# Concurrency design pattern

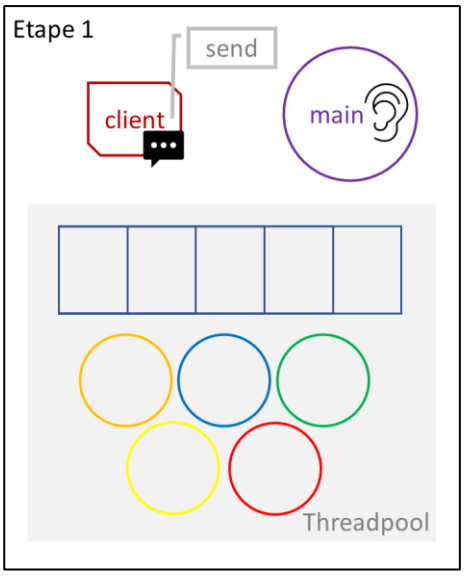
## I. Objectif

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

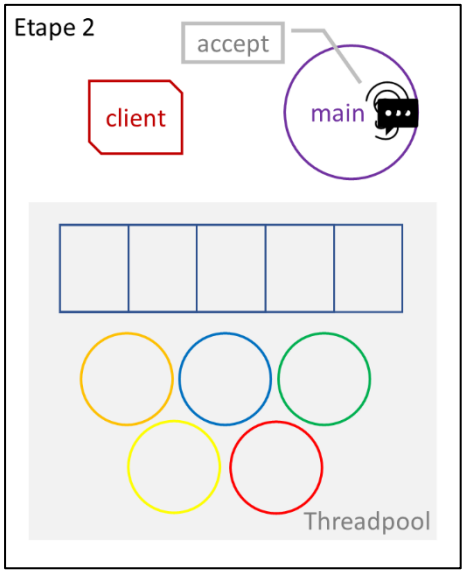
### Principe

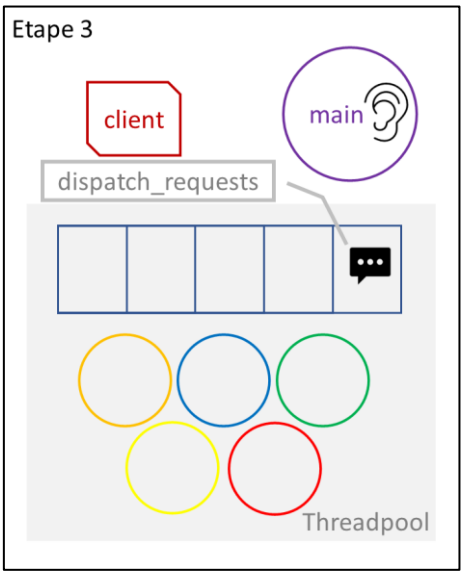
Une première idée est d'avoir plusieurs threads qui gèrent les requêtes. C'est toujours le même thread main qui accepte les sockets, mais il envoie ensuite la requête au thread pool. Le processus est détaillé ci-dessous.



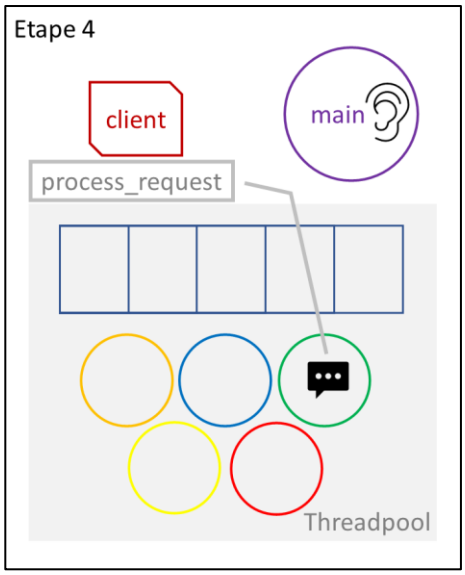

Etape 1 : le client envoie une requête par la fonction send.

