Concurrency design pattern

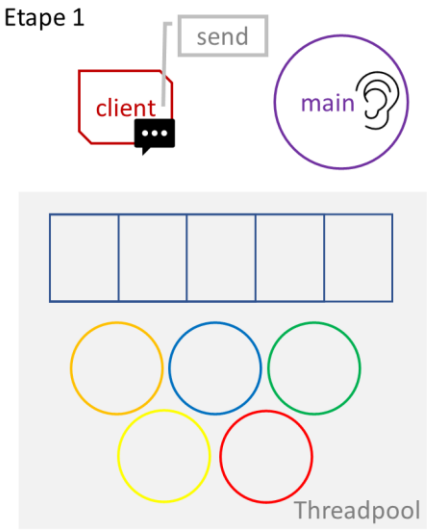
I. Objectif

On va implémenter deux serveurs web, qui diffèrent par leur gestion des requêtes. Un serveur qui attend une requête, la gère, répond puis attend pour une autre requête fonctionne mais ne servira à rien en conditions réelles : si on lui envoie beaucoup de requêtes, il sera incapable de tout traiter.

II. Thread pool

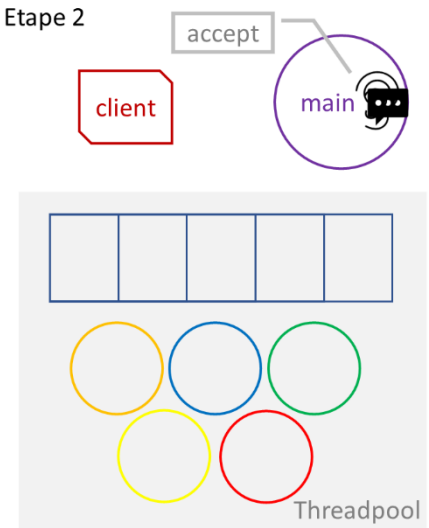
Principe

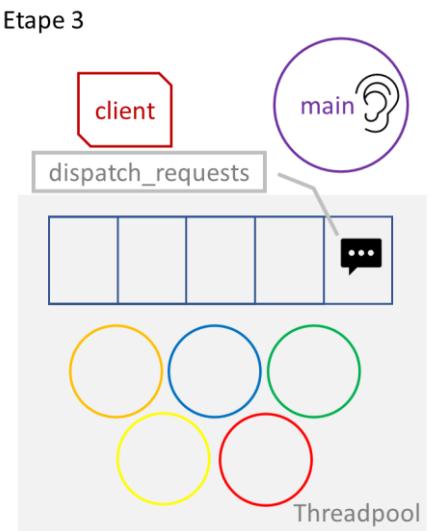
Une première idée est d'avoir plusieurs threads qui gèrent les requêtes. C'est toujours le même thread main qui accepte les sockets, mais il envoie ensuite la requête au thread pool. Le processus est détaillé ci-dessous.



Etape 1 : le client envoie une requête par la fonction send.

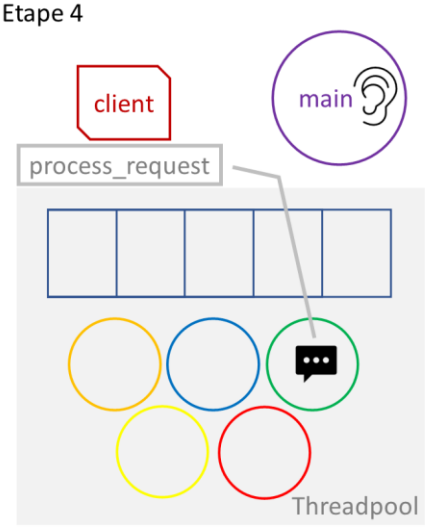
Etape 2 : le thread principal, qui exécute la fonction main, attend qu'une socket arrive. Il reçoit la requête avec la fonction accept.





Etape 3 : avec la fonction dispatch_requests, le thread principal donne la requête au thread pool. La requête (ou plutôt la fonction process_request, avec comme paramètre la requête) est placée dans la pile FIFO du pool, en attente.

