Cas pratique: communications S2S

Objectifs de l'exercice

- Voir et pratiquer les patrons de conception en dehors d'un cadre magistral
- Étudier un cas réel pour en extraire une architecture
- Appliquer une méthode SOLID sur un cas réel
- Le prototypage comme preuve de validation des designs

Sujet

Vous faites partie d'une équipe technologique chargée par votre employeur de réaliser une librairie facilitant les communications S2S (Server-to-Server) au sein de l'entreprise. Cette librairie se doit évidemment d'être portable sur les différentes plateformes de l'entreprise Windows et Linux (moderne). Les différents corps de métier de l'entreprise sont composés en grande majorité de développeurs système et outils utilisant couramment le C++ moderne et le C#.

Les communications S2S

Les communications S2S (Serveur à Serveur) sont vitales pour toutes les entreprises modernes dans le domaine de l'informatique. Elles permettent à 2 serveurs de s'échanger des informations. Avec l'avènement du cloud computing, des micro-services et des backend serverless, elles sont devenues la colonne vertébrale de nombreux services connectés.

Cette famille de communications partage de nombreux points communs avec les communications client-serveur standard. Néanmoins certains attributs les rendent très spécifiques :

1. **L'élasticité** : les communications doivent toujours fonctionner qu'il y ait 10 messages par seconde à envoyer ou 10 millions. L'infrastructure vous supportant a été conçue pour cela et va, elle aussi, grandir avec les besoins (mémoire, nombre de thread, stockage, etc.).
2. **La variété des canaux :** la plupart des cloud fonctionnent avec un protocole http(s) pour faire communiquer leurs serveurs entre eux. Néanmoins, beaucoup permettent aussi l'envoi et la réception de messages par d'autres type de canaux. Parmi eux on trouve le Web Socket des protocoles de messagerie ou même de la mémoire partagée entre différents processus.
3. **La faible variété des plateformes :** contrairement aux applications clientes, la grande majorité des communications S2S se déroulent sur des serveurs. Les systèmes d'exploitation peuvent varier mais de manière générale il y a peu de chance que vous manquiez de bande passante, de mémoire ou de puissance de calcul.
4. **La latence :** quand il s'agit de communications entre serveurs, l'échelle de mesure de la latence passe de la milliseconde à la microseconde. Si la page de votre client doit être affichée en 40ms

vous ne pouvez passer tout ce budget latence dans des communications entre les différents composants de votre application côté serveur.

Cet ensemble de points à retenir est assez concis et loin d'être exhaustif. Nous étudierons ces communications et comment les mettre en place plus tard.

Votre mandat

Votre entreprise est parfaitement au courant que votre équipe est bien trop petite pour porter un tel projet jusqu'en production voire même en live. Néanmoins elle pense que votre taille vous donne l'agilité nécessaire pour produire un design qui va, plus tard, permettre d'étendre et de supporter de manière efficace cette librairie. Par conséquent votre travail est de produire le design général de l'application au format UML standardisé ainsi qu'un prototype démontrant son applicabilité. Vous avez deux semaines pour arriver avec un prototype fonctionnel.

Le prototype

Beaucoup d'architectes sous-estiment la valeur d'un prototype quand il s'agit de design. Il est vrai qu'un prototype n'a presque aucune valeur pour démontrer l'extensibilité ou la facilité de maintenance d'une librairie devant des comités d'architectes. Néanmoins, si deux projets d'architectures sont équivalents en qualité de design, la plupart des entreprises et décideurs choisiront toujours la version qui a un prototype. En effet, en codant votre prototype, qui est loin d'avoir les meilleures performances ou le plus beau des codes, vous avez démontrer une chose : votre design n'est pas qu'un gribouillis d'architecte qui justifie son salaire devant son chef, il fonctionne.

Qu'est-ce qu'un prototype? Ce n'est pas une application finale possédant un joli 100% de couverture en tests unitaires, ayant un pipeline complet d'utilisation voire même toutes les fonctionnalités désirées. C'est une application de test permettant de démontrer que toutes les parties du design sont applicables facilement en termes de programmation. Évidemment elle doit compiler et ne pas crasher si une démo doit être faite devant les décideurs. Par conséquent, gardez à l'esprit qu'un prototype bien fait et plus petit à souvent plus de valeur qu'un monstre essayant de couvrir toutes les fonctionnalités mais plantant dans les premières 30 secondes d'une démonstration devant durer 2 heures.

Détails de l'exercice

Votre application doit pouvoir supporter un nombre infini de files d'exécution (thread), permettre d'ajouter de nouvelles façons de communiquer facilement, exposer une API propre et facile à utiliser mais puissante.

