

Concurrency design pattern

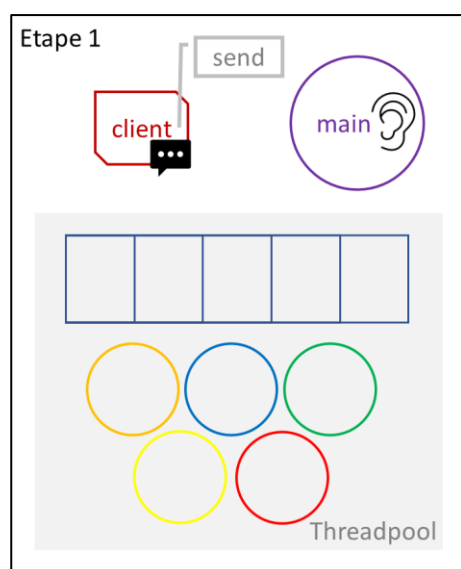
I. Objectif

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.

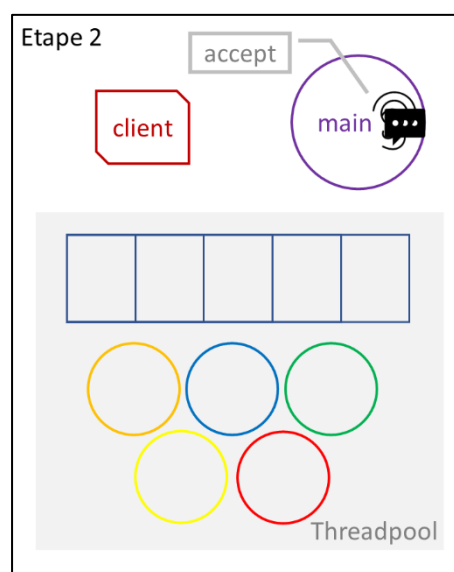
II. Thread pool

Principe

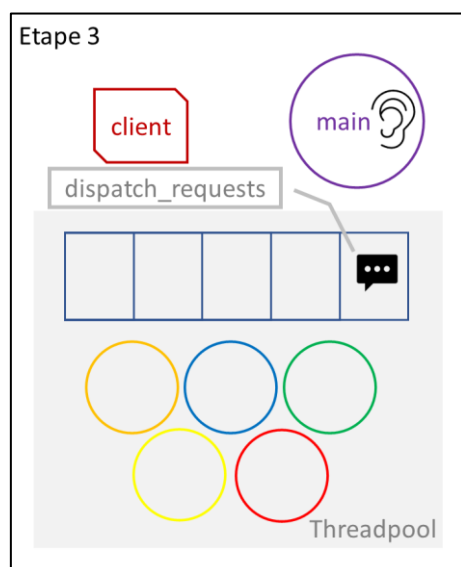
Une première idée est d'avoir plusieurs threads qui gèrent les requêtes. C'est toujours le même thread main qui accepte les sockets, mais il envoie ensuite la requête au thread pool. Le processus est détaillé ci-dessous.



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

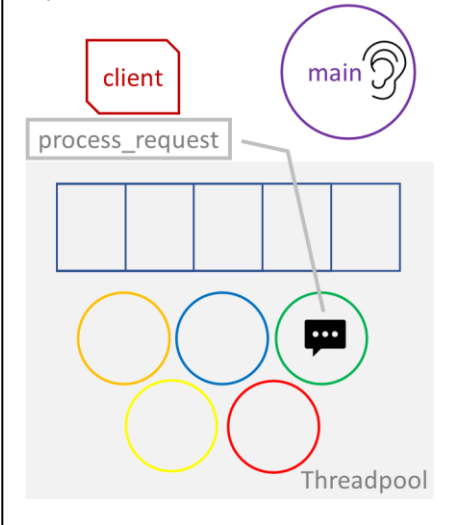


Etape 2 : le thread principal, qui exécute la fonction main, attend qu'une socket arrive. Il reçoit la requête avec la fonction accept.



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.

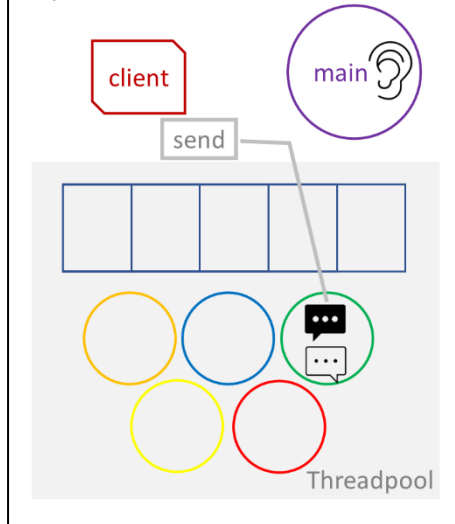
Etape 4



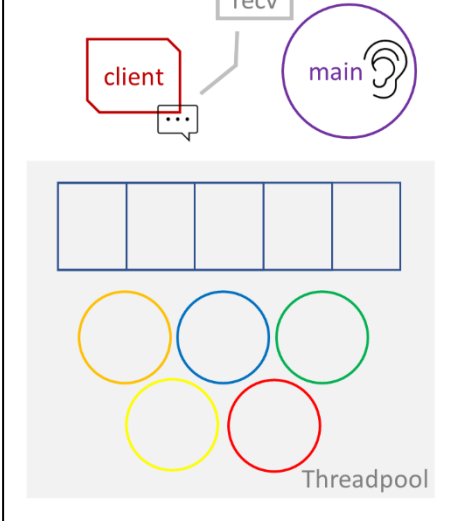
Etape 4 : un des threads du pool est en attente, et prend en charge la tâche qui est première dans la FIFO. Le thread exécute donc `process_request` sur la requête du client.