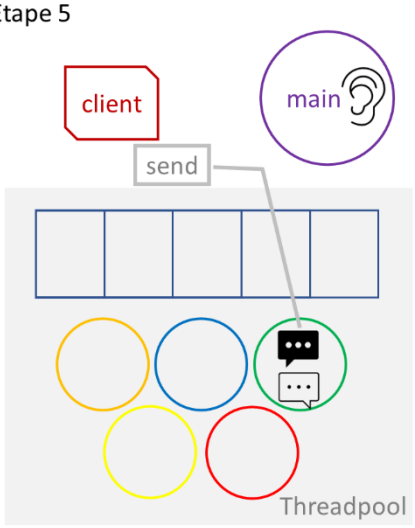
Etape 4



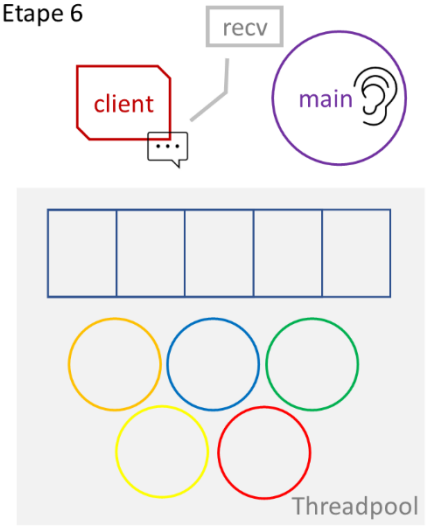
Etape 4 : un des threads du pool est en attente, et prend en charge la tâche qui est première dans la FIFO. Le thread exécute donc process_request sur la requête du client.

Etape 5 : au cours de l'exécution de process_request, le thread en charge crée une réponse et l'envoie au client avec la fonction send.

Etape 5



Etape 6



Etape 6 : le client reçoit la réponse à sa requête, avec la fonction recv.

Limites

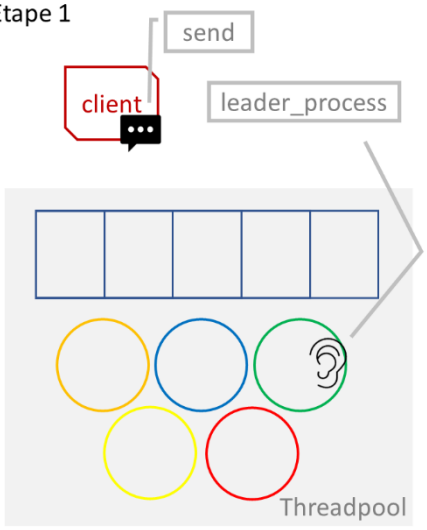
Avec cette méthode, on a toujours au minimum deux threads, même si le serveur web est peu appelé et ne fait pas de calculs pour répondre à une requête.

III. Leader / Followers

Principe

Cette fois on peut n'avoir qu'un seul thread. Le thread principal n'est pas fixé, le rôle peut être passé d'un thread à l'autre. C'est le thread leader qui attend pour une requête, et qui la prend en charge. Quand il reçoit une requête, il donne le rôle de leader à un thread en attente, devient follower et exécute process_request. Le processus est décrit plus en détail ci-dessous.

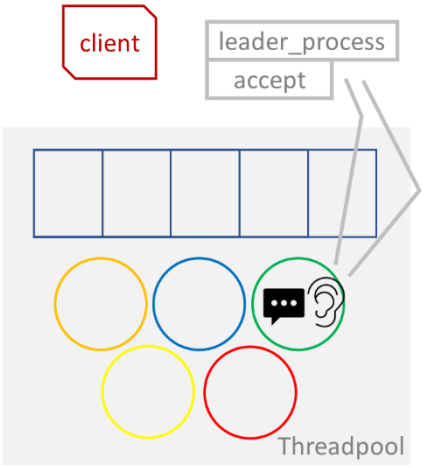
Etape 1



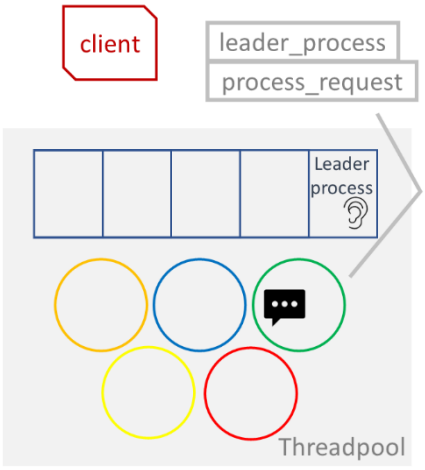
Etape 1 : le client envoie une requête par la fonction send.