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



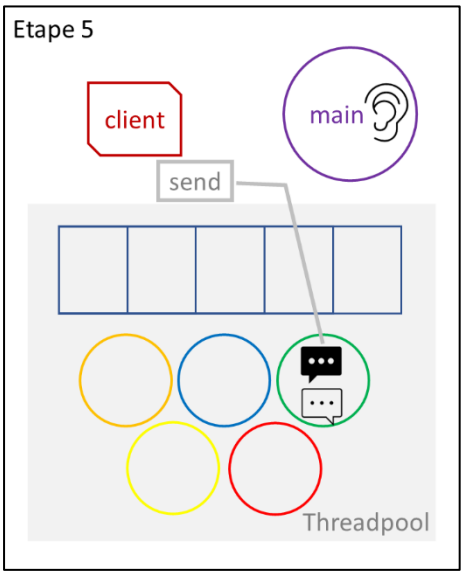



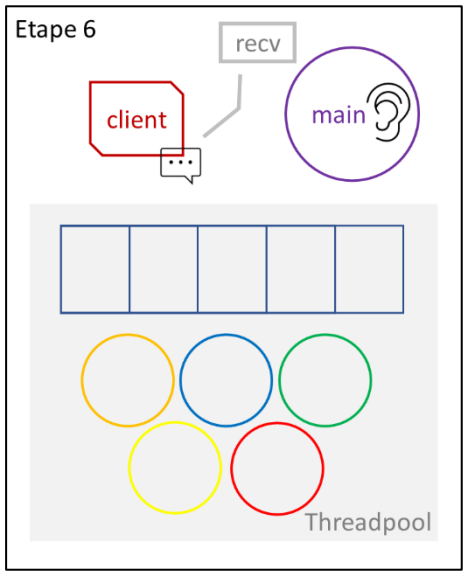

Etape 3 : avec la fonction dispatch_requests, le thread principal donne la requête au thread pool. La requête (ou plutôt la fonction process_request, avec comme paramètre la requête) est placée dans la pile FIFO du pool, en attente.




Etape 4 : 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 6 : le client reçoit la réponse à sa requête, avec la fonction recv.

