Troisième rendu :

Architecture

Salmane Bah & Tristan Braquelaire & Wassim Romdan & Timothée Sollaud

Université de Bordeaux

5 juillet 2014

Table des matières

Introduction

Dans le cadre de l'unité d'enseignement "Projet de Programmation", suivie durant la formation de Master 1 informatique de l'Université de Bordeaux, il nous a été proposé de réaliser une plateforme de compilation à distance. Celle-ci doit être basée sur docker afin de sécuriser la compilation et l'exécution de code émis par un client distant. Dans ce rapport, nous allons présenter l'architecture envisagée pour les principales versions qui seront développées.

Pour tous les schémas présentés dans ce rapport, nous avons utilisé la même symbolique : les flèches simples représentent les liens de cardinalité 1.1 alors que les flèches doubles représentent les liens de cardinalité 0.N (collection).

1 L'architecture générale du serveur

L'architecture générale est celle d'un système Client/Serveur classique. Cependant, dans notre projet, nous n'avons rien à prévoir côté client. En effet, celui-ci a simplement besoin que nous publions l'API relative aux services que proposera notre plateforme de compilation & d'exécution à distance.

Le serveur quant à lui sera constitué de trois grandes parties :

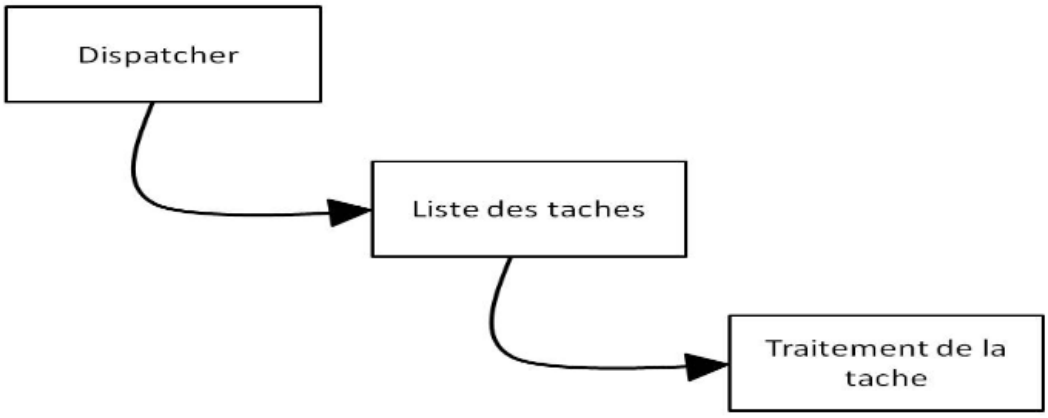

Figure 1 – Diagramme général de l'architecture du système

1. **Le dispatcher**
   En plus de permettre la réception en continu des requêtes clientes, il génère les objets nécessaires à la gestion d'un client et de ses tâches. Ainsi il gèrera à terme, les sessions avec leurs tâches et traitera les requêtes de faible traitement qui ne consistent qu'en une simple lecture des objets (à savoir l'intégralité des fonctions *connect*, *disconnect*, *getLanguages*, *getDetails*, *getStatus* et la création d'une tâche associée à un *sendRequest*). Il enverra les tâches complexes (nécessitant une compilation ou une execution) sur une file d'attente, afin qu'elles soient traitées dans une sandbox docker.

2. **File de Tâches**
   La file de tâche représente l'ensemble d'objets permettant la répartition des tâches lourdes reçues du dispatcher vers la partie de traitement.

3. **Traitement de la tâche**
   C'est là que seront réellement effectuées les requêtes complexes et sensibles du client. Il s'agit précisément de la compilation et de l'exécution d'une tâche. Cet environnement sera sécurisé à l'aide de Docker, utilisé en sandbox. A terme, plusieurs environnement docker pourront tourner simultanément.