Etape 2 : le thread leader, qui exécute la fonction leader_process, attend qu'une socket arrive. Il reçoit la requête avec la fonction accept.

Etape 2

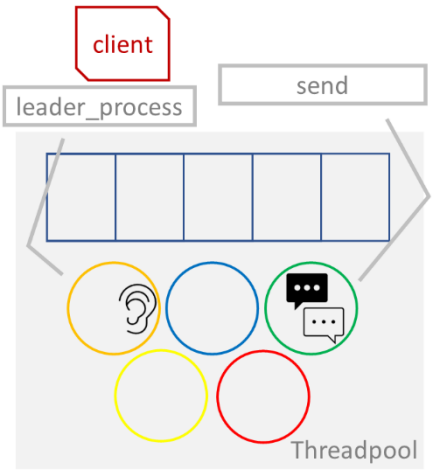


Etape 3

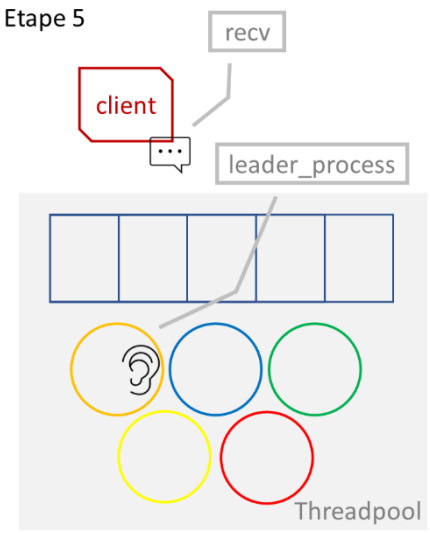


Etape 3 : le thread leader continue d'exécuter leader_process. Il met la tâche « exécuter leader_process avec pour argument threadpool » dans la pile FIFO du threadpool. Il devient donc follower et propose à un autre thread de devenir leader. Il exécute process_request sur la requête du client.

Etape 4



Etape 4 : un autre thread, ici le jaune, est disponible et prend donc la première tâche de la pile FIFO. Il devient donc leader et attend de recevoir une requête. Pendant ce temps, le thread vert continue d'exécuter process_request et envoie une réponse au client avec la fonction send.



Etape 5 : le client reçoit la réponse avec la fonction recv. Le thread orange est toujours en attente d'une requête. Le thread vert a terminé sa tâche et passe en mode idle.

IV. Benchmarking

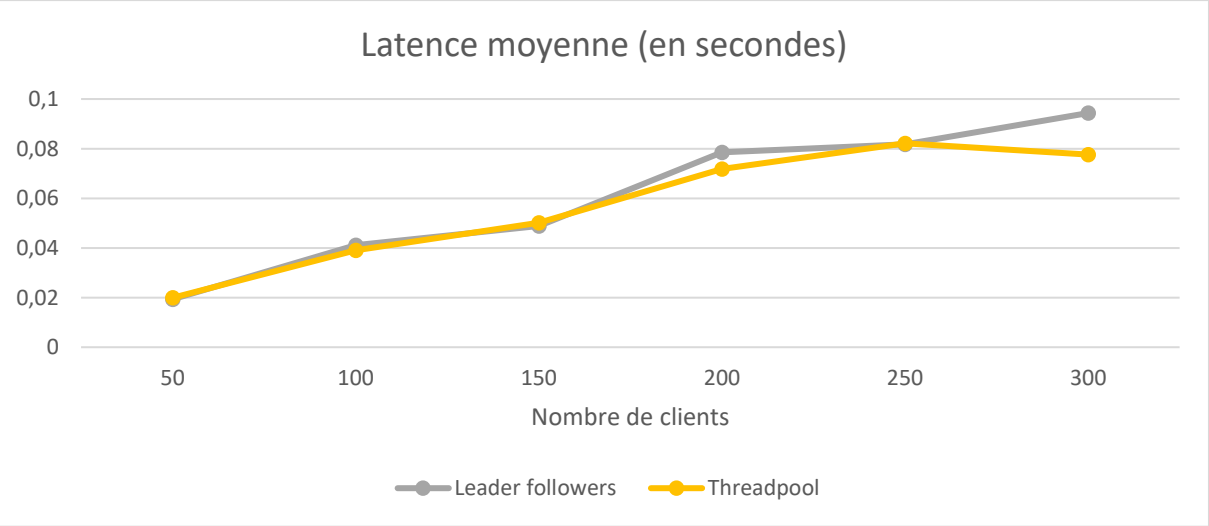
Nous avons testé les deux algorithmes en lançant N clients, chaque client envoyant une seule requête. Nous avons fait varier N entre 50 et 300, par pas de 50.

