"Danish":0.6346630710399888,"Finnish":0.6147224456145699,"Turkish":0.5997050418821117,"Icelandic":0.575130565777709,"Dutch":0.5750181986881807,"Polish":0.5646380501018822,"German":0.5576350641271284,"Czech":0.4722812017027719}

MOST LIKELY

Italian
0.9931971495032902
```

Pour ce qui est de l'emploi de RabbitMQ, comme espéré le programme renvoie les résultats en un temps raisonnable. RabbitMQ a permis de gérer la distribution des tâches de manière équitable entre les serveurs, de constituer une queue dans laquelle stocker toutes les tâches à réaliser sans risquer d'en perdre ou d'en voir non-réalisées, et de constituer une queue de résultats fournissant les mêmes avantages.

Également, grâce à ce système, il n'y a plus besoin de faire de changements si on veut rajouter des serveurs, puisque la queue s'occupera de distribuer les tâches entre les différents senders correspondants aux serveurs ajoutés.

La possibilité de monitorer, via le port 15672, les messages envoyés, le détail du remplissage et de l'état des queues, rend l'outil encore plus utile, lorsqu'il s'agit de corriger des erreurs dans le programme par exemple.

Enfin, la manière dont fonctionnent les modules associés à rabbitMQ dans les différents langages le rendent facile à encapsuler dans n'importe quel programme, et dans le projet, ici, en particulier.

9. Conclusion

En conclusion, le programme a montré que l'on pouvait facilement estimer la langue dans laquelle était écrite un texte en se basant sur une analyse fréquentielle de ses lettres, et ce, sur des entrées de la taille pouvant descendre jusqu'au paragraphe.

Par ailleurs, il s'agit d'un bon exemple de système distribuant des calculs entre différentes nodes (ici les serveurs), et utilisant des modes de communications variés et asynchrones, ainsi que la technologie de Docker pour simuler la distribution sur de nombreux serveurs.

Enfin, ce projet fournit une démonstration des capacités de RabbitMQ et de ses utilisations dans le contexte d'une application spécifique. On en retiendra ses nombreux avantages, qui ont pu être listés dans les parties précédentes.

L'intégralité du code du projet, de la documentation et des rendus associés peut être consultée ici: https://github.com/Julien29121998/LangAnalyzer.