Banque de données distribuée avec notions de respect de la qualité privée des données et tolérante aux pannes

1 Description

On désire concevoir une banque de données distribuée offrant une certaine protection des données privées et tolérante aux pannes. Cette banque de données est composée de plusieurs serveurs hébergeant les données, ainsi que d'un serveur d'accès, que des clients peuvent joindre afin de lire ou écrire des données.

La banque de données contient des données utilisateurs. Elle possède, pour chaque utilisateur, plusieurs champs les concernant (par exemple, âge, taille, poids, ...). Ces données sont **scindées** en plusieurs groupes. Autrement dit, **aucun serveur ne contient l'intégralité des information sur un utilisateur**.

Pour accéder à toute l'information concernant un utilisateur, il faudrait récupérer les différents champs sur **différents** serveurs de stockage. De plus, **les serveurs sont dupliqués afin que plusieurs serveurs puissent servir une même requête**.

Un serveur est « dépositaire » pour chaque utilisateur d'un **couple d'information** : un identifiant d'utilisateur et **une et une seule** propriété pour cet utilisateur (son nom, son âge, sa couleur préférée, ...). Un **même couple est déposé sur minimum deux serveurs** pour des raisons de redondance et de tolérance aux pannes.

On considérera que le nombre maximum de serveurs constituant la banque de données est faible (de l'ordre de la dizaine par exemple).

Le serveur d'accès est le point central de l'architecture décrite. Un client communique avec ce serveur dans le but de modifier ou lire des données contenues dans la banque de données. C'est également le serveur d'accès qui effectue les requêtes nécessaires auprès des serveurs de données.

Le **serveur d'accès chargera un fichier contenant la définition des différents utilisateurs**, leur mot de passe (ou un hash sur le mot de passe pour plus de sécurité) ainsi que le nom des différents champs auxquels chaque utilisateur aura accès en mode lecture. Chaque ligne de ce fichier d'accès aura le format suivant :

```
<utilisateur>:<mot-de-passe>(:<nom-champ>)*
```

Un utilisateur aura accès en **écriture à ses propres informations uniquement**. Pour les autres utilisateurs, il ne pourra **lire** que les **champs mentionnés dans le fichier d'accès** décrit ci-dessus. Chaque utilisateur a accès en lecture à minimum 0 champs mais peut accéder à plusieurs champs (tous les champs listés dans le fichier de permission). On suppose pour des raisons de simplicité que ce fichier ne contient pas d'utilisateur en double.

Un utilisateur souhaitant lire ou stocker de l'information **s'authentifiera** d'abord à l'aide de son login et mot de passe. Il enverra ensuite une **requête d'accès** au serveur qui la validera si l'utilisateur a les droits requis ou répondra par un message d'erreur dans le cas inverse.

Voici les **différentes commandes** supportées par le serveur d'accès :

— Lire 1 ou plusieurs champs dont les noms sont fournis en arguments.

```
lire (<champ1>)+
```

— Écrire 1 ou plusieurs champs dont les noms et valeurs sont fournies en arguments.

`ecrire (<champ1>:<valeur>)+`

— Supprimer tous les champs relatifs à un utilisateur

`supprimer`

Lorsque l'accès est octroyé, le serveur d'accès enverra différentes requêtes aux serveurs de stockage afin de satisfaire la demande.

La commande de **lecture enverra l'ensemble des valeurs** stockées sur le serveur hébergeant le champ passé en argument. L'**écriture quand à elle ne concerne que l'utilisateur** étant auteur de la requête. Cela signifie que lors d'une écriture le serveur doit communiquer au serveur de stockage l'identité de l'utilisateur qui émet la requête. Cet identifiant est unique et stable dans le temps. Il ne peut pas s'agir de l'adresse IP de connexion du client (càd du logiciel utilisateur).

Une opération de **lecture sera envoyée à un des nœuds** de stockages hébergeant l'information. Par contre les opérations d'**écriture/modification doivent écrire les valeurs demandées sur tous les nœuds** responsables de l'information demandée afin de garder les données consistantes. Il en sera de même pour la suppression de toutes les données d'un utilisateur.

Le serveur d'accès doit donc savoir quel serveur contient quel couple d'information, afin de savoir à qui émettre les requêtes nécessaires.

Au démarrage, chaque nœud de stockage **annonce le champ dont il est responsable**. Le serveur d'accès **maintiendra une liste des nœuds de stockage, ainsi que le champ supporté par le nœud**. Le serveur maintient cette liste à jour. Il supprime les nœuds non joignables et ajoute les nouveaux nœuds. Un nouveau nœud de stockage obtient au démarrage les informations disponibles pour son champ via le serveur d'accès. On suppose qu'un noeud qui se déconnecte perd toutes ses données et les redemande au noeud d'accès.

Pour des raisons de simplicité, les communications ainsi que les données stockées ne seront pas chiffrées. Cela ainsi que le stockage en clair des mots de passe, a bien entendu des conséquences sur la sécurité des données Une telle application ne peut se prétendre sécurisée mais est développée à des fins d'apprentissage uniquement. La notion de donnée privée n'est respectée que par le fait qu'un nœud de stockage n'a pas accès à plus d'un champ si les nœuds sont répartis sur des machines physiques différentes.

La figure 1 illustre un exemple de fonctionnement de l'architecture décrite ci-dessus. Tout échange commence par une authentification du client au serveur d'accès. Ce-dernier vérifie si le client est connu et si le mot de passe est correct.

