Eric Larsen
Jean-Philippe Raymond
IFT2245
25 mars 2015

Travail pratique 2
Rapport d'activité

Le serveur

L'application serveur exécute d'abord la lecture du fichier de configuration. Celui-ci précise notamment certains paramètres contrôlant les propriétés de la génération des requêtes de même que l'identifiant des interfaces de connexion ('sockets') et du canal de transfert ('pipe') utilisés. Noter que le réglage de la taille de la file d'attente des interfaces de connexion ('socketBackLogSize') et du nombre de fils d'exécution ('serverThreads') doivent être ajustés en correspondance avec le nombre de fils d'exécution du client afin d'éviter l'engorgement des interfaces. Le serveur procède ensuite à l'initialisation des variables statiques qui permettent la description complète de son état initial et sa mise à jour ultérieure en fonction du traitement par ses fils d'exécution des requêtes émanant des fils d'exécution du client qui transitent par les interfaces de connexion. Cette initialisation comporte notamment la génération aléatoire des ressources totales disponibles et des ressources maximales exigibles par les fils d'exécution du client. Un paramètre spécifié dans le fichier de configuration permet pour plus de souplesse d'ajuster les propriétés du générateur aléatoire qui dépend par ailleurs du nombres de clients. Les fils d'exécution sont ensuite créés et démarrés, chacun de ceux-ci étant muni du code permettant l'écoute sur l'interface de connexion et le lancement du traitement des requêtes du client au fur et à mesure de leur arrivée.
L'essentiel du développement de la classe serveur réside dans la construction de la méthode *processRequest* qui assure le traitement des requêtes. Celle-ci fait d'abord appel à la méthode *parseCheckRequest* pour analyser la requête reçue, déterminer son intelligibilité, son admissibilité et son sens (celle-ci doit soit requérir des ressources ou soit offrir de les libérer). Sur la base de cette analyse préliminaire, la requête subit dans *processRequest* une analyse logique détaillée qui détermine son acceptation, son rejet ou sa mise en attente. Noter que conformément au traitement exigé, la variable de comptabilité *countDispatched* n'est incrémentée que lorsqu'un client a retourné toutes ses ressources après en avoir consommé une quantité non nulle au préalable. *processRequest* fait appel à des méthodes de calcul et de vérification auxiliaires dont la plus importante pour la prévention des interblocages ('deadlocks') causés par la concurrence pour les ressources est la méthode *isNotSafe*. Celle-ci implémente l'algorithme de sûreté du banquier et est appliquée à toute demande de prélèvement de ressources avant son autorisation. On y représente l'état nouveau du serveur dans l'hypothèse où la requête serait acceptée au moyen des tableaux d'allocation et de disponibilité hypothétiques *HypoAllocation* et *HypoAvailable*, de sorte que la sûreté d'une demande de ressources peut être vérifiée tout en réduisant la période de blocage des variables *Allocation* et *Available* qui décrivent l'état présent du système. Dans le cas où la requête examinée n'est pas sûre, le message retourné au client indique un temps d'attente à respecter avant de la soumettre à nouveau. Ce temps d'attente est fonction du temps total d'exécution de la dernière requête et du nombre de fils d'exécution de l'application client.
Puisque la méthode *processRequest* est appelée simultanément par les fils d'exécution du serveur pour traiter les requêtes émanant du client et que les variables d'état de même que les variables de comptabilité sont concurrement mises à jour, il importe d'éviter une concurrence critique ('race condition') en synchronisant soigneusement les lectures et les écritures de ces variables au moyen de verrous. Les verrous du serveur implémentent la technique RAII ('Resource Acquisition Is Initialisation') qui est conçue pour prévenir les blocages survenant lorsque l'interruption d'un programme empêche leur ouverture, qui est largement utilisée en pratique et qui est appliquée ici à titre d'exercice d'apprentissage. Elle repose sur la définition d'une classe de verrouillage, ici la classe interne *LockGuard*, dont le constructeur définit et applique le verrou qui sera automatiquement relâché par le destructeur à la sortie du bloc où son instanciation survient.
Lorsque les requêtes se tarissent, les fils d'exécution du serveur s'interrompent et celui-ci communique avec le client par l'intermédiaire d'un canal de tranfert ('pipe') pour signifier la fin de ses travaux. En dernière étape de son fonctionnement, l'application serveur affiche et sauvegarde les résultats de son exécution, libére la mémoire utilisée puis s'interrompt.

Le client

L'application client procède d'abord à la lecture du fichier de configuration commun qui comporte notamment le nombre de fils d'exécution du client et le nombre fixe de requêtes que chacun doit soumettre au serveur. Les fils sont sont ensuite créés et lancés, chacun muni de la méthode *clientThreadCode* qui gère l'expédition du nombre de requêtes en créant pour chacune une interface de connexion ('socket') distincte. La méthode *sendRequest* est responsable de leur expédition. Ces requêtes sont générées pseudo-aléatoirement par les méthodes auxiliaires *randomAllocations* et *randomReleases* et respectent notamment les contraintes suivantes : elles ne peuvent excéder les maximums de ressources octroyés au fil d'exécution du client, la première requête présentée ne peut être une libération, la dernière requête doit être une demande de libération totale, une libération totale n'est jamais demandée avant la dernière requête, une requête nulle ne peut être que la dernière. En outre,