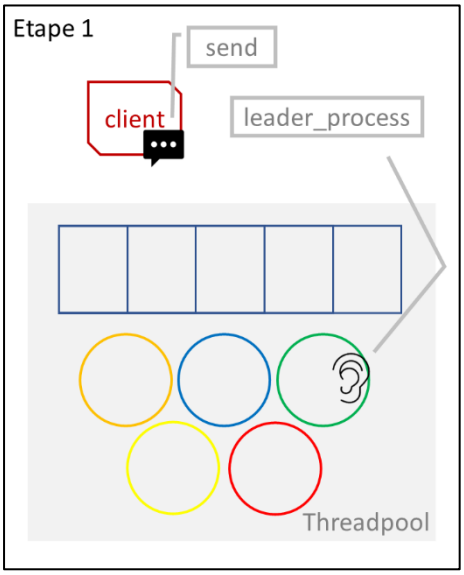
## Limites

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

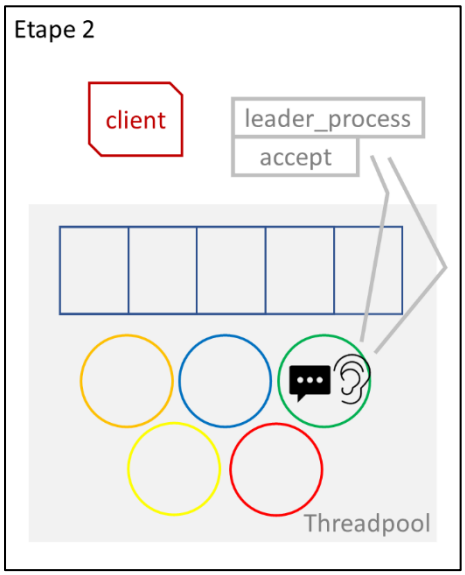
## Principe

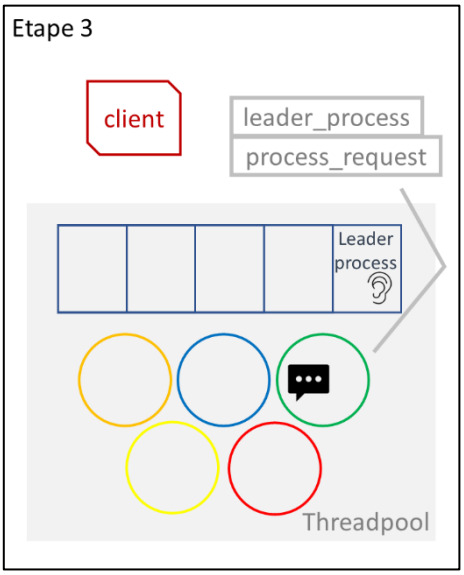
Cette fois on peut n'avoir qu'un seul thread. Le thread principal n'est pas fixé, le rôle peut être passé d'un thread à l'autre. C'est le thread leader qui attend pour une requête, et qui la prend en charge. Quand il reçoit une requête, il donne le rôle de leader à un thread en attente, devient follower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



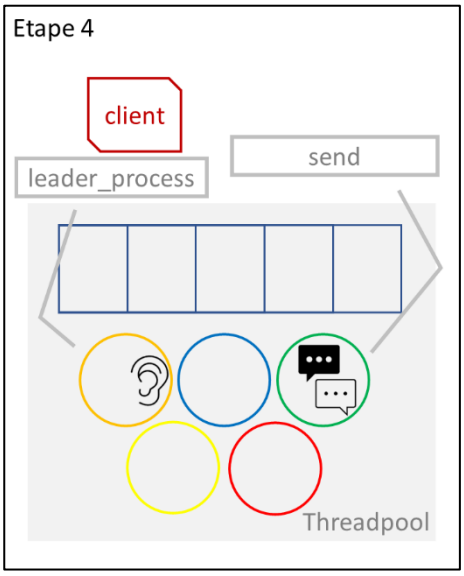

Etape 1 : le client envoie une requête par la fonction send.

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

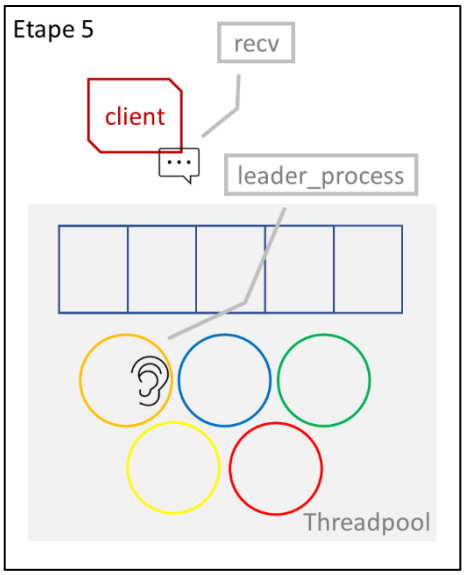



Etape 3 : le thread leader continue d'exécuter leader_process. Il met la tâche « exécuter leader_process avec pour argument threadpool » dans la pile FIFO du threadpool. Il devient donc follower et propose à un autre thread de devenir leader. Il exécute process_request sur la requête du client.




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.

