

Déroulement du projet

Le projet se déroulera sur environ **trois semaines**, en **groupe de trois étudiants**. Il vous est communiqué **en amont** afin que vous puissiez commencer à y réfléchir dès maintenant, notamment pour anticiper certains choix techniques. Profitez de ce temps pour **travailler sur vos modélisations UML** et poser les premières bases de votre application.

👉 **Chaque fin de semaine**, vous devrez remettre un **livrable intermédiaire** comprenant un **rapport d'avancement** et le **code source** associé.

⚠️ L'objectif des deux premières semaines n'est **pas** d'avoir une version des livrables finalisée, mais de **montrer une évolution constante** de votre projet. L'évaluation portera donc essentiellement sur la **progression** de vos travaux.

À l'issue du projet, un **livrable final complet** sera attendu, composé des éléments suivants :

- Un **rapport final** comprenant :
 - une **introduction**,
 - les différentes **phases de développement** (problématiques, choix techniques, solutions mises en place),
 - une **partie sur l'organisation du travail**,
 - une **conclusion**.
- Un **support de présentation** qui devra :
 - être compréhensible par tous (même ceux extérieurs au projet),
 - mettre en lumière les **problématiques rencontrées** et les **choix techniques** effectués pour y répondre (ex. : choix du protocole réseau, threads vs processus, etc.).
- Une **démonstration fonctionnelle** de l'application.
- Le **code source**.

💡 *Le projet peut sembler conséquent lorsqu'on l'aborde seul, mais **le travail en équipe est votre atout principal** !*

Organisez-vous efficacement.

- La **partie 1** constitue le **socle commun** de l'application : elle doit être cohérente et solide.
- En revanche, les **parties 2, 3 et 4** sont relativement **indépendantes** : vous pouvez donc facilement vous répartir les tâches entre les membres du groupe

💡 *Avant de vous lancer à l'aveugle dans le projet, lisez attentivement chacune des parties détaillées afin d'avoir en tête l'application dans sa globalité.*

Objectif du projet

Ce projet a pour objectif de développer une application de messagerie textuelle distante (au minimum sur deux machines distinctes). Les utilisateurs pourront créer des salons de discussion, envoyer des messages privés, et exécuter des commandes sur le serveur. Deux types de profils seront pris en charge : les administrateurs et les utilisateurs standards.

Afin d'assurer la pérennité des données, des mécanismes de sauvegarde seront mis en place côté serveur, sous forme de fichiers.

Le développement du projet sera structuré en quatre grandes étapes :

1. Mise en place des fondations de l'application
2. Gestion d'un client unique
3. Implémentation de la sauvegarde des données dans des fichiers
4. Intégration des salons de communication

Partie 1 : Mise en place des communications réseau

Dans cette première phase, vous poserez les bases du système de communication réseau de l'application.

Vous devrez définir les exécutables côté client et côté serveur. L'application devra être capable de gérer **N utilisateurs**. Chaque client devra pouvoir **envoyer et recevoir simultanément des messages** via le serveur. Pour cela, chaque client devra utiliser **au moins deux threads ou processus** : un dédié à l'envoi et un autre à la réception.

Le choix entre l'utilisation de **threads ou de processus** vous revient. Pour vous aider à trancher, tenez compte des éléments suivants :

- **Les threads** partagent la même mémoire, ce qui permet une communication plus simple (pas besoin d'IPC).
- **Les processus**, en revanche, ont des espaces mémoire séparés, nécessitent l'usage d'IPC pour échanger des données, mais permettent l'utilisation de **signaux** (grâce aux PID) pour débloquer des processus en attente.

Vous devrez justifier votre choix, appuyez vous sur une modélisation pertinente (diagramme UML au choix) pour détailler le comportement du client en fonction de ces critères.

En ce qui concerne les **communications réseau**, vous devrez également choisir entre les protocoles **TCP** et **UDP**, étudiés en cours. Ce choix aura un impact direct sur votre implémentation.

Pour prendre votre décision, modélisez la partie serveur avec **N clients** dans les deux cas :

- En **TCP**, chaque client dispose d'une socket dédiée, active tant que la connexion reste ouverte.
- En **UDP**, une unique socket peut suffire pour gérer les échanges avec plusieurs clients.


En fonction du protocole choisi, vous devrez ensuite déterminer comment gérer plusieurs clients côté serveur : **multi-processus** ou **multi-thread** ?

Ce choix impactera aussi votre architecture serveur, en particulier la manière dont les données reçues sont traitées et acheminées vers le bon destinataire.

Encore une fois, gardez à l'esprit les implications de la mémoire partagée (threads) versus les IPC (processus) et les besoins potentiels en synchronisation entre les phases de réception, de traitement et d'envoi.

Pour normaliser les échanges et faciliter l'identification des commandes et de leurs arguments, vous devrez concevoir un **système de requêtes structuré**. Il s'agira de définir un ou plusieurs types de messages (hiérarchie de requête) permettant de distinguer clairement les différentes opérations. L'objectif est de garantir une **cohérence homogène de la communication** à travers toute l'application. Conseil : envisager d'y insérer la partie commande ébauché en partie 2. Votre structure doit vous permettre d'avoir les informations nécessaires (commande, données, etc) rapidement. Au besoin, ajouter des fonctions afin de faciliter sa manipulation.

