

Laboratoire 5 : Serveur - Phase 2

Auteurs : Oussama Lagha et Adam Zouari

Date : 8 Juin 2018

But du laboratoire

Ajout d'un threadpool avec une taille maximale afin d'éviter les phénomènes néfastes liés au lancement d'un thread par requête, l'idée est d'allouer dynamiquement des threads au fur et à mesure que cela devient nécessaire jusqu'à atteindre la taille maximale, si on atteint la taille maximale on doit réutiliser les threads qui ont déjà fini leur travail.

Choix du mécanisme

Durant cette phase du laboratoire, nous avons fait le choix d'utiliser Mesa afin d'implémenter le mécanisme de threadpool et on a amélioré le mécanisme implémenté durant la phase 1 pour le `producerconsumerbuffer` pour traiter le cas de la taille maximale.

Implémentation

Nous avons commencé par améliorer la classe `producerconsumerbuffer` pour traiter le cas où on veut ajouter dans le tampon et qu'il est plein, les threads ne doivent pas attendre.

On a ensuite implémenté le mécanisme de threadpool qui va gérer un nombre de thread fixe pour protéger le serveur. Le threadpool gère aussi le recyclage des threads une fois qu'ils sont en état de waiting. On a pu gérer correctement la destruction des threads en utilisant la méthode `requestInterruption()`.

On a ajouté aussi au niveau de la classe `producerconsumerbuffer` la fonction `tryPut(T item)`, cette fonction teste si le tampon est plein ou non, si il y a encore de la place elle va ajouter dans le tampon sans soucis et finit par retourner `true` sinon elle retourne `false` et rejette la request.

Enfin, on a adapté le fonctionnement du code de traitement des requêtes entrantes au niveau du thread principal dans le fichier `fileserver.cpp` en utilisant `tryPut()` au lieu de `put()`.

Reponses aux questions

La taille de pool optimale est (selon mes recherches [source](#)) influencé par le nombre de processeurs N , l'utilisation souhaitée des processeurs (entre 0 et 1) et du rapport entre le temps d'attente et le temps de calcul.

$$N_{\text{threads}} = N_{\text{cpu}} \times U_{\text{cpu}} \times (1 + W/C)$$

Dans notre application nous avons décidé d'ignorer le rapport entre le temps d'attente et le temps de calcul et d'utiliser nos processeurs à 100%. Cela revient au nombre de processeurs.

Desormais, quand on inonde le serveur, le temps de réponse est plus grand mais le serveur ne crash pas, la stabilité est donc meilleure.

Les threads occupent tout les processeurs mais n'utilisent pas plus de memoire que la version 1.

Nous avons réitéré le test de l'etape 1 avec 10000 requetes, et le serveur est toujours up.

The screenshot displays a development environment with three main components:

- Top Left (Code Editor):** Shows the source code of a C++ application. The code defines a thread pool, a dispatcher, and a server that listens for WebSocket connections. It includes headers for `ThreadPool`, `Dispatcher`, and `Server`.
- Top Right (Web Client):** A browser window showing a web application interface. It has a text input field with the URL `ws://localhost:1234` and a button labeled "Number of requests" set to 1000. Below the input, it displays the content of a file named `shakespeare.txt`, which contains text from King Henry IV.
- Bottom (System Monitoring):** A terminal window showing the output of the `top` command. It displays system statistics such as CPU usage (100%), memory usage (100%), and a list of running processes. The processes include `radame@xpsradame`, `radame`, `root`, `avahi`, `htop`, `chromium`, `telegram`, `terminator`, `docker-containerd`, and `docker`.