2 Première version : gestion des sessions avec un seul environnement docker

Cette première version présentera un "dispatcher" déjà bien formé. En effet, si cette version offrira déjà la gestion de multiples sessions, elle se limitera au strict minimum concernant les deux autres parties : une simple file et un environnement docker unique pour le traitement de toutes les tâches de compilation et d'exécution.

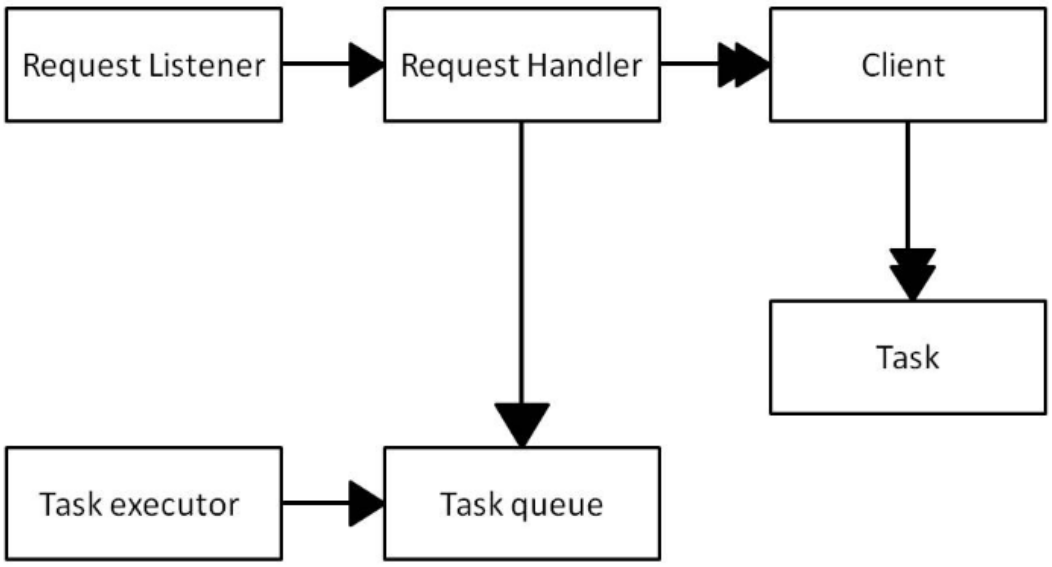

FIGURE 2 – Diagramme de l'architecture de la première version du système

Le dispatcher se décomposera en deux parties :

- **requestListener**
  Objet qui se contentera de maintenir en continu un port d'écoute. Il ne connaîtra qu'un unique objet : "requestHandler", auquel il enverra toutes les requêtes reçues.

- **requestHandler**
  Objet qui permettra de lancer le traitement de la requête. Il connaîtra la liste des clients, et gèrera pour chaque client, la liste de ses tâches associées. C'est lui qui transmettra une tâche à traiter à la file d'attente (taskQueue).

Finalement, la file d'attente ne sera ici qu'une simple file : taskQueue. L'unique environnement docker "taskExecutor" viendra prendre les tâches, dans cette unique file d'attente, une à une afin de les traiter. Il mettra également à jour les champs de la tâche pour y renseigner les résultats du traitement.

3 Deuxième version : gestion des sessions et de plusieurs environnements docker

Dans cette version plus complexe, le dispatcher ne sera pas modifié. En revanche, l'objectif principal de cette version est de permettre la gestion de multiples plateformes docker gérant différents langages.