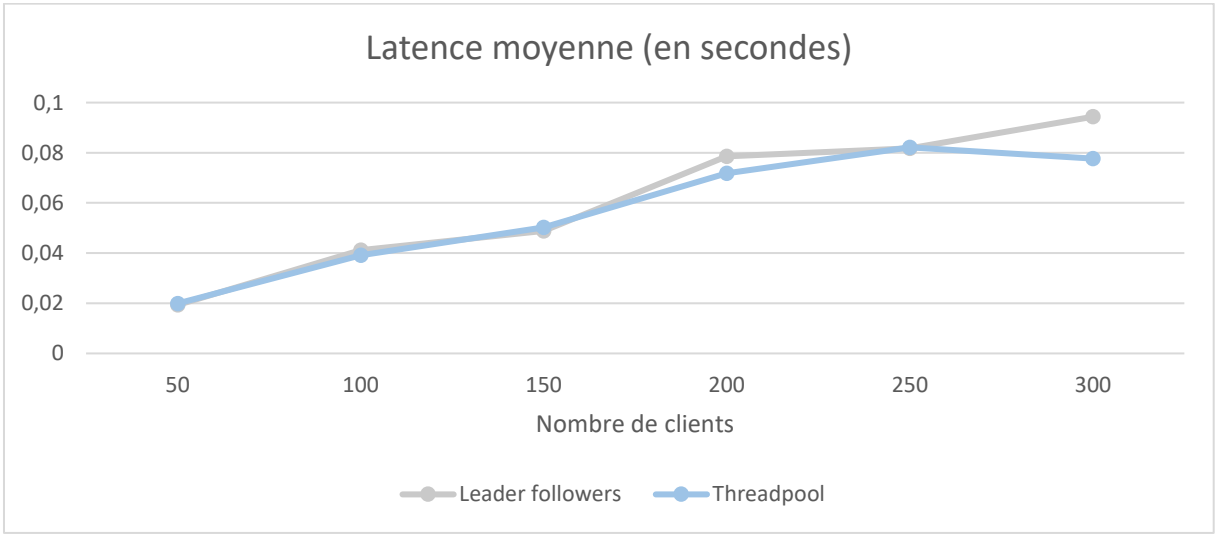



Etape 5 : le client reçoit la réponse avec la fonction recv. Le thread orange est toujours en attente d'une requête. Le thread vert a terminé sa tâche et passe en mode idle.

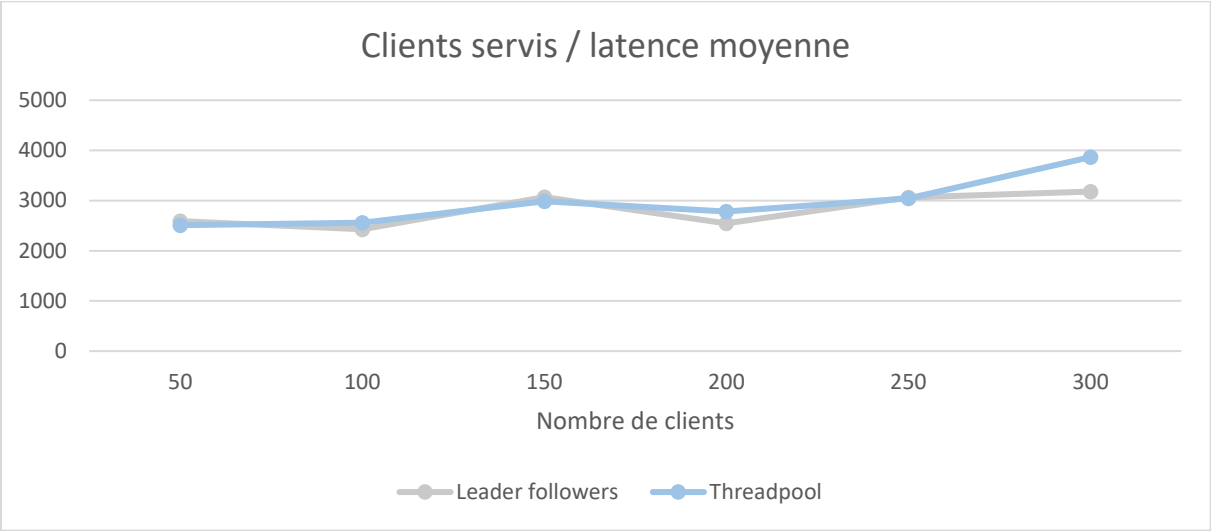
# IV. Benchmarking

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. Nos résultats varient beaucoup en fonction de l'ordinateur utilisé, et nous obtenons sensiblement la même chose pour les deux algorithmes.