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`.

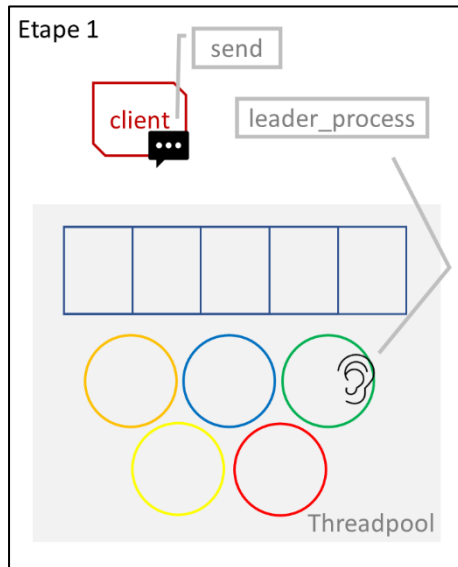
Limites

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

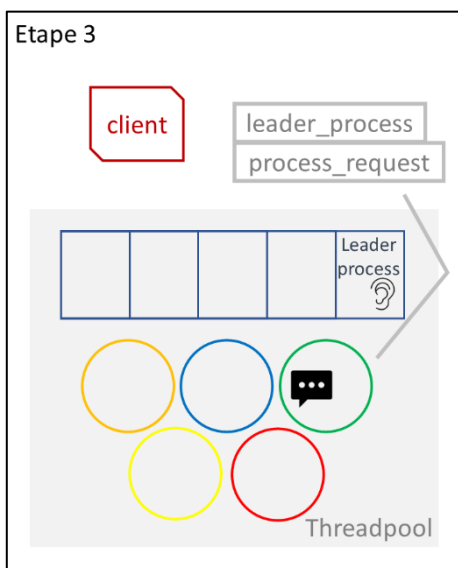
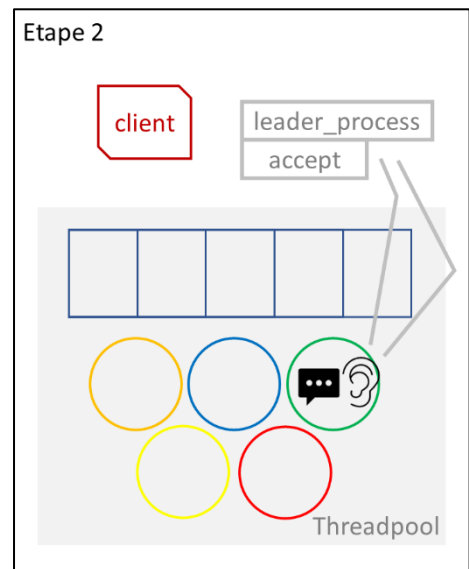
Principe

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.



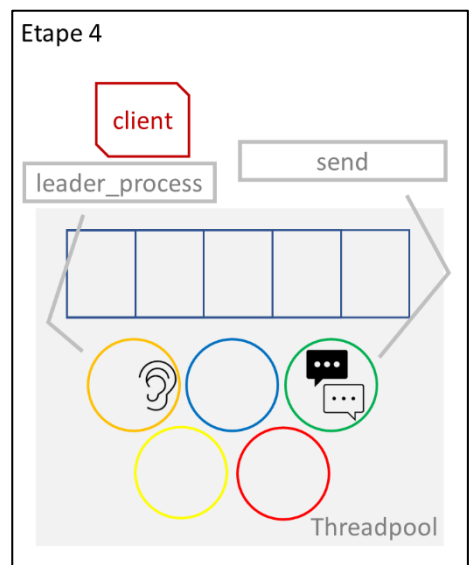
Etape 1 : le client envoie une requête par la fonction `send`.

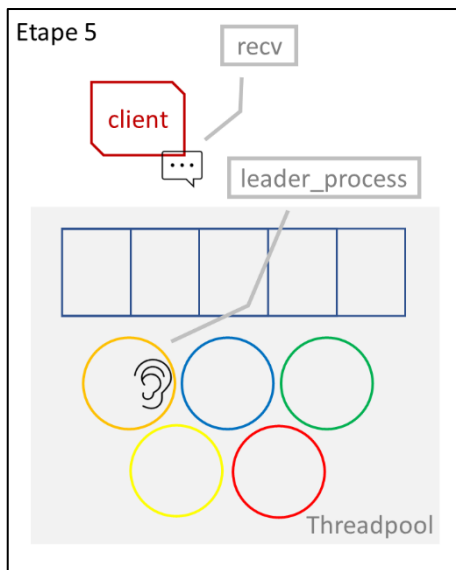
Etape 2 : le thread leader, qui exécute la fonction `leader_process`, attend qu'une socket arrive. Il reçoit la requête avec la fonction `accept`.



Etape 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 : 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

Comparaison des deux algo