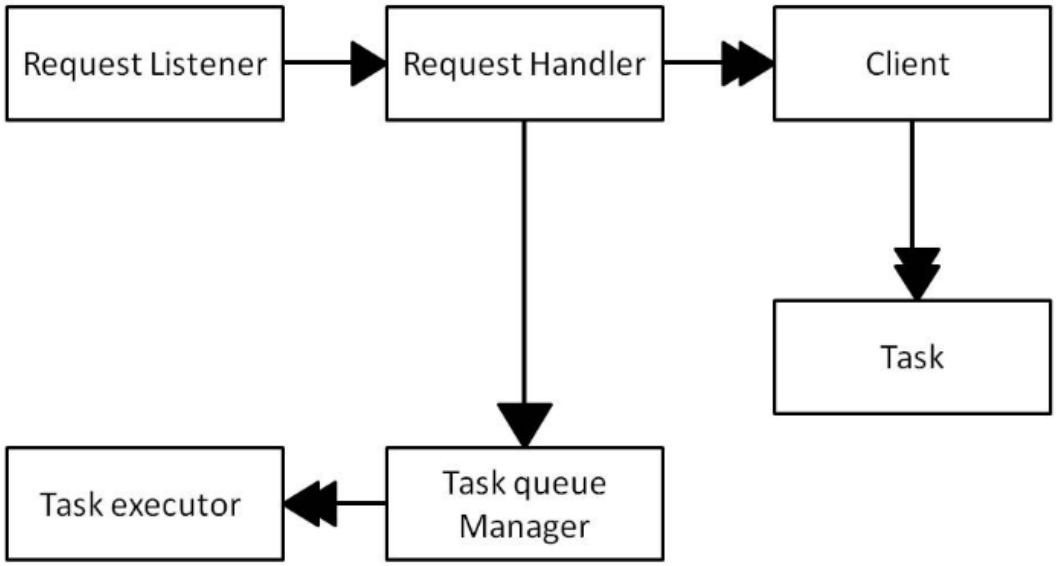

FIGURE 3 – Diagramme de l'architecture de la deuxième version du système

La simple file d'attente alors utilisée jusqu'à présent sera remplacée ici par un objet plus complet : taskQueueManager. Son rôle principal sera de répartir efficacement les tâches sur les différents environnements docker à sa disposition. Le taskQueueManager représentera l'ordonnanceur des tâches et sera finalement le cœur algorithmique de notre projet. Nous avons envisagé plusieurs stratégies de distribution possibles :

1. Une file globale (les serveurs de compilations viennent chercher des tâches à exécuter).
   Avantages :
   – Simple à implémenter
   – Peu de travail pour le taskQueueManager
   Inconvénients :
   – Beaucoup de synchronisations
   – Complexité en O(n)

2. Une file principale dans le Task Queue Manager recevant les requêtes de Request Handler, qui répartie les tâches à chaque serveur, selon le(s) langage(s) pris en compte. Chaque serveur possèdent une file de tâches lui étant propre.
   Avantages :
   – il n'y a plus de synchronisation entre les différents serveurs de compilation/exécution
   Inconvénients :
   – utilisation mémoire : une file par serveur, une file principale

3. Une file principale dans le Task Queue Manager recevant les requêtes de Request Handler. Celles ci sont réparties dans de nouvelles files selon le langage nécessaire à leur exécution. Les serveur viennent demander une tâche au Task Queue Manager, qui examinera les langages pris en charge par le serveur et ira chercher une tâche dans l'une des files adéquate.
   Avantages :
   – synchronisation réduite pour les serveurs de compilation par rapport à la stratégie 1 (seuls les serveurs concernés par un langage se synchronisent)
   Inconvénients :

- utilisation mémoire : une file principale, une file pour chaque langage (qui augmente à chaque ajout d'un langage)

4. Une file principale dans le Task Queue Manager recevant les requêtes de Request Handler. Celles-ci sont envoyés dans une liste principale si elles concernent des langages traités par plusieurs serveurs se trouvant dans le Task Queue Manager, ou dans la file de l'unique serveur traitant le langage de la tâche. Le Task Queue Manager observe en continue l'activité des serveurs afin de leur donner les tâches adéquates lorsqu'ils sont à nouveau disponibles.
   Avantages :
   - Bonne répartition des tâches à langage moins utilisé sur les différents serveurs de compilation
   - Équilibre de la charge de travail entre TaskQueueManager et les serveurs de compilation/exécution

   Inconvénients :
   - Synchronisation (quoique faible) sur la file principale pour les langages communs aux serveurs de compilation/exécution

4 Troisième version : Gestion des sessions, de plusieurs environnement docker et de l'administration

Cette dernière version offrira une API dédiée à l'administration de la plateforme de compilation. Il s'agira concrètement d'offrir la possibilité, à un administrateur distant, de modifier la configuration des environnements docker disponibles : leur nombre mais aussi la liste des langages qu'ils gèreront.