Les résultats ci-dessous ont été calculés en faisant la moyenne de 5 essais, c'est-à-dire en lançant 5 fois 50 clients, puis 5 fois 100 clients… Les capacités de nos ordinateurs ne nous permettant pas d'aller au-delà.

Les spécifications du test sont dans les fichiers *client_bench1.sh* et *basic_client_func.sh*.

Latence moyenne

Comme on peut le voir sur la première figure, la latence moyenne du client a tendance à augmenter avec le nombre de requêtes servies, ce qui est attendu. Le nombre de points de mesure ne nous permet pas d'en déduire une loi statistiquement viable mais on peut extrapoler une régression linéaire : $y = 0.015x + 0.008$ ($R^2 = 0.9597$) pour leader-followers, $y = 0{,}0126x + 0{,}0129$ ($R^2 = 0{,}9131$) pour threadpool.



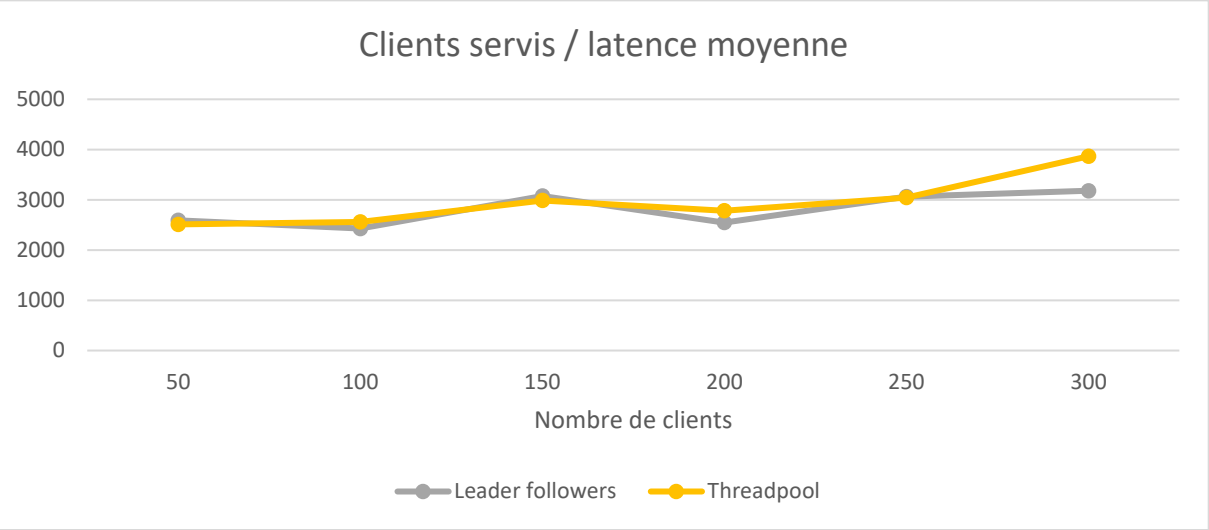
Le fait que l'ordonnée à l'origine soit positive peut apparaître comme la latence incompressible du service des requêtes, mais l'on s'attendrait plutôt à ce qu'elle soit négative, ce qui traduirait que la latence de la file d'attente n'apparaît que pour un certain nombre de clients nécessaires pour maintenir la file d'attente non vide.

Cependant au vu des très faibles valeurs (et surtout des fortes variations d'une simulation à l'autre) on ne peut rien en conclure sur la performance réelle du serveur.