la méthode *sendRequest* tient la comptabilité à jour en appliquant les verrous appropriés pour éviter la concurrrence critique et recueille les réponses en provenance du serveur et les retransmet à *clientThreadCode* qui gère notamment les délais d'attente indiqués par le serveur. Il s'agit dans le cas de l'application client de verrous standard. Les fils d'exécution du client se terminent individuellement lorsque les requêtes qu'ils ont expédiées ont obtenu leurs réponses et ont terminé la mise à jour de la comptabilité qui les concerne. Contrairement au serveur, l'architecture du client est ainsi conçue que les fils d'exécution ne sont pas bloquants. Le processus de l'application client est donc passé à l'exécution de la méthode *waitUntilServerFinishes* alors même que les fils s'exécutent et pourrait poursuivre avec l'affichage et la sauvegarde des résultats de la comptabilité s'il n'était arrêté par l'instruction *sleep* originellement contenue dans *waitUntilServerFinishes*. Le remplacement de l'instruction *sleep* par l'attente bloquante pour un message de fin d'exécution provenant du serveur dans la méthode *waitUntilServerFinishes* est donc nécessaire pour généraliser l'application du serveur à une durée d'exécution quelconque de ses fils d'exécution. En dernière étape de son fonctionnement, l'application client affiche et sauvegarde les résultats de son exécution, libére la mémoire utilisée puis s'interrompt.

La prévention des blocages

D'une part, les risques de blocage de l'application pourraient émerger de la concurrence pour l'utilisation des ressources qui sont l'objet des requêtes du client L'utilisation de fils d'exécution multiples, notamment sur le serveur, multiplie ces risques en raison de la simultanéité. Pour prévenir toute possibilité de ce type de blocage, les ressources offertes par le serveur ne sont accordées que si leur octroi ne peut conduire qu'à un état sûr. C'est un objectif primordial de la méthode *acceptRequest* et de sa méthode auxiliaire *isNotSafe*. En outre, des découplages entre la vérification et l'octroi de ressources sont soigneusement évités au moyen de verrous afin d'empêcher que des vérifications successives entre deux fils du serveur conduisent à des octrois successifs de ressources qui négligeraient leur effet cumulatif. En d'autres termes, la vérification de la sûreté d'une requête et son octroi proprement dit appartiennent toujours à une même région critique connexe. D'autre part, des blocages pourraient accidentellement émerger du mésusage des verrous qui contrôlent l'accès aux variables d'état et de comptabilité. Ici encore, l'utilisation de fils d'exécution multiples, notamment sur le serveur, multiplie ces risques. Pour minimiser les risques de blocage de ce type, les verrous ont été soigneusement disposés de sorte qu'ils ne puissent faire l'objet d'une attente réciproque par plusieurs fils d'exécution concurrents et les applications ont été soumises à une batterie de tests simulant des conditions d'utilisation anormales.

La corruption des données

D'une part, les interfaces de connexion ('socket') sont bien entendu vulnérables aux erreurs de transmission lors de véritables communications à distance sur des réseaux. Des erreurs du matériel peuvent également survenir à toutes les autres étapes du traitement et des échanges. Deux types de solutions minimisant les risques et les impacts de ces erreurs sont envisagebles : Premièrement, faire usage d'un protocole de connexion muni d'un contrôle des erreurs, par exemple TCP. Deuxièmement, concevoir l'application de sorte à mener une détection systématique des messages erronés et à implémenter les plans de contingence appropriés. C'est ce qu'accomplissent la méthode *sendRequest* du client et les méthodes *acceptRequest* et *parseCheckRequest* du serveur. Par exemple, ces dernières distinguent une pléthore de cas pathologiques : message entièrement inintelligible, dont seul le processus d'origine est lisible, dont la requête est mixte et donc irrecevable, qui propose une libération à la première requête, une demande à la dernière requête, une libération insuffisante à la dernière requête, qui survient au-delà de la dernière requête prévue. D'autre part, une concurrence critique ('race condition') incontrôlée entre les fils d'exécution du serveur ou encore entre les fils d'exécution du client peut aussi être cause de corruption des variables partagées qui décrivent leur état et leur comptabilité respectifs. L'utilisation systématique de verrous pour isoler et protéger les régions critiques des applications serveur et client permet de prévenir ces occurrences.

La synchronisation du client

Un canal de transfert dédié ('named pipe') est utilisé afin de communiquer au client le message de fin des travaux du serveur. Ce canal se présente comme un fichier dans lequel la méthode *signalFinishToClient* du serveur écrit son message et la méthode *waitUntilServerFinishes* du client le lit. Tel que convenu, il est identifié par la chaîne qui identifie le numéro de port utilisé pour les interfaces de connexion et qui est spécifiée dans le fichier de configuration. Ce canal est instancié dès la mise en branle du serveur dans sa méthode *createAndStart* afin de garantir sa préexistence lorsque le client a terminé l'expédition de ses requêtes et boucle sur des tentatives de lecture du message sur le canal. Contrairement au serveur, l'architecture du client est ainsi conçue que ses fils d'exécution ne sont pas bloquants. Le processus principal de l'application client ('*server.cpp'*) passe donc à l'exécution de la méthode *waitUntilServerFinishes* alors même que les fils s'exécutent et pourrait même poursuivre avec l'affichage et la sauvegarde de résultats bien entendu hâtifs et incorrects de la comptabilité s'il n'était arrêté par l'instruction *sleep* originellement contenue dans *waitUntilServerFinishes*. Le remplacement dans cette méthode de l'instruction *sleep* par l'attente bloquante pour un message de fin d'exécution provenant du serveur permet donc de généraliser le domaine d'application du serveur à une durée d'exécution indéterminée de ses fils d'exécution.