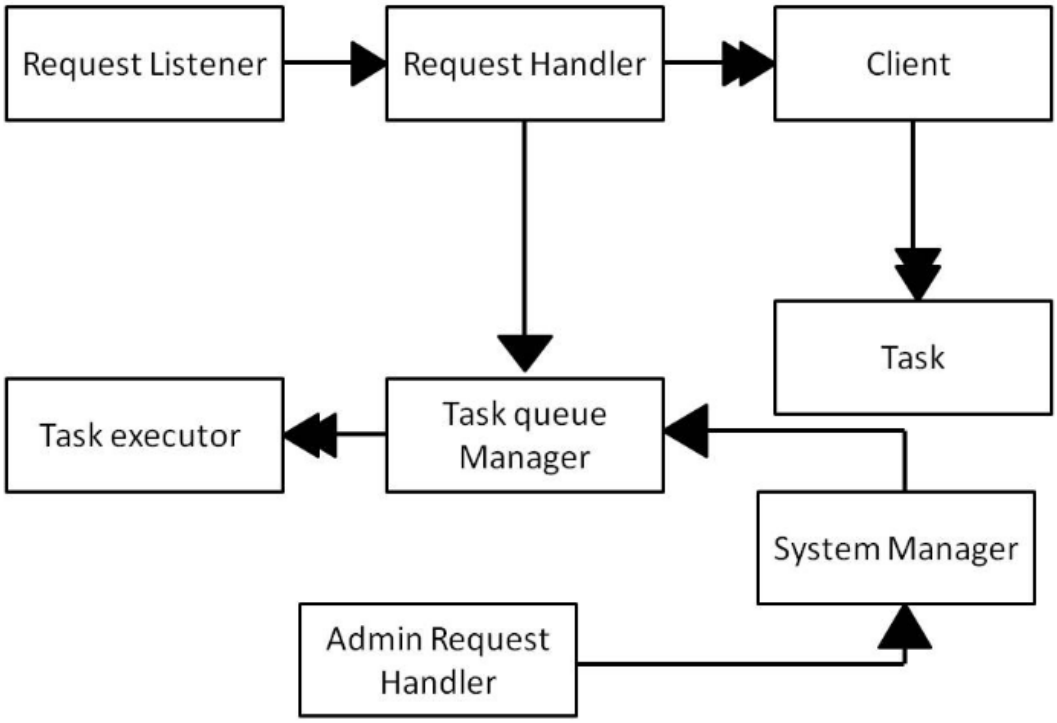

FIGURE 4 – Diagramme de l'architecture de la deuxième version du système

Pour cela, nous ajouterons l'objet "AdminRequestHandler" qui offrira un port d'écoute dédié aux requêtes d'administration. Ces dernières agiront directement sur l'objet systemManager qui sera chargé de mettre en place les modifications au niveau des dockers et d'en informer le "taskQueueManager", afin qu'il puisse répartir convenablement les tâches. Ce système sera bien entendu protégé par un système d'authentification.

5 Conclusion

La mise en œuvre de ce projet nécessitera la mise en application ainsi que d'approfondissement de nos connaissances en réseau. Ce projet apporte également une notion que nous n'avons pas eu l'occasion de traiter au cours des années précédentes, à savoir le fonctionnement et la conception d'une API distante.

Pour venir à bout de ce projet, il nous faudra non seulement avoir recours à la programmation objet, avec une utilisation possible de divers patterns (strategy, observer, etc.), mais aussi nous intéresser à l'algorithmique, afin d'optimiser la distribution des tâches aux serveurs notamment pour l'implémentation de la deuxième ou quatrième stratégie. Nous aurons également recours à de la programmation système pour les serveurs de compilation/execution (gestion des threads, des processus et communication inter-processus).