Clients servis

Nous avons également regardé une approximation du nombre de clients servis par seconde défini comme suit : nombre de clients servis au total divisé par la latence moyenne.

On obtient un résultat assez constant, très similaire pour les deux protocoles.



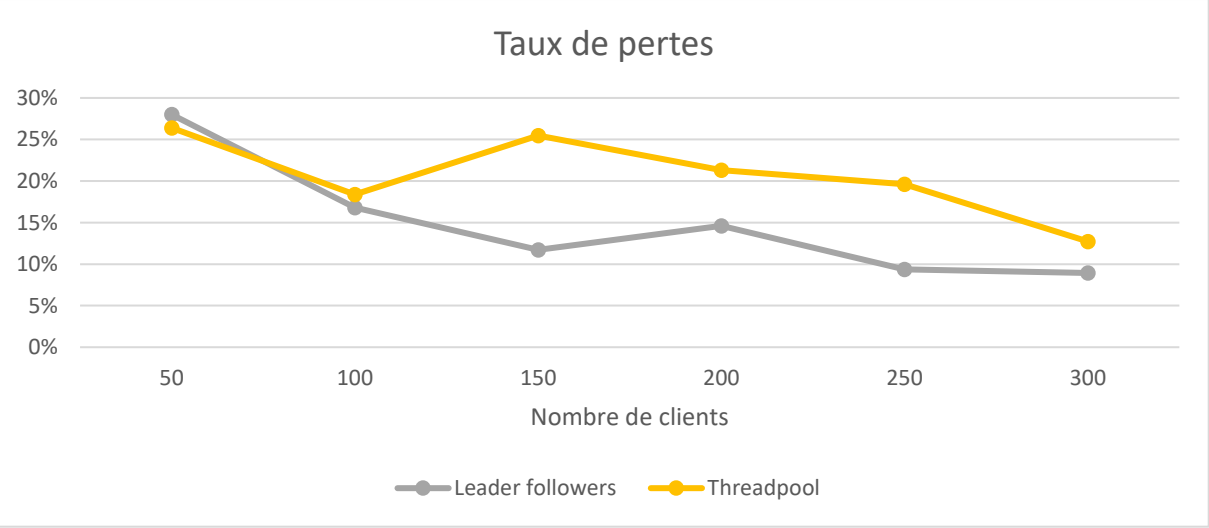
Taux de pertes

Enfin nous avons observé que certaines requêtes ne recevaient jamais de réponse (du moins dans un temps raisonnable de quelques secondes à une minute) ; ceci se traduit par le fait qu'un client est lancé mais ne termine jamais, et est donc considéré comme perdu.

Nous ne pouvons à notre niveau trouver la source de ces disparitions ni leur impact sur la performance mesurée. Nous soupçonnons que ce phénomène soit dû à un buffer overflow, soit au niveau de la queue du serveur soit au niveau système.

Nous avons calculé notre taux de pertes et nous avons vérifié que celui-ci reste dans des marges acceptables.

Comme le montre le graphe suivant, le protocole threadpool a un taux plus élevé mais qui suit la même tendance que dans le protocole leader followers. Étrangement le taux de pertes diminue quand on sert plus de clients ; on s'attendait au contraire.



Remarques

Il est à noter que ces résultats dépendent plus de l'environnement d'exécution que de l'algorithme du serveur lui-même. En effet nos résultats varient beaucoup en fonction de l'ordinateur utilisé et du contexte (des autres processus lancés en même temps).

Ces conditions ne nous ont malheureusement pas permis de mettre au jour les capacités limites des serveurs, puisque le facteur limitant s'est avéré être l'environnement.

Pour implémenter un banc de test rigoureux il aurait fallu à la fois

- mettre en place un parc de machines/cœurs lançant un certain nombre de clients en même temps,
- relier ces machines par un réseau qui soit assez performant pour ne pas créer de goulots d'étranglement,
- s'assurer que les threads créés dans le threadpool ont chacun un cœur exclusif pour tourner purement en parallèle.

Cette étude a néanmoins permis de mettre en évidence deux paradigmes différents et potentiellement complémentaires pour implémenter un serveur web.

Au niveau de programmation auquel agissent ces paradigmes il est tout à fait naturel que les performances d'une architecture dépendent essentiellement du matériel sur lequel elle tourne.