Afin de permettre de fermer l'application client ou serveur proprement, notamment avec l'utilisation du CTRL+C de votre terminal, vous implémenterez une fermeture basée sur la gestion des signaux.

 *Si vous rencontrez des difficultés, n'hésitez pas à découper cette partie en plus petit morceau. Par exemple, limiter vous à simplement la connexion client/serveur avec uniquement 1 client. Puis, avec des tailles de messages de longueur dynamique. Ensuite, intégrez les structures de requêtes et finalement, intégrez la gestion des N clients.*

Partie 2 : Système de commandes et gestion des utilisateurs

En s'appuyant sur la structure de requêtes définie précédemment et les communications déjà établies, vous allez implémenter un **système de commandes**, à la manière de Discord.

Par exemple :

```
@help          → affiche la liste des commandes disponibles
@ping          → le serveur répond "pong"
@msg <user> <msg> → envoie un message privé à un utilisateur
@help -> affiche l'aide utilisateur (contenue dans un fichier à partir de la partie 2)
@credits -> affiche les credits de l'application (contenue dans un fichier à partir de la partie 2)
@shutdown -> permet d'éteindre correctement le serveur (basé sur la gestion des signaux)
```

Ce système est évolutif : l'objectif est de poser une base claire, facile à enrichir au fil du projet (notamment nécessaire pour la partie 4).

Dans cette partie, vous introduirez également un **système d'authentification simple** basé sur :

- un **pseudo unique**,
- un **mot de passe**.

Ces identifiants seront transmis **au moment de la connexion** entre un client et le serveur. Cette transmissions peut également se faire par l'appel d'une commande, @connect login mdp, appelé directement quand l'utilisateur à insérer son pseudo. Le serveur devra vérifier :

- l'unicité du pseudo parmi les clients connectés et enregistrés,
- puis, si le pseudo est valide, l'associer à une **adresse IP + port**.

Les identifiants (pseudos et mots de passe) seront stockés côté serveur.

(*Optionnel*) : Vous pouvez mettre en place un système de **rôles** pour différencier les privilèges (ex. : admin vs utilisateur standard), afin de restreindre certaines commandes à des utilisateurs spécifiques.

Pour clôturer cette deuxième partie, chaque client doit être capable :

- **d'envoyer un message au serveur**, qui l'affichera,
- **de recevoir un message du serveur**, qu'il devra afficher localement.

💡 *N'hésitez pas à définir une structure spécifique aux commandes afin de factoriser le code et faciliter l'ajout de commandes futures.*

Partie 3 : Gestion des fichiers et sauvegarde des données

Cette partie introduit deux fonctionnalités majeures :

1. La **sauvegarde des données utilisateurs**
2. Le **partage de fichiers** entre les clients et le serveur

Sauvegarde des utilisateurs

Pour assurer la persistance des données, vous devrez :

- Implémenter une **structure dédiée à la gestion de fichiers**, qui centralise les opérations courantes : ouverture, fermeture, lecture, écriture.
- Utiliser cette structure pour sauvegarder les informations des utilisateurs sur le disque (ex. fichier `save_users.txt` , en texte ou binaire).

À chaque connexion d'un client :

- Vérifiez s'il est déjà connecté ou enregistré.
- S'il est nouveau, enregistrez ses informations dans le fichier.

Création des fichiers README.txt et Credits.txt

- modification des commandes `@help` et `@credits` pour que le retour de ces deux commandes, soit l'affichage du contenu de leurs fichiers respectifs.


Transfert de fichiers : commandes `@upload` et `@download`

- `@upload <filename>` : le client envoie un fichier au serveur, qui le stocke.
- `@download <filename>` : le client demande à recevoir un fichier existant côté serveur.

Ces commandes permettent de gérer de manière simple des transferts de fichiers entre client et serveur.

(Optionnel) : Pour éviter les problèmes liés à l'envoi de **gros fichiers**, vous pouvez implémenter une **fragmentation des messages**.

Cela consiste à découper un fichier (ou message) en plusieurs morceaux, chacun transmis séparément. Ce principe fait écho à la **fragmentation IP** vue en cours.

 Pour le transfère de fichier, gardez en tête qu'un fichier n'est qu'un flux d'octets et qu'il peut-être stocké dans un `char*`.

Partie IV : Gestion des salons de discussion

Pour clôturer ce projet, vous allez implémenter un **système de salons** permettant de regrouper plusieurs clients côté serveur. Dans un premier temps, le nombre de clients maximum dans un channel peut être définis.

L'objectif est simple : lorsqu'un client envoie un message dans un salon, **tous les membres** de ce salon doivent le recevoir.

Cette fonctionnalité implique :

- l'ajout de **nouvelles commandes** dans le système existant (par exemple `@join` , `@leave` , `@create` , etc.),
- la gestion de **groupes de clients** côté serveur, en fonction du salon auquel ils sont rattachés.

Afin d'assurer la **persistance des salons**, vous devrez également :

- sauvegarder leur état (nom, membres, etc.) dans un **fichier**,
- charger ce fichier automatiquement au démarrage du serveur, pour restaurer les salons existants.

Ce mécanisme permettra aux utilisateurs de retrouver leurs salons même après un redémarrage du serveur.

(Optionnel) Proposer une solution cohérente permettant un nombre de client dynamique.

Bon courage !