Construisez toujours une application de bas en haut. Commencez par les couches de bas niveau puis bâtissez la au fur et à mesure jusqu'à remonter à l'API publique. Dans votre cas, commencez par le système supportant le multithreading. La façon la plus simple de fournir une fonctionnalité de multithreading dense (*dense multithreading*) est de bâtir un système de tâches basé sur le patron **Command** et utilisant un **CommandExecutor** qui peut exécuter des tâches sur différents threads. Exposez

une interface propre et facile à utiliser qui permet de facilement créer des tâches, les ajouter à la file d'exécution et d'avoir, s'il existe, le résultat de celle-ci de manière asynchrone. Souvenez-vous c'est un composant bas niveau vous êtes donc votre propre utilisateur, facilitez-vous la tâche avec une interface propre et extensible. Est-ce que le patron **Singleton** s'applique à votre système de tâches? Si oui, pouvez vous citer une solution qui n'utilisera pas ce dernier (bonus)? Comment prévenir l'utilisateur que la tâche est terminée. Proposez au moins un patron qui s'applique parfaitement à ce cas très précis. À la fin de cette étape vous devriez avoir un design UML d'un système de tâches permettant d'enfiler des tâches et d'avoir le résultat de celles-ci. Évidemment le nombre de thread est à spécifier à l'exécution.

Une fois votre base solide, passez au système de communication qui se situe au-dessus. Dans le cadre de cet exercice, nous allons rester simple et simplement afficher les données dans une console mais rien n'empêche de bien concevoir notre système. On doit supporter plusieurs canaux de communication différents mais conserver la même interface. En effet, que ce soit à travers la mémoire, le réseau ou même votre console, vous devez autoriser la création de flux d'information de la même manière. Pour cela commencez par créer une interface représentant le concept d'un protocole de communication. Elle doit permettre d'envoyer et recevoir de l'information de manière non bloquante. Pour ce faire, l'interface doit être en mesure de créer des flux de données. Ces derniers, peuvent envoyer et recevoir de l'information en clair, chiffrée ou compressée. Évidemment, il est possible de créer un flux chiffré et compressé. Quel patron va-vous aider à construire des flux basés sur un protocole tout en gardant toujours la même interface? Quel patron utiliser pour composer les flux afin de créer des flux chiffrés et compressés? Est-ce que votre instance de protocole de communication devrait être un flux? Pourquoi?

Pour finir, exposez une API simple permettant de bien configurer votre librairie. Vous devez permettre de créer une instance de votre librairie en lui spécifiant des paramètres :

- Nombre de thread
- Protocole de communication à utiliser
- Propriétés des flux (clairs, chiffrés, compressés)

Quel patron va-vous permettre de construire votre instance en permettant de facilement ajouter des paramètres de construction plus tard? Appliquez-le à votre design. Une fois l'instance créée, elle doit permettre (tout en gardant toujours la même interface) d'envoyer et recevoir de l'informations de manière asynchrone. Est-ce que cette instance de classe devrait être un singleton? Pourquoi?

Conseils

- Faites votre design au complet pendant que je suis là pour vous aider là-dessus
- Le logiciel StarUML est facile à utiliser pour faire des diagrammes UML
- Utilisez un langage dans lequel vous êtes à l'aise
- N'hésitez pas à poser des questions, ce n'est pas facile

Rendu

- Le rendu est à faire avant le 10 février 2020 à 23H59

- Le rendu doit être fait en utilisant Git
- Les équipes doivent être composées de 2 à 3 personnes
- Rendez-moi sur votre Git :
    - Votre prototype compilable sur ma machine (attention aux librairies)
    - Votre design en images ou au format StarUML
    - Un court rapport (~1 page) expliquant votre design et comment utiliser votre librairie
- **N'oubliez pas de m'envoyez vos Git et équipes avant la date du rendu !!!!!**
- **Votre prototype doit compiler sur ma machine (Linux ou Windows ça dépend de mon humeur)**

Pour le prototype

- Ne vous embêtez pas à faire de vrais protocoles de communication. Affichez simplement le message sur la console après un délai de 500ms.
    - BONUS : implémentez le protocole Web Socket pour démontrer que votre librairie fonctionne réellement
- Ne vous embêtez pas avec de la cryptographie ou des algorithmes de compression affichez simplement en début du message les mots **CHIFFRÉ** ou **COMPRESSÉ**
- Je vous demande cependant de faire le système de tâches au complet avec des vrais threads
- C# ou C++17
- Git obligatoire
- Si je vous demande un patron dans le sujet ce n'est pas juste pour poser la question : appliquez-le