



Institut Supérieur d'Electronique et Automatique ISEA  
Master 1 EEA IMEEN  
Mesures Énergétiques pour les Énergies Nouvelles

---

TP :

---

# **Développement d'un mini chat client-server en TCP**

---

Réalisé par : Rayen ALLAOUA

Sous la supervision de : Mr **MORERE Yann**

Année 2023/2024

## Table des matières

I. Recherche technique et bibliographique .....	4
1. Introduction .....	4
2. Sélection des Outils .....	4
2.1 Système d'Exploitation : Raisons de l'utilisation de Linux .....	4
2.2 IDE (Environnement de Développement Intégré) : Comparaison des IDE populaires ...	7
3. Les principaux langages de programmation utilisés sous Linux.....	9
• Langage C : .....	9
• Python : .....	9
4. Bibliothèques Graphiques .....	11
4.1 Définition des bibliothèques graphiques et leur rôle dans le développement d'interfaces utilisateur : .....	11
4.2 Les bibliothèques graphiques les plus populaires: .....	12
• Qt : .....	12
• Tkinter : .....	13
• wxWidgets: .....	14
4.3 Comparaison entre les bibliothèques graphiques les plus populaires : .....	14
5. Bibliothèques réseau et bibliothèques TCP : .....	16
5.1 Quelques points clés : .....	16
• Qu'est-ce qu'une bibliothèque réseau ? : .....	16
• Qu'est-ce qu'une bibliothèque TCP ? : .....	16
• Fonctionnalités clés des bibliothèques TCP : .....	16
• Avantages des bibliothèques TCP : .....	17
5.2 Lien avec l'application Chat TC : .....	17
II. Développement du mini chat client-serveur en TCP .....	19
1. Justification choix du langage python/ Tkinter pour la création du chat tcp : .....	19
1.1 python : .....	19
1.2 python : .....	19
2. L'architecture d'un chat TCP: .....	20

2.1 Code du Serveur TCP :	21
2.2 Code du Client TCP :	24
3. Défis et Solutions :	26
4. Compilation et test du code final	28
5. Conclusion	31

# I. Recherche technique et bibliographique

## 1. Introduction

Dans le cadre de ce projet, nous nous pencherons sur le développement d'un système de chat client-serveur.

Le principe du système de messagerie client-serveur repose sur un modèle où un serveur centralisé gère les connexions et la transmission des messages entre les clients connectés. Ces derniers se connectent au serveur pour échanger des messages, créant ainsi un environnement collaboratif de communication.

Les attentes pour ce projet sont diverses. En matière de fonctionnalités, notre objectif est de mettre en place un système fonctionnel, capable de gérer efficacement les connexions simultanées, la distribution des messages, tout en offrant une expérience utilisateur conviviale.

En résumé, notre projet vise à concevoir un système de messagerie client-serveur fonctionnel, fiable et extensible, fournissant une plateforme de communication interactive et contemporaine répondant aux besoins des utilisateurs.

## 2. Sélection des Outils

### 2.1 Système d'Exploitation : Raisons de l'utilisation de Linux

Linux est un système d'exploitation open source largement utilisé dans le monde de l'informatique pour plusieurs raisons clés :

1. **Open Source** : Linux est basé sur le modèle open source, ce qui signifie que son code source est disponible gratuitement et peut être modifié et distribué par quiconque. Cela favorise la transparence, l'innovation et la collaboration au sein de la communauté des développeurs.

2. **Stabilité et Fiabilité** : Linux est réputé pour sa stabilité et sa fiabilité. Il est conçu pour fonctionner sans plantages fréquents, ce qui en fait un choix idéal pour les systèmes critiques et les serveurs.

3. **Personnalisable** : Linux offre une grande flexibilité et personnalisation. Les utilisateurs peuvent choisir parmi de nombreuses distributions (comme Ubuntu, Fedora, CentOS, etc.) et personnaliser le système selon leurs besoins spécifiques.
4. **Sécurité** : Linux est réputé pour sa sécurité. Il offre un modèle de sécurité robuste avec des autorisations granulaires, des mises à jour régulières et un contrôle accru sur les paramètres de sécurité.
5. **Performance** : Linux est connu pour ses performances élevées, même sur du matériel moins puissant. Il est efficace dans la gestion des ressources système, ce qui le rend adapté à une variété d'applications, des systèmes embarqués aux serveurs de données.
6. **Communauté Active** : Linux bénéficie d'une communauté de développeurs et d'utilisateurs passionnée et active. Cela se traduit par un support technique robuste, des forums de discussion, des guides et des ressources en ligne abondantes.

### **Environnement de Développement et Pourquoi c'est le Choix des Informaticiens**

Un environnement de développement, ou IDE (Integrated Development Environment), est un ensemble d'outils logiciels qui facilitent le processus de développement d'applications. Voici pourquoi Linux est souvent le choix préféré des informaticiens pour leur environnement de développement :

1. **Compatibilité avec les Outils de Développement** : Linux est compatible avec une large gamme d'outils de développement populaires tels que GCC (GNU Compiler Collection), Python, Java Development Kit (JDK), Docker, Git, et bien d'autres. Cela permet aux développeurs d'utiliser leurs outils préférés sans problème.
2. **Terminal Puissant** : Linux offre un terminal puissant et personnalisable qui permet aux développeurs d'automatiser des tâches, d'exécuter des scripts, d'accéder rapidement aux commandes système et de gérer efficacement les environnements de développement.
3. **Environnement de Ligne de Commande** : Les développeurs apprécient souvent l'efficacité de la ligne de commande Linux pour exécuter des commandes complexes, naviguer dans le système de fichiers, manipuler des données et effectuer des opérations de développement.
4. **Liberté de Personnalisation** : Linux offre une liberté de personnalisation qui permet aux développeurs de configurer leur environnement de développement de manière optimale. Ils

peuvent choisir leur interface graphique, leurs outils de développement, leurs raccourcis clavier, etc.

**5. Support pour le Développement Open Source :** Étant un système d'exploitation open source lui-même, Linux favorise le développement d'applications open source. De nombreux développeurs préfèrent donc travailler sur Linux pour contribuer à des projets open source et collaboratifs.

Voici un tableau comparatif mettant en avant les avantages de Linux par rapport à Windows :

Caractéristique	Linux	Windows
<b>Coût</b>	Gratuit (open source)	Coût de licence pour les versions complètes
<b>Personnalisation</b>	Grande flexibilité et personnalisation	Personnalisation limitée, surtout dans les versions grand public
<b>Stabilité et Fiabilité</b>	Très stable et fiable, peu de plantages	Parfois sujet à des problèmes de stabilité
<b>Sécurité</b>	Architecture sécurisée, moins de vulnérabilités	Sujets à un plus grand nombre de menaces de sécurité
<b>Performances</b>	Performances élevées même sur des matériels modestes	Parfois plus exigeant en termes de ressources système
<b>Environnement de développement</b>	Vaste gamme d'outils de développement disponibles	Outils de développement spécifiques (Visual Studio)
<b>Support matériel</b>	Supporte une large gamme de matériels	Support pour un nombre limité de matériels
<b>Mises à jour</b>	Mises à jour régulières et flexibles	Mises à jour parfois contraignantes et non personnalisables
<b>Communauté et support</b>	Communauté active, support technique robuste	Support technique mais moins d'options communautaires
<b>Virtualisation</b>	Excellente prise en charge pour la	Prise en charge de la virtualisation, mais moins

	virtualisation	intégrée
<b>Évolutivité</b>	Adapté aux systèmes embarqués et aux serveurs	Principalement utilisé sur les postes de travail

## 2.2 IDE (Environnement de Développement Intégré) : Comparaison des IDE populaires

Un IDE est un outil essentiel pour les développeurs car il offre un ensemble complet de fonctionnalités pour faciliter le développement de logiciels. Voici une comparaison des IDE populaires en mettant en avant leurs caractéristiques et leurs points forts :

IDE	Caractéristiques Clés	Langages de Programmation Supportés	Points Forts
<b>Visual Studio Code (VS Code)</b>	<ul style="list-style-type: none"> <li>- Éditeur de code polyvalent et léger</li> <li>- Intégration avec Git et autres outils de développement</li> <li>- Extensions riches pour améliorer la productivité</li> <li>- Prise en charge des systèmes d'exploitation multiples</li> </ul>	<ul style="list-style-type: none"> <li>- Prise en charge de nombreux langages</li> <li>- JavaScript, TypeScript, Python, Java, C#, etc.</li> <li>- HTML, CSS, PHP, Ruby, etc.</li> </ul>	<ul style="list-style-type: none"> <li>- Extensions riches et personnalisables</li> <li>- Intégration avec des outils de développement populaires comme Docker, Kubernetes, etc.</li> <li>- Communauté active avec de nombreuses extensions disponibles pour tous les besoins de développement</li> </ul>
<b>JetBrains IntelliJ IDEA</b>	<ul style="list-style-type: none"> <li>- Éditeur de code intelligent avec analyse avancée</li> <li>- Outils de refactoring puissants</li> <li>- Support pour le développement Web et mobile</li> <li>- Version payante avec plus de fonctionnalités</li> </ul>	<ul style="list-style-type: none"> <li>- Java, Kotlin, Scala, Groovy, etc.</li> <li>- HTML, CSS, JavaScript, TypeScript, etc.</li> </ul>	<ul style="list-style-type: none"> <li>- Fonctionnalités avancées pour la programmation Java et Android, y compris la détection d'erreurs en temps réel</li> <li>- Intégration avec des frameworks populaires comme Spring, Hibernate, etc.</li> <li>- Prise en charge de projets multi-modules avec une gestion avancée des dépendances</li> </ul>

<b>Eclipse</b>	<ul style="list-style-type: none"> <li>- Plateforme extensible avec une grande communauté</li> <li>- Intégration avec des frameworks et outils de développement</li> <li>- Prise en charge de l'outillage de développement collaboratif</li> <li>- Version Eclipse IDE for Enterprise Java and Web Developers</li> </ul>	<ul style="list-style-type: none"> <li>- Java, C/C++, PHP, JavaScript, etc.</li> <li>- Python, Ruby, etc.</li> </ul>	<ul style="list-style-type: none"> <li>- Personnalisable avec une large gamme de plugins et d'extensions pour divers besoins de développement</li> <li>- Large écosystème de développement avec des outils pour le développement Web, mobile, et bien d'autres domaines</li> <li>- Utilisé dans de nombreux projets open source et commerciaux, offrant une grande adaptabilité et flexibilité aux développeurs</li> </ul>
<b>NetBeans</b>	<ul style="list-style-type: none"> <li>- Interface utilisateur conviviale et personnalisable</li> <li>- Prise en charge des environnements de développement Java EE</li> <li>- Intégration avec Apache Maven et autres outils de build</li> </ul>	<ul style="list-style-type: none"> <li>- Java, PHP, HTML5, JavaScript, etc.</li> <li>- C/C++, Python, Ruby, etc.</li> </ul>	<ul style="list-style-type: none"> <li>- Support complet pour le développement Java EE avec des outils pour la création d'applications Web, mobiles et de bureau</li> <li>- Outils de développement Web avec prise en charge des frameworks populaires comme AngularJS, Vue.js, etc.</li> <li>- Facilité d'utilisation pour les débutants avec une courbe d'apprentissage douce et des fonctionnalités avancées pour les experts</li> </ul>

Ces IDE sont largement utilisés dans le développement logiciel et offrent une gamme variée de fonctionnalités pour répondre aux besoins des développeurs, que ce soit pour le développement Web, mobile, ou de logiciels d'entreprise. Chaque IDE a ses forces et ses domaines d'expertise, donc le choix dépend souvent des langages de programmation utilisés et des préférences personnelles des développeurs.



### 3. Les principaux langages de programmation utilisés sous Linux

Les principaux langages de programmation utilisés sous Linux sont :

- **Langage C :**

Le langage C est le plus utilisé pour le développement du noyau Linux. C'est un langage de programmation système très puissant et portable. Il permet d'écrire du code à bas niveau, proche du matériel. Le C est idéal pour développer des applications système, des pilotes de périphériques, des interpréteurs, etc.

- **Python :**

Python est un langage de programmation interprété, orienté objet et multi-paradigme. Il est très populaire pour le scripting système, le développement web, l'analyse de données et l'intelligence artificielle. Python est facile à apprendre et dispose d'une vaste bibliothèque standard. Il est très utilisé pour automatiser des tâches sous Linux.

Critères	Python	Java	C++
<b>Syntaxe</b>	Claire et concise	Verbeuse par rapport à Python, syntaxe plus rigide	Complexité syntaxique, plus proche du langage machine
<b>Facilité d'apprentissage</b>	Facile à apprendre	Accessible mais plus de concepts à assimiler	Plus difficile, nécessite une bonne maîtrise
<b>Performance</b>	Performances généralement satisfaisantes	Bonnes performances, mais dépend de la JVM	Performances élevées grâce à la compilation
<b>Gestion de la mémoire</b>	Gestion automatique de la mémoire (garbage collector)	Gestion automatique de la mémoire via la JVM	Gestion manuelle de la mémoire

<b>Bibliothèques</b>	Large gamme de bibliothèques pour les sockets et le réseau	API robuste pour les communications réseau	Bibliothèques étendues pour les opérations réseau
<b>Portabilité</b>	Bonne portabilité sur Linux et autres plateformes	Très portable grâce à la JVM	Dépendant du compilateur et de l'environnement
<b>Performance réseau</b>	Bon support pour les opérations réseau	API réseau robuste	Frameworks comme Boost offrant des performances élevées
<b>Adaptabilité</b>	Convient aux applications nécessitant un développement rapide	Adapté aux applications nécessitant une portabilité élevée	Adapté aux applications nécessitant des performances élevées
<b>Complexité</b>	Moins complexe, adapté aux débutants	Niveau intermédiaire en termes de complexité	Plus complexe, nécessite une expertise plus avancée

### Bash :

Bash (Bourne-Again SHell) est l'interpréteur de commandes par défaut sur la plupart des distributions Linux. Il permet d'écrire des scripts pour automatiser des tâches système, combiner des commandes et développer des interfaces en ligne de commande. Bash est un outil indispensable pour les administrateurs Linux.

### Autres langages

D'autres langages comme C++, Ruby, Perl, Rust, Go sont aussi utilisés sous Linux pour des usages spécifiques<sup>1</sup>. Le choix du langage dépend des besoins du projet, des performances requises, de la portabilité et des bibliothèques disponibles. En résumé, C, Python et Bash sont les langages les plus répandus pour le développement système et applicatif sous Linux. Chacun a ses forces et convient à des usages différents. Un développeur Linux doit maîtriser au moins l'un de ces langages pour pouvoir créer des programmes efficaces et interagir avec le système d'exploitation.

## 4. Bibliothèques Graphiques

### 4.1 Définition des bibliothèques graphiques et leur rôle dans le développement d'interfaces utilisateur :

Les bibliothèques graphiques, aussi appelées boîtes à outils graphiques ou GUI toolkits, sont des ensembles de composants logiciels qui permettent de créer des interfaces utilisateur graphiques (GUI) pour les applications. Leur rôle principal est de fournir une abstraction au-dessus des fonctionnalités graphiques de bas niveau du système d'exploitation, afin de faciliter le développement d'interfaces utilisateur portables et cohérentes.

Voici quelques points clés concernant les bibliothèques graphiques et leur rôle dans le développement d'interfaces utilisateur :

1. **Création d'éléments graphiques** : Les bibliothèques graphiques permettent de créer des éléments visuels tels que fenêtres, boutons, menus, barres de progression, zones de texte, graphiques, etc. Elles offrent des méthodes pour dessiner et manipuler ces éléments de manière programmable.
2. **Interactivité utilisateur** : Elles facilitent la gestion des événements utilisateur tels que clics de souris, touches du clavier, défilement, etc. Les développeurs peuvent définir des actions à effectuer en réponse à ces événements, ce qui rend l'interface utilisateur interactive et réactive.
3. **Gestion des mises en page** : Les bibliothèques graphiques proposent souvent des systèmes de gestion de mise en page pour organiser les éléments graphiques à l'écran. Cela inclut des fonctionnalités telles que les grilles, les boîtes, les onglets, les couches, etc., facilitant la création d'interfaces bien structurées.
4. **Personnalisation et style** : Elles permettent de personnaliser l'apparence des éléments graphiques en modifiant les couleurs, les polices, les tailles, les styles de bordure, les icônes, etc. Cela aide à créer des interfaces esthétiquement attrayantes et cohérentes avec l'image de marque de l'application.
5. **Support multimédia** : Certaines bibliothèques graphiques offrent des fonctionnalités avancées pour intégrer des éléments multimédias tels que des images, vidéos, animations, sons, ce qui enrichit l'expérience utilisateur.
6. **Compatibilité multiplateforme** : De nombreuses bibliothèques graphiques sont conçues pour être compatibles avec plusieurs systèmes d'exploitation, ce qui permet aux développeurs de

créer des applications qui fonctionnent de manière transparente sur différentes plateformes.

En résumé, les bibliothèques graphiques jouent un rôle crucial dans le développement d'interfaces utilisateur modernes et interactives en offrant des outils et des fonctionnalités qui simplifient la conception, la personnalisation et la gestion des éléments visuels dans les applications logicielles. Elles contribuent ainsi à améliorer l'expérience utilisateur et à rendre les applications plus attrayantes et conviviales.

## 4.2 Les bibliothèques graphiques les plus populaires:

Les bibliothèques graphiques les plus populaires sont :

- **Qt :**



- Langage de programmation : Principalement utilisé avec C++ mais prend également en charge d'autres langages comme Python grâce à PyQt.
- Caractéristiques :
  - Framework multi-plateforme qui fonctionne sur des systèmes d'exploitation tels que Windows, Linux, macOS, Android, iOS, etc.
  - Offre une grande variété de widgets et d'outils pour la création d'interfaces graphiques avancées.
  - Intégration étroite avec l'environnement de bureau KDE sous Linux.
- Utilisation : Très populaire pour le développement d'applications graphiques riches et complexes.

- **GTK (GIMP Toolkit) :**

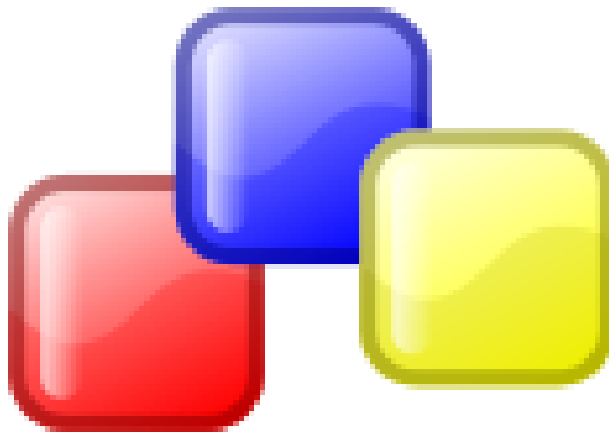


- Langage de programmation : Écrit en C avec des bindings disponibles pour de nombreux langages comme Python (PyGTK), C++, etc.
- Caractéristiques :
  - Boîte à outils graphique libre, utilisée principalement dans l'environnement GNOME sous Linux.
  - Possède une large gamme de widgets pour la création d'interfaces utilisateur modernes et esthétiques.
  - Fonctionne sur plusieurs plateformes, y compris Linux, Windows et macOS.
  - Utilisation : Adapté pour le développement d'applications GNOME et d'autres applications graphiques sous Linux.
- **Tkinter :**



- Langage de programmation : Interface Python pour la bibliothèque Tk.
- Caractéristiques :
  - Incluse par défaut dans Python, ce qui en fait un choix pratique pour les développeurs Python.
  - Idéale pour la création d'interfaces utilisateur simples et rapides à mettre en œuvre.
  - Offre des widgets de base pour la création de fenêtres, de boutons, de menus, etc.

- Utilisation : Convient aux applications nécessitant des interfaces utilisateur simples et fonctionnelles en Python.
- **wxWidgets:**



- Langage de programmation : Principalement utilisé avec C++ mais dispose de bindings pour d'autres langages comme Python (wxPython).
- Caractéristiques :
  - Bibliothèque multi-plateforme prenant en charge Windows, Linux, macOS, etc.
  - Offre une grande variété de widgets et de fonctionnalités pour la création d'applications graphiques.
  - Utilisée par des applications populaires telles que FileZilla.
- Utilisation : Appropriée pour le développement d'applications avec des interfaces riches et personnalisables.

En résumé, chacune de ces bibliothèques graphiques a ses propres avantages et convient à différents types de projets en fonction des besoins spécifiques en matière de plateforme, de langage de programmation et de complexité de l'interface utilisateur. Le choix de la bibliothèque dépendra donc des exigences et des objectifs du projet de développement.

#### **4.3 Comparaison entre les bibliothèques graphiques les plus populaires :**

Le tableau suivant compare les bibliothèques graphiques en fonction de différents critères tels que le langage de programmation, les plateformes prises en charge, les fonctionnalités et widgets disponibles, la popularité et l'usage, l'intégration avec le langage, les capacités graphiques et la performance. Chaque bibliothèque a ses propres avantages et convient à différents types de projets en fonction des besoins spécifiques en matière d'interface utilisateur.

Critères	Qt	GTK	Tkinter	wxWidgets
<b>Langage de programmation</b>	C++ (Python avec PyQt)	C (bindings pour plusieurs langages)	Python (interface pour Tk)	C++ (bindings pour plusieurs langages)
<b>Plateformes prises en charge</b>	Multi-plateforme (Windows, Linux, macOS, Android, iOS, etc.)	Multi-plateforme (Linux, Windows, macOS)	Multi-plateforme (Incluse par défaut dans Python)	Multi-plateforme (Windows, Linux, macOS)
<b>Environnements spécifiques</b>	KDE sous Linux	GNOME sous Linux	N/A	N/A
<b>Widgets et fonctionnalités</b>	Large variété de widgets avancés	Large variété de widgets modernes	Widgets de base pour des interfaces simples	Grande variété de widgets et fonctionnalités
<b>Popularité et usage</b>	Très populaire pour des applications riches et complexes	Utilisé principalement dans l'environnement GNOME	Idéal pour des interfaces simples en Python	Utilisé par des applications comme FileZilla
<b>Intégration avec le langage</b>	Bonne intégration avec C++ et Python	Bonne intégration avec le langage C	Incluse par défaut dans Python	Bonne intégration avec C++ et d'autres langages
<b>Capacités graphiques</b>	Très bonnes capacités	Bonnes capacités graphiques	Fonctionnalités de base pour des interfaces simples	Très bonnes capacités graphiques et

	graphiques			fonctionnalités
<b>Performance</b>	Performant et efficace	Performant	Efficace pour des interfaces simples	Performant et efficace pour des interfaces riches

## 5. Bibliothèques réseau et bibliothèques TCP :

### 5.1 Quelques points clés :

Les bibliothèques réseau, et plus particulièrement les bibliothèques TCP, jouent un rôle essentiel dans la communication et l'échange de données sur les réseaux informatiques. Voici quelques points clés :

- Qu'est-ce qu'une bibliothèque réseau ? :

Une bibliothèque réseau est un ensemble de fonctions et de routines logicielles permettant à une application de communiquer sur un réseau. Elles fournissent une interface de programmation (API) standardisée pour simplifier l'accès aux services réseau.

- Qu'est-ce qu'une bibliothèque TCP ? :

Les bibliothèques TCP (Transmission Control Protocol) sont un type spécifique de bibliothèque réseau qui implémentent le protocole TCP, l'un des principaux protocoles de communication sur Internet. Elles permettent d'établir des connexions fiables et ordonnées entre applications sur un réseau.

- Fonctionnalités clés des bibliothèques TCP :

- Établissement et gestion de connexions TCP
- Envoi et réception de données de manière fiable
- Gestion des acquittements, retransmissions et contrôle de flux
- Prise en charge de la fragmentation et du réassemblage des paquets



- Fonctions de haut niveau comme la résolution de noms de domaine
- Avantages des bibliothèques TCP :

- Simplification du développement d'applications réseau
- Gestion transparente des aspects techniques du protocole TCP
- Portabilité des applications sur différentes plateformes
- Fiabilité et robustesse des communications

En résumé, les bibliothèques réseau, et en particulier les bibliothèques TCP, sont des composants logiciels essentiels pour permettre aux applications de communiquer efficacement sur un réseau informatique.

## 5.2 Lien avec l'application Chat TC :

Les bibliothèques réseau, en particulier les bibliothèques TCP, sont essentielles pour le développement d'une application de chat comme celle qu' on a créée. Voici comment elles sont liées à notre application de chat TCP :

1. **Communication réseau** : Les bibliothèques réseau, telles que la bibliothèque TCP qu'on a utilisé, permettent à notre application de chat d'établir des connexions réseau entre les clients et le serveur. Dans le contexte d'une application de chat, TCP est souvent préféré en raison de sa fiabilité pour la transmission des messages.
2. **Transmission fiable des messages** : TCP garantit une transmission fiable des messages entre les clients et le serveur. Cela signifie que les messages envoyés par un client seront reçus dans l'ordre correct et sans perte par le serveur et vice versa.
3. **Gestion des sockets** : Les bibliothèques TCP fournissent des fonctionnalités pour la création, la liaison et l'utilisation de sockets TCP. Dans notre application de chat, chaque client et le serveur utilisent des sockets pour établir des connexions et échanger des messages.
4. **Gestion des connexions** : TCP gère la logique de connexion entre les clients et le serveur. Il permet d'établir des connexions persistantes (connexion orientée session) pour que les clients puissent envoyer et recevoir des messages continuellement pendant la session de chat.
5. **Traitement des erreurs et des exceptions** : Les bibliothèques TCP incluent généralement des mécanismes de gestion des erreurs et des exceptions liées à la communication réseau. Cela garantit une meilleure robustesse et une meilleure fiabilité de notre application de chat en cas de problèmes de réseau ou de communication.
6. **Intégration avec l'interface utilisateur** : Bien que les bibliothèques réseau se concentrent

principalement sur la communication réseau, elles peuvent être intégrées avec des interfaces utilisateur comme Tkinter (dans le cas de Python) pour créer une application de chat avec une interface graphique conviviale.

## II. Développement du mini chat client-serveur en TCP

### 1. Justification choix du langage python/Tkinter pour la création du chat tcp :

#### 1.1 python :

Tout d'abord, la simplicité et la lisibilité de Python sont des atouts majeurs. Cela facilite la compréhension du code et accélère le processus de développement, tout en réduisant les risques d'erreurs.

Ensuite, Python offre une large gamme de bibliothèques, notamment pour la programmation réseau. On peut citer le module **socket** intégré qui simplifie la mise en place des communications TCP/IP.

Python est également compatible avec plusieurs systèmes d'exploitation, ce qui assure que notre application de chat peut fonctionner sur différents environnements, qu'il s'agisse de Windows, macOS ou Linux malgré que pour ce mini projet on se focalise que sur Linux.

La communauté Python est très active, offrant de nombreuses ressources, tutoriels, forums et bibliothèques tierces. Cela nous permet de résoudre les problèmes et d'étendre les fonctionnalités de notre application.

Enfin, Python est flexible et extensible. On peut développer des applications complexes tout en les rendant facilement extensibles. Cela nous donne la possibilité d'ajouter des fonctionnalités supplémentaires au chat TCP, comme des interfaces graphiques avec Tkinter, des protocoles de sécurité, etc.

En résumé, le choix du langage Python pour notre chat TCP est motivé par sa simplicité, sa puissance, sa compatibilité multiplateforme, sa vaste communauté et ses ressources abondantes, ainsi que sa flexibilité pour répondre à nos besoins évolutifs.

#### 1.2 Tkinter :

Pour l'interface graphique, on a choisi Tkinter pour la création de ce chat TCP, cela peut être justifié par plusieurs raisons :

1. **Intégration avec Python** : Tkinter est la bibliothèque d'interface graphique standard de Python, ce qui signifie qu'elle est directement intégrée à l'écosystème Python sans nécessiter d'installations supplémentaires. Cela facilite le développement et la distribution de l'application.
2. **Facilité d'apprentissage** : Tkinter est réputé pour être convivial et relativement simple à

apprendre pour les débutants. Sa syntaxe intuitive et sa documentation complète en font un bon choix pour les développeurs qui souhaitent créer rapidement des interfaces graphiques.

3. **Large gamme de widgets** : Tkinter offre une variété de widgets prêts à l'emploi tels que des boutons, des étiquettes, des champs de texte, etc. Cela permet de concevoir une interface utilisateur interactive et fonctionnelle pour un chat TCP.

4. **Compatibilité multiplateforme** : Les applications Tkinter fonctionnent de manière transparente sur plusieurs plateformes telles que Windows, macOS et Linux. Cela garantit une portabilité et une accessibilité maximales pour les utilisateurs finaux.

5. **Support communautaire** : Étant une bibliothèque populaire et largement utilisée, Tkinter bénéficie d'un fort soutien de la communauté Python. Il existe de nombreux tutoriels, exemples de code et forums de discussion qui peuvent aider les développeurs à résoudre des problèmes et à améliorer leurs compétences.

En résumé, le choix de Tkinter pour ce projet de chat TCP repose sur sa simplicité, sa compatibilité multiplateforme, sa documentation exhaustive et son support communautaire, ce qui en fait un outil efficace pour le développement d'interfaces graphiques en Python.

## 2. L'architecture d'un chat TCP:

L'architecture d'un chat TCP repose sur plusieurs composants et interactions qui assurent le bon fonctionnement de la communication entre les utilisateurs. Voici une description générale de l'architecture d'un tel système :

**Serveur TCP** : Le serveur TCP est le point central du chat. Il est responsable de la gestion des connexions entrantes des clients, de la réception et de la diffusion des messages entre les clients connectés.

**Clients TCP** : Les clients TCP sont les applications installées sur les appareils des utilisateurs. Chaque client se connecte au serveur TCP pour pouvoir envoyer et recevoir des messages.

**Communication TCP** : La communication entre le serveur et les clients se fait via le protocole TCP (Transmission Control Protocol). TCP assure une connexion fiable et orientée flux entre les machines, garantissant que les données sont correctement reçues et dans l'ordre.

**Sockets** : Les sockets sont des interfaces de programmation réseau utilisées pour établir des connexions TCP entre les clients et le serveur. Chaque socket est associé à une adresse IP et un

numéro de port.

**Protocole de communication :** Un protocole de communication définit les règles et le format des messages échangés entre le serveur et les clients. Il peut inclure des commandes spéciales pour la gestion des utilisateurs, la diffusion de messages, etc.

**Gestion des connexions :** Le serveur doit gérer les connexions entrantes des clients, attribuer des identifiants uniques à chaque client (nicknames), maintenir une liste des clients connectés, gérer les déconnexions et les reconnexions, etc.

**Diffusion des messages :** Lorsqu'un client envoie un message au serveur, celui-ci doit diffuser ce message à tous les autres clients connectés. La diffusion peut être réalisée de manière synchrone ou asynchrone, en fonction des besoins de l'application.

**Interface utilisateur :** Chaque client dispose d'une interface utilisateur qui permet à l'utilisateur d'entrer des messages, de visualiser les messages reçus et d'interagir avec d'autres fonctionnalités du chat.

**Gestion des erreurs :** L'architecture doit également prendre en compte la gestion des erreurs et des exceptions, telles que les déconnexions inattendues, les problèmes de réseau, etc., pour assurer la robustesse et la fiabilité du système.

En résumé, l'architecture d'un chat TCP comprend un serveur central, des clients connectés via des sockets TCP, un protocole de communication, des mécanismes de gestion des connexions et des messages, ainsi qu'une interface utilisateur pour chaque client. Cela forme un système cohérent qui permet aux utilisateurs de communiquer en temps réel via un réseau TCP.

## 2.1 Code du Serveur TCP :

Regardons de plus près comment chaque partie du code contribue à l'architecture d'un chat TCP (le code en entier est dans l'annexe) :

### 1. Initialisation du serveur et des variables

- Au début du code, nous définissons l'hôte (HOST) et le port (PORT) sur lesquels le serveur écoutera les connexions des clients. Cela est crucial car cela détermine comment les clients peuvent se connecter au serveur.

```
# Définition de l'adresse IP et du port du serveur
HOST = "127.0.0.1"
PORT = 8080

# Initialisation des variables pour le nom du client, la liste des clients et la liste des noms des clients
client_name = ""
clients = []
client_names = []
```

## 2. Fonctions pour gérer les clients

- La fonction **send\_recv\_client\_msg** est essentielle car elle gère la communication bidirectionnelle avec chaque client. Elle reçoit les messages des clients et les redistribue aux autres clients connectés. Cela reflète le flux de messages dans un chat, où chaque participant peut envoyer et recevoir des messages.

```
def send_recv_client_msg(client_connection, client_ip_addr):
    ...

def get_client_index(client_list, curr_client):
    ...
```

```
# Fonction pour envoyer et recevoir des messages des clients
def send_recv_client_msg(client_connection, client_ip_addr):
    global server, client_name, clients, clients_addr # Déclaration des variables globales utilisées dans la fonction
    client_msg = "" # Initialisation du message du client
    client_name = client_connection.recv(4096) # Réception du nom du client
    client_name = client_name.decode("utf-8") # Décodage du nom du client
    print(type(client_name))
    welcome_message = "Welcome " + client_name + ". Use 'exit' to quit" # Message de bienvenue
    client_connection.send(welcome_message.encode("utf-8")) # Envoi du message de bienvenue au client
    client_names.append(client_name) # Ajout du nom du client à la liste des noms de clients

    # Mise à jour de l'affichage des noms de clients sur l'interface graphique
    update_client_names_display(client_names)

    while True:
        print("Waiting to receive message from a client")
        data = client_connection.recv(4096) # Réception des données du client
        if not data: break # Sortir de la boucle si aucune donnée n'est reçue
        if data == "exit": break # Sortir de la boucle si le client envoie "exit"

        client_msg = data.decode("utf-8") # Décodage des données du client
        idx = get_client_index(clients, client_connection) # Obtention de l'indice du client dans la liste des clients
        sending_client_name = client_names[idx] # Obtention du nom du client envoyant le message

        for client in clients:
            if client != client_connection:
                message_to_other_clients = sending_client_name + "->" + client_msg # Création du message à envoyer aux autres clients
                client.send(message_to_other_clients.encode("utf-8")) # Envoi du message aux autres clients
```

```
# Fonction pour obtenir l'indice d'un client dans la liste des clients
def get_client_index(client_list, curr_client):
    for i in range(0, len(client_list)):
        if client_list[i] == curr_client:
            return i
    return -1
```

## 3. Gestion des connexions clients

- La fonction **accept\_clients** est responsable de la gestion des nouvelles connexions entrantes des clients. En utilisant des threads, elle permet au serveur de gérer plusieurs clients simultanément sans bloquer le flux principal du programme.

Cela est crucial pour assurer une expérience fluide aux utilisateurs du chat.

```
# Fonction pour accepter les connexions des clients
def accept_clients(some_server):
    print("got to accept clients function")
    while True:
        client, addr = some_server.accept() # Acceptation de la connexion d'un client
        print("Server accepted a client")
        clients.append(client) # Ajout du client à la liste des clients
        threading._start_new_thread(send_rcv_client_msg, (client,addr,)) # Démarrage d'un thread pour gérer la communication avec le client
```

#### 4. Démarrage et arrêt du serveur

- Les boutons "Start" et "Stop" dans l'interface graphique fournissent un moyen convivial de démarrer et d'arrêter le serveur. Cela donne au serveur une interface de contrôle utilisateur, permettant à l'administrateur du chat de gérer facilement l'état du serveur.

```
# Fonction pour démarrer le serveur
def start_server():
    startButton.config(state=tk.DISABLED) # Désactivation du bouton de démarrage
    stopButton.config(state=tk.NORMAL) # Activation du bouton d'arrêt

    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Création du socket serveur
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # Configuration du socket
    print("Created server socket")
    server.bind((HOST, PORT)) # Liaison du socket à l'adresse et au port spécifiés
    server.listen(5) # Attente de connexions entrantes
    print("Server is listening")
    threading._start_new_thread(accept_clients, (server,)) # Démarrage d'un thread pour accepter les connexions des clients

    hostLabel["text"] = "Host: " + HOST # Mise à jour du label d'adresse IP
    portLabel["text"] = "Port: " + str(PORT) # Mise à jour du label de port

# Fonction pour arrêter le serveur
def stop_server():
    stopButton.config(state=tk.DISABLED) # Désactivation du bouton d'arrêt
    startButton.config(state=tk.NORMAL) # Activation du bouton de démarrage
```

#### 5. Interface graphique

- L'interface graphique fournit une vue conviviale de l'état du serveur et des clients connectés. Elle affiche l'hôte et le port du serveur, ainsi qu'une liste des clients actuellement connectés. Cela améliore l'expérience utilisateur en offrant une visualisation claire de ce qui se passe dans le chat.

```
# Création de la partie supérieure de l'interface graphique avec les boutons de démarrage et d'arrêt du serveur
topFrame = tk.Frame(window)

startButton = tk.Button(topFrame, text="Start", command=start_server)
startButton.pack(side=tk.LEFT)

stopButton = tk.Button(topFrame, text="Stop", command=stop_server, state=tk.DISABLED)
stopButton.pack()
topFrame.pack(side=tk.TOP, pady=(5,0))

# Création de la partie centrale de l'interface graphique avec les labels d'adresse IP et de port
centerFrame = tk.Frame(window)
hostLabel = tk.Label(centerFrame, text="Host: x.x.x.x")
hostLabel.pack(side=tk.LEFT)
portLabel = tk.Label(centerFrame, text="Port: ###")
portLabel.pack()
centerFrame.pack(side=tk.TOP, pady=(5,0))

# Création de la partie inférieure de l'interface graphique pour afficher les noms des clients connectés
clientsFrame = tk.Frame(window)
clientsLabel = tk.Label(clientsFrame, text="*****CONNECTED CLIENTS*****")
clientsLabel.pack() # Ajout du label pour indiquer les clients connectés
scrollBar = tk.Scrollbar(clientsFrame) # Création d'une barre de défilement
scrollBar.pack(side=tk.RIGHT, fill=tk.Y) # Placement de la barre de défilement à droite
tkDisplay = tk.Text(clientsFrame, height=15, width=40) # Création d'une zone de texte pour afficher les noms des clients
tkDisplay.pack(side=tk.LEFT, fill=tk.Y, padx=(5,0)) # Placement de la zone de texte à gauche
scrollBar.config(command=tkDisplay.yview) # Configuration de la barre de défilement pour fonctionner avec la zone de texte
tkDisplay.config(yscrollcommand=scrollBar.set, background="#F4F6F7", highlightbackground="grey", state="disabled") # Configuration de la zone de texte
clientsFrame.pack(side=tk.BOTTOM, pady=(5, 10)) # Placement du cadre des clients en bas de la fenêtre

window.mainloop() # Lancement de la boucle principale pour exécuter l'interface graphique
```

## 6. Création du socket serveur :

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Création du socket serveur
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # Configuration du socket
print("Created server socket")
server.bind((HOST, PORT)) # Liaison du socket à l'adresse et au port spécifiés
server.listen(5) # Attente de connexions entrantes
print("Server is listening")
threading._start_new_thread(accept_clients, (server,)) # Démarrage d'un thread pour accepter les connexions des clients
```

Explications :

1. **socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)**: Cette ligne crée un objet socket de type TCP/IP (**SOCK\_STREAM**) pour le serveur.
2. **server.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)**: Cette ligne configure le socket pour réutiliser l'adresse et le port rapidement après la fermeture du serveur, ce qui est utile lors du redémarrage rapide du serveur.
3. **server.bind((HOST, PORT))**: Cette ligne lie le socket à l'adresse IP (**HOST**) et au port (**PORT**) spécifiés. C'est ici que l'adresse IP et le port sont définis pour que le serveur puisse écouter les connexions entrantes.
4. **server.listen(5)**: Cette ligne indique au serveur d'attendre jusqu'à 5 connexions entrantes en attente. Cela signifie que le serveur peut traiter jusqu'à 5 clients simultanément.
5. **threading.\_start\_new\_thread(accept\_clients, (server,))**: Cette ligne démarre un nouveau thread pour gérer l'acceptation des connexions des clients, permettant ainsi au serveur de gérer plusieurs clients en parallèle sans bloquer le thread principal du serveur.

Dans l'ensemble, ce code combine efficacement les éléments clés de l'architecture d'un chat TCP : la gestion des connexions, la communication entre les clients et le serveur, et une interface utilisateur intuitive. Cela crée une base solide pour un chat TCP fonctionnel et convivial.

## 2.2 Code du Client TCP :

La correspondance entre chaque partie du code et son rôle dans l'architecture d'un chat TCP (le code en entier est dans l'annexe) :

### 1. Interface graphique utilisateur (GUI) :

- Les lignes de code qui créent les éléments de l'interface graphique, tels que **tkinter**, les étiquettes (**Label**), les champs de texte (**Entry**, **Text**), les boutons (**Button**), et la gestion des événements comme l'appui sur la touche "Entrée", sont les parties du code qui correspondent à cette fonctionnalité.



```
import tkinter as tk # Importation de la bibliothèque tkinter pour créer une interface graphique
from tkinter import messagebox # Importation de la fonction messagebox de tkinter pour afficher des messages
```

```
window = tk.Tk() # Création de la fenêtre principale de l'interface graphique
```

```
...
```

```
nameLabel = tk.Label(topFrame, text="Name:")
```

```
entName = tk.Entry(topFrame)
```

```
connectButton = tk.Button(topFrame, text="Connect", command=connect)
```

```
tkMessage = tk.Text(bottomFrame, height=2, width=55)
```

```
tkMessage.bind("<Return>", (lambda event:
```

```
getChatMessage(tkMessage.get("1.0", tk.END))))
```

```
...
```

## 2. Connexion au serveur :

- La fonction **connect\_to\_server** qui crée un socket client TCP (**socket.socket**) et se connecte au serveur (**client.connect**) correspond à cette partie. L'envoi du nom d'utilisateur au serveur (**client.send**) est également inclus ici.

```
def connect_to_server(name):
    global client, HOST_PORT, HOST_ADDR # Déclaration des variables globales client, HOST_PORT et HOST_ADDR
    try:
        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Création du socket client TCP
        client.connect((HOST_ADDR, HOST_PORT)) # Connexion au serveur avec l'adresse et le port spécifiés
        client.send(name.encode("utf-8")) # Envoi du nom d'utilisateur encodé au serveur

        entName.config(state=tk.DISABLED) # Désactivation du champ de nom
        connectButton.config(state=tk.DISABLED) # Désactivation du bouton de connexion
        tkMessage.config(state=tk.NORMAL) # Activation de la zone de message pour envoyer des messages au serveur

        threading.start_new_thread(recv_msg, (client,)) # Démarrage d'un thread pour recevoir les messages du serveur
    except socket.error as e:
        print(e) # Affichage de l'erreur en cas d'échec de connexion
```

## 3. Réception et affichage des messages :

- La fonction **recv\_msg** qui fonctionne en arrière-plan pour recevoir les messages du serveur (**sock.recv**) et les afficher dans la zone de texte correspond à cette fonctionnalité.

```
def recv_msg(sock):
    while True:
        data = sock.recv(4096) # Réception des données/messages du serveur
        if not data: # Vérification si aucune donnée n'est reçue
            break
        # Activation de la zone d'affichage et insertion du texte reçu
        texts = tkDisplay.get("1.0", tk.END).strip()
        tkDisplay.config(state=tk.NORMAL) # Activation de la zone de texