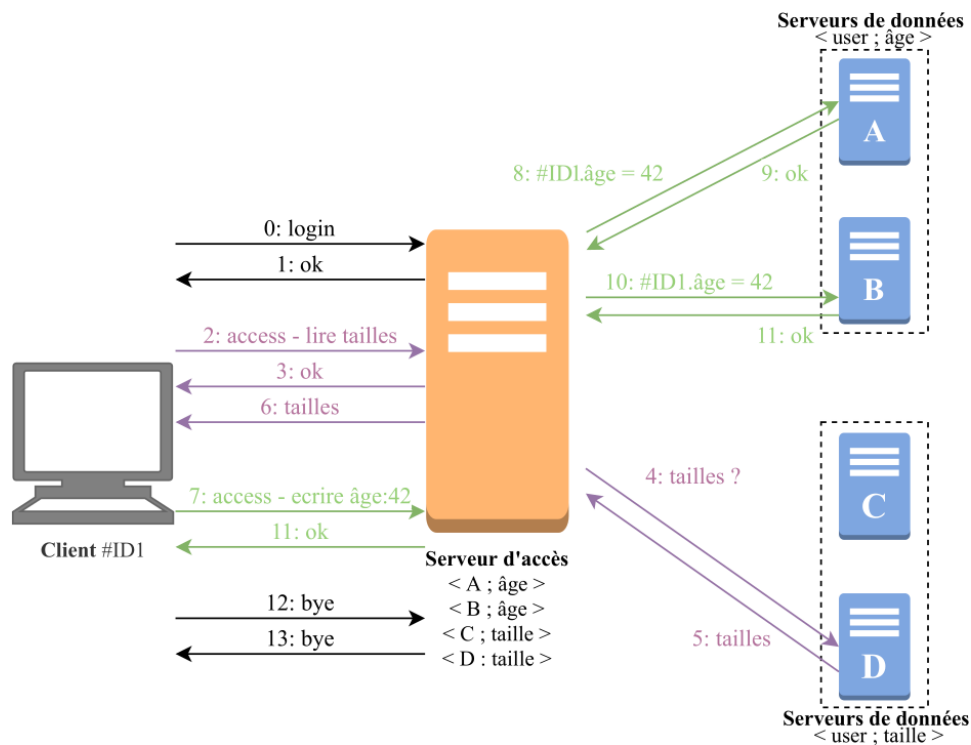


# Rapport du projet de réseau

Danyl El-Kabir et François Grabenstaetter

22 novembre 2019



## 1 Architecture de la banque de données

L'architecture de notre projet contient **4 modules** :

- Le *serveur d'accès* (relais)
- Le *client*
- Le *serveur de données* (nœud)
- Les *outils* réseau partagés (common)

Nous avons rendu possible la communication de multiples clients en parallèle avec le serveur d'accès ainsi que celle des serveurs de données avec le serveur d'accès.

## 1.1 Le serveur d'accès

Le serveur d'accès **stocke une liste** :

- Des *infos clients* (login, mot de passe et champs autorisés en lecture).
- Des *clients connectés* (uniquement ceux dont l'authentification a réussi). On stocke leurs *coordonnées* (IP + port), leur *login*, et la *date* (timestamp) de leur dernier message (déconnexion forcée après **timeout**).
- Des *serveurs de données connectés*. On stocke leurs *coordonnées* (IP + port), le *champ* qu'ils desservent (âge, taille, etc.), un *identifiant* unique qui leur est attribué (utile pour faire correspondre un message reçu avec un serveur de données), la *date* (timestamp) de dernière communication avec ce dernier (retiré de force après **timeout**), ainsi qu'un booléen *active* pour savoir si le serveur de données est en capacité de traiter des requêtes ou non.

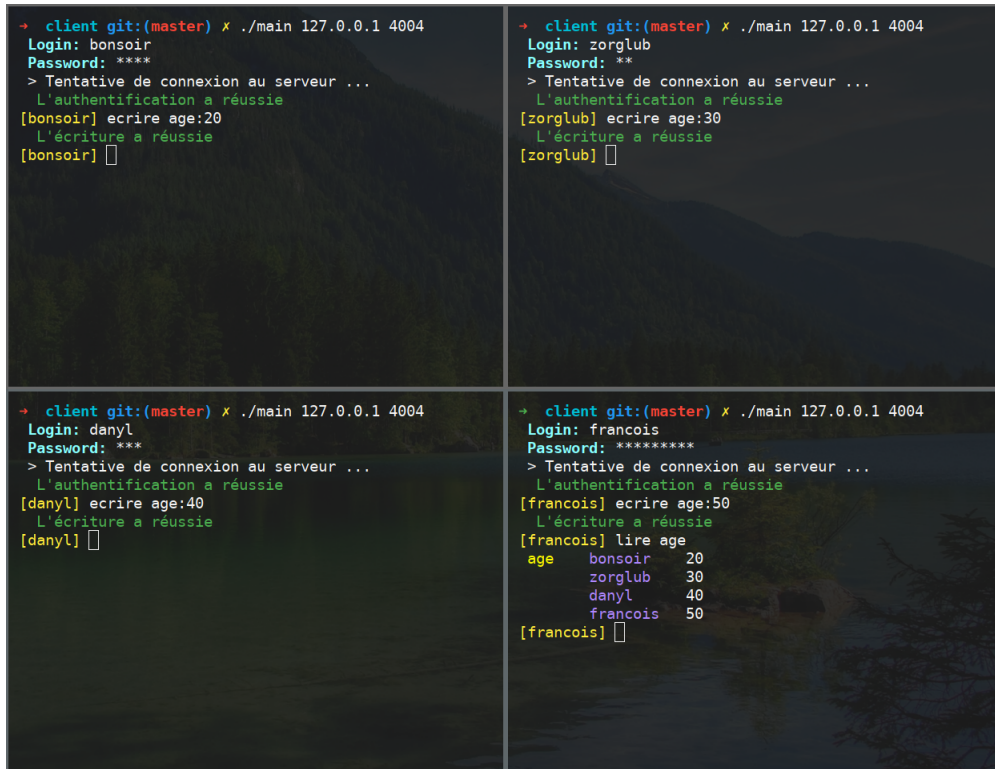
Par exemple, un serveur de données est *inactif* ( $active == false$ ) lorsqu'il vient de se connecter et qu'il y a déjà **un serveur de données de même champ** connecté, le temps que ce serveur de données déjà connecté lui transmette ses données (requêtes SYNC + GETDATA).

- Des *réponses en attente*. Il s'agit de faire correspondre une requête *Serveur d'accès* → *Serveurs de données* avec les réponses à cette requête afin de n'envoyer qu'une seule et unique réponse au client l'ayant engendré en fonction des réponses / non réponses des serveurs de données interrogés (pour les requêtes *écriture* et *suppression*).

On stocke un *identifiant* unique pour la structure de réponses en attente, le *client connecté* ayant engendré la requête (et qui devra recevoir la réponse), le *nombre de requêtes envoyées* ainsi que le *nombre de requêtes reçues*, le *type de requête* associé (écriture ou suppression) ainsi que la *date* (timestamp) de création de la structure (envoi d'une réponse forcée après **timeout**).

Les **timeout** ( $\theta$ ) associés aux clients et aux serveurs de données sont déclenchés après un certain temps  $\phi$ , mais il y a d'abord *envois successifs* de messages **PING** durant un interval de temps  $\mu$ , de sorte que :  $\phi = \lambda + k\mu$ , avec  $\lambda, k \in \mathbb{N}$ .

## 1.2 Le client



The image displays four terminal windows arranged in a 2x2 grid, showing the interaction of a client application with a server. Each window starts with the command `./main 127.0.0.1 4004`. The top-left window shows a successful login for 'bonsoir' with password '\*\*\*\*', followed by writing 'age:20'. The top-right window shows a successful login for 'zorglub' with password '\*\*', followed by writing 'age:30'. The bottom-left window shows a successful login for 'danyl' with password '\*\*\*', followed by writing 'age:40'. The bottom-right window shows a successful login for 'francois' with password '\*\*\*\*\*', followed by writing 'age:50' and then reading the age of all clients, displaying a list: 'age bonsoir 20', 'age zorglub 30', 'age danyl 40', and 'age francois 50'.

```
→ client git:(master) x ./main 127.0.0.1 4004
Login: bonsoir
Password: ****
> Tentative de connexion au serveur ...
  L'authentification a réussie
[bonsoir] écrire age:20
  L'écriture a réussie
[bonsoir]

→ client git:(master) x ./main 127.0.0.1 4004
Login: zorglub
Password: **
> Tentative de connexion au serveur ...
  L'authentification a réussie
[zorglub] écrire age:30
  L'écriture a réussie
[zorglub]

→ client git:(master) x ./main 127.0.0.1 4004
Login: danyl
Password: ***
> Tentative de connexion au serveur ...
  L'authentification a réussie
[danyl] écrire age:40
  L'écriture a réussie
[danyl]

→ client git:(master) x ./main 127.0.0.1 4004
Login: francois
Password: *****
> Tentative de connexion au serveur ...
  L'authentification a réussie
[francois] écrire age:50
  L'écriture a réussie
[francois] lire age
age    bonsoir    20
      zorglub    30
      danyl      40
      francois   50
[francois]
```

Le client a pour tâche principale de *traduire les requêtes* de l'utilisateur (lecture, écriture, suppression, et autres) en requêtes pour le serveur d'accès.

Après une authentification réussie, l'utilisateur peut entrer les commandes suivantes :

- **lire <champ{,...}>** : obtenir les informations de tous les clients pour les champs demandés, si le client en a la permission.
- **écrire <champ1 :valeur1{,...}>** : modifier ses informations pour les champs demandés.
- **supprimer** : suppression de toutes ses informations.
- **aide** : afficher l'aide.
- **clear** : effacer l'écran actuel.
- **bye** : quitter proprement (prévenir le serveur d'accès de sa déconnexion, afin qu'il n'ai pas à attendre le timeout pour forcer la déconnexion).

Nous avons pris soin d'offrir une interface utilisateur simple et efficace à travers des petits "bonus" comme le masquage du mot de passe en astérisques, l'affichage d'un prompt, des couleurs, etc.

La déconnexion "propre" d'un client (bye) est également réalisée lors de la capture du signal **SIGINT**.

### 1.3 Le serveur de données

Le serveur de données doit traiter les requêtes du serveur d'accès et lui envoyer une réponse en fonction de ses données. Pour se faire, il doit donc stocker une liste associant un login à une valeur (donnée), ainsi que le champ qu'il doit gérer (âge, taille, etc.).

La méthode de connexion d'un serveur de données au serveur d'accès est la suivante :

1. Le serveur de données envoie une requête **MEET** au serveur d'accès.
2. Si le serveur d'accès lui répond, le serveur de données a bien été connecté.
3. Si un serveur de données de *même type* existait déjà dans le serveur d'accès, ce dernier va envoyer ses informations au nouveau afin de permettre une *synchronisation permanente* des données au sein de la banque de données.

Une fois connecté au serveur d'accès, le serveur de données peut donc recevoir ses requêtes et les traiter en lui envoyant une réponse. Lors de chaque message reçu, on vérifie que celui-ci *provient bien du serveur d'accès* pour une question de **sécurité**.

### 1.4 Les outils réseau (common)

Les trois modules précédents partagent tous la même **API** réseau, qui a pour but de :

- *Unifier* l'envoi et la réception de données au bas niveau ainsi que d'autres utilitaires comme l'attente (sck.c).
- *Fournir et automatiser* la "couche réseau" séparément de la couche principale des différents modules, afin qu'ils n'aient pas à gérer les mécanismes et problèmes susceptibles d'arriver lors des communications : il leur suffit d'appeler une fonction qui s'occupe de tout, par exemple :

```
1 int dgram_create_send (const int sck, dgram **dgsent, dgram **dgres, const uint16_t id,  
2                       const uint8_t request, const uint8_t status, const uint32_t addr,  
3                       const in_port_t port, const uint16_t data_size, const char *data);
```

En utilisant cette API, les **principaux mécanismes** ci-dessous sont *automatisés* :

- Gestion des *pertes*.
- *Transformation* des données brutes en structures "dgram" (parser).
- Envoi d'un *ACK* pour valider la réception d'un paquet.
- Assurer l'*intégrité des données*.
- *Renvoi* des données si pas de ACK reçu après timeout.
- S'assurer que le message est "entier" (si non entier, attente des autres morceaux).

Nous avons également pris soin d'implémenter un **système de débogage** en affichant les messages *entrants* (IN) et *sortants* (OUT) en affichant les données traduites en langage humain des en-têtes et des données :

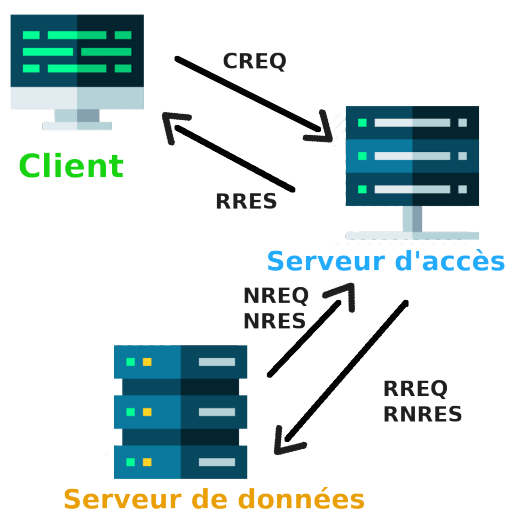
```
>> [IN] DGRAM {id=307, request=RREQ_READ, status=NORMAL, data_size=12, data_len=12, checksum=7714
      data=francois:age
<< [OUT] DGRAM {(ACK pour) id=307, request=ACK, status=NORMAL, data_size=0, data_len=0, checksum=0
      data=
<< [OUT] DGRAM {id=5, request=NRES_READ, status=SUCCESS, data_size=51, data_len=51, checksum=48858
      data=francois:bonsoir:20,zorglub:30,danyl:40,francois:50
>> [IN] DGRAM {(ACK pour) id=5, request=ACK, status=NORMAL, data_size=0, data_len=0, checksum=0
      data=
>> [IN] DGRAM {id=313, request=PING, status=NORMAL, data_size=0, data_len=0, checksum=0
      data=
<< [OUT] DGRAM {(ACK pour) id=313, request=ACK, status=NORMAL, data_size=0, data_len=0, checksum=0
      data=
>> [IN] DGRAM {id=212, request=PTMG, status=NORMAL
```

## 2 Protocoles utilisés

### 2.1 Types de messages et status

Nous avons choisis de séparer les *types de messages* en 4 catégories :

1. Client → serveur d'accès (CREQ).
2. Serveur d'accès → serveur de données (RREQ et RNRES).
3. Serveur de données → serveur d'accès (NREQ et NRES).
4. Serveur d'accès → client (RRES).



Ces préfixes sont suivis d'un type explicite de requête / réponse (AUTH, READ, WRITE, DELETE, MEET, GETDATA, SYNC, etc.).

**MEET** permet la connexion d'un serveur de données au serveur d'accès.

**GETDATA** permet de demander à un serveur de données qu'il envoie toutes ses données (pour futur SYNC).

**SYNC** demander à un serveur de données de remplacer toutes ses données par celles envoyées.

Le *status* d'un message peut être :

- SUCCESS → succès de la requête (écriture réussie, etc.).
- NORMAL → état normal (requête de lecture, etc.).
- ERR\_NOREPLY → erreur, pas de réponse (le relais ne répond pas, etc.).
- ERR\_AUTHFAILED → erreur, l'authentification a échoué.
- ERR\_NONODE → erreur, aucun serveur de données correspondant trouvé.
- ERR\_NOPERM → erreur, pas les permissions requises pour la lecture.
- ERR\_SYNTAX → erreur de syntaxe.
- ETC

## 2.2 Format des messages

Chaque message qui transite entre un client, le serveur d'accès et un serveur de données contient un **en-tête** et les **données** elles-mêmes.

ID	Requête	Status	Taille données	Somme de contrôle	Données
16 bits	8 bits	8 bits	16 bits	16 bits	de 0 à 65536 octets

## 2.3 Structure de message et associés

La structure qui représente un message (**dgram**) stocke, en plus des différents attributs d'en tête et des données vues ci-dessus, divers éléments importants :

- *IP* et port de provenance / de destination.
- La *date* de création / de réception (timestamp).
- Le nombre de renvoi (lorsque non réception de ACK de la part du récepteur du message).
- La *fonction de callback* appelée lorsque le nombre de renvoi a enclenché le timeout associé, ainsi qu'un *paramètre* pour ce callback.

Les *fonctions associées* aux **dgram** nécessitent souvent une ou deux listes, qui sont propres à chaque module, parmi :

- **dgsent** : la liste des *dgram envoyés*. Chaque message envoyé est **automatiquement supprimé** lors de la réception du ACK correspondant, ou après timeout. Les conserver permet de les *ré-envoyer* si le destinataire n'envoie pas de ACK de confirmation après un certain temps.

- **dgreceived** : la liste des *dgram reçus*. Chaque message reçu est automatiquement supprimé après envoi du ACK correspondant. Les conserver permet d'attendre que le message soit entier avant de l'utiliser (et timeout si non entier après un certain temps).

## 2.4 Format des données

Les messages ont un *format bien précis* (en-tête + données), mais les données ont aussi un format précis qui varie selon qui est l'émetteur et qui est le récepteur entre le client, le serveur d'accès et le serveur de données. Connaître les formats utilisés (ci-dessous) ainsi que le format des messages, leur types et status permet l'*interopérabilité* entre les modules développés ici et des modules externes dans le cas où tous utilisent les mêmes protocoles.

On suppose ici que les requêtes réussissent totess, et donc la section données n'est pas vide (sauf exceptions).

### 2.4.1 Client → Serveur d'accès (CREQ)

- *Authentication* : <login> :<mot\_de\_passe>
- *Lecture* : <champ>(<...>)
- *Écriture* : <champ1> :<valeur1>(<...>)
- *Suppression* : ∅
- *Bye* : ∅

### 2.4.2 Serveur d'accès → Serveur de données (RREQ, RNRES)

Dans le cas de RREQ (requête du serveur d'accès vers un serveur de données) :

- *Lecture* : <login\_utilisateur>
- *Écriture* : <login\_utilisateur> :<valeur>
- *Suppression* : <login\_utilisateur>
- *Synchronisation (SYNC)* : <login1> :<valeur1>(<...>)
- *Obtention données (GETDATA)* : <id\_serveur\_donnée\_pour\_sync>

Dans le cas de RNRES (réponse du serveur d'accès à un serveur de données) :

- *Rencontre (MEET)* : ∅

### 2.4.3 Serveur de données → Serveur d'accès (NREQ, NRES)

Dans le cas de NREQ (requête du serveur de données vers le serveur d'accès) :

- *Rencontre (MEET)* : <champ>

Dans le cas de NRES (réponse du serveur de données au serveur d'accès) :

- *Lecture* : <login1> :<valeur1>(...)
- *Écriture* : ∅
- *Suppression* : ∅
- *Obtention données (GETDATA)* : <id\_serveur\_données\_pour\_sync> :<login1> :<valeur1>(<login2> :<valeur2>(...))

### 2.4.4 Serveur d'accès → Client (RRES)

- *Authentification* : ∅
- *Lecture* : <champ> :<login1> :<valeur1>(<login2> :<valeur2>(...))
- *Écriture* : ∅
- *Suppression* : ∅
- *Bye* : ∅

## 3 Conclusion

Réaliser ce projet n'a pas été des plus faciles (même le plus difficile jusqu'à présent, pour nous) mais nous a permis d'approfondir nos connaissances de la *programmation sockets* et du *réseau* en général et nous en sommes très content. Le développement de ce projet nous a également aidé à nous organiser pour le *développement en collaboration* (Git).