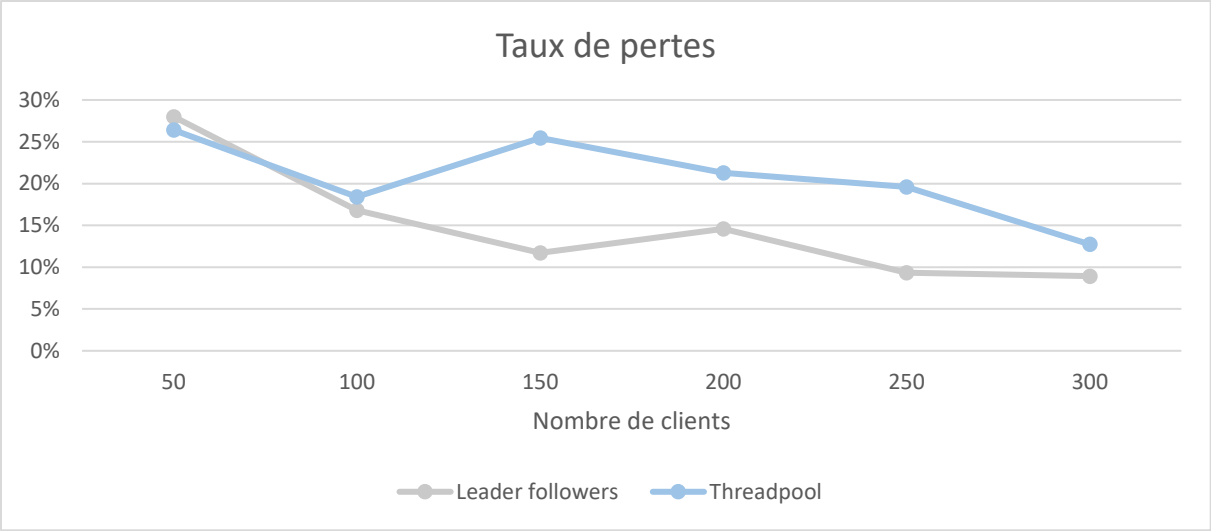
La latence moyenne du client a tendance à augmenter avec le nombre de requêtes servies ; le nombre de points de mesure ne nous permet pas d'en déduire une loi statistiquement viable mais on peut extrapoler une régression linéaire : $y = 0.015x + 0.008$ ($R^2 = 0.9597$) pour leaders followers, $y = 0,0126x + 0,0129$ ($R^2 = 0,9131$) pour threadpool.



Nous avons également regardé une approximation du nombre de clients servis par seconde, en calculant le nombre de clients servis divisé par la latence moyenne. On obtient un résultat assez constant, très similaire pour les deux protocoles.




Enfin nous avons regardé notre taux de pertes : le protocole threadpool a un taux plus élevé mais qui suit la même tendance que dans le protocole leader followers. Etrangement le taux de pertes diminue quand on sert plus de clients, on s'attendait au contraire.




Ces résultats 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… Nos ordinateurs n'étaient pas capables de faire plus de 100 essais sans ramer beaucoup trop. De plus, nous avons obtenus ces résultats avec l'ordinateur d'Arthur ; sur l'ordinateur de Tianchi on avait un taux de perte qui montait jusqu'à 80%, sans raison.

En conclusion, on observe assez peu de différences entre les deux protocoles, mais on observe beaucoup de différences entre l'ordinateur du CI, celui d'Arthur et celui de Tianchi. Nos données sont donc assez peu fiables…