Le client fait ensuite une demande d'accès afin de consulter le champ "taille" des utilisateurs (en violet sur la figure). Après avoir vérifié si le client a accès à ces données, le serveur valide la requête. Le serveur d'accès consulte ensuite sa liste afin de trouver un serveur ayant accès à la donnée demandée. Il effectue ensuite la requête nécessaire et transmet les données au client.

Ensuite, le client fait une demande d'accès afin de modifier son âge (en vert sur la figure). Le client ayant forcément le droit de modifier ses propres informations, le serveur d'accès envoie directement la demande de modification aux deux serveurs contenant le champ concerné, afin de s'assurer de garder la banque de données dans un état cohérent. Dès que les deux serveurs de données ont confirmé leur modification, le serveur d'accès confirme la modification au client.

L'échange se termine par un message de fin échangé entre le client et le serveur d'accès.

2 Implémentation

Le projet est à réaliser sur les machines de la salle T40 (*Ubuntu18.04*) en langage C à l'aide des *sockets* (vous n'utiliserez pas d'autre mécanisme de programmation réseau).

Pour éviter les délais imposés par TCP lorsqu'un serveur est en panne, vous utiliserez exclusivement UDP et vous lancerez certaines requêtes en parallèle. Comme vous utiliserez UDP, vous prendrez soin de contrôler que les données sont correctement transférées.

La banque de données est à tolérance aux pannes, mais cela ne signifie pas qu'il ne peut pas y avoir

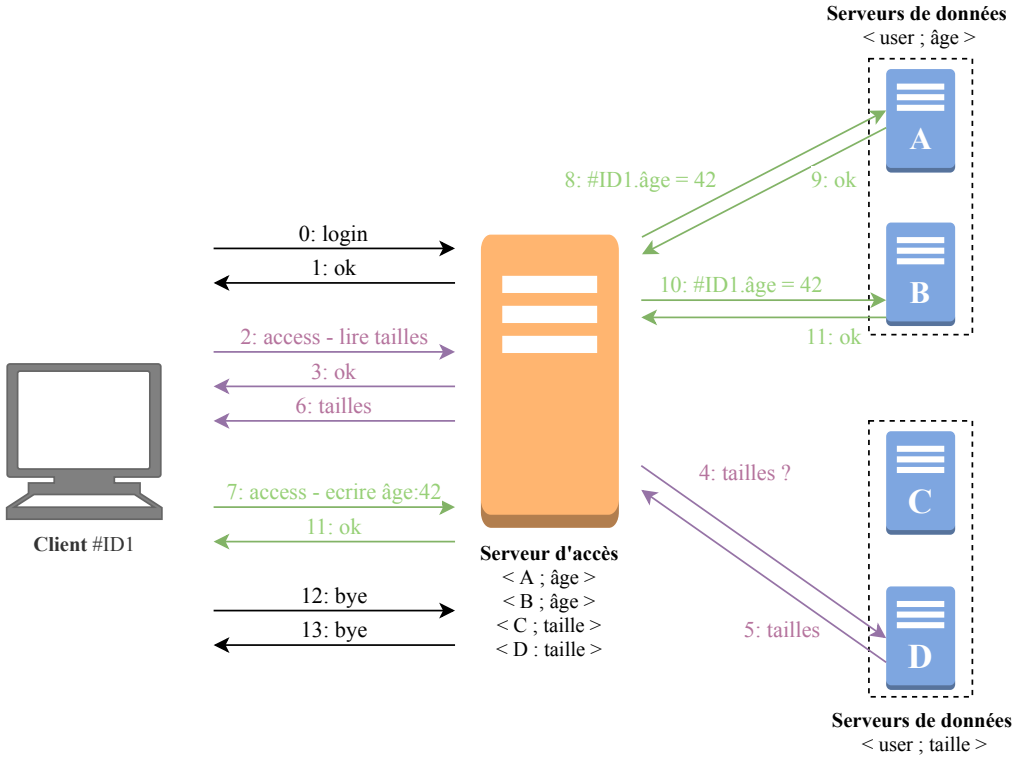

FIGURE 1 – Exemple d'échange lors de requêtes client

des erreurs. En particulier, si une transaction (lecture ou écriture) ne peut être achevée avec succès (par exemple si l'un des deux serveurs contactés tombe en panne lors de l'écriture d'un champ), la commande doit se terminer en indiquant une erreur. Vous prendrez soin toutefois de laisser la banque de données dans un état cohérent.

3 Travail demandé

Il vous est demandé :

- de décrire et de justifier l'architecture générale de la banque de données ;
- de décrire le protocole (échanges, messages, formats des messages, ...) que vous mettez en œuvre entre les différents acteurs, en justifiant vos choix. Votre description doit être suffisamment précise pour permettre à quelqu'un d'autre de programmer individuellement l'un des composants (les commandes des clients, le serveur d'accès, les noeuds de stockage) ;
- de réaliser une documentation utilisateur, en décrivant le format des fichiers utilisés (exlusivement pour les formats non fournis dans cet énoncé) ;
- de programmer les composants (client, serveur d'accès et serveur de données) de l'application en langage C sous Unix, avec les sockets.
- d'indenter et de commenter votre code.

4 Modalités de remise

Ce projet est à réaliser en **binôme ou individuellement**.

Vous déposerez sur Moodle, dans l'espace de devoirs prévu à cet effet, votre projet (documentation, sources, Makefile) sous forme d'une archive au format « *login*.tar.gz » où *login* est votre nom de login sur Turing. Vous prendrez soin de supprimer tous les fichiers binaires (exécutables, objets).

Date limite : le dimanche 17 novembre 2019 à 23 heures.

Sauf contre-ordre, une soutenance aura lieu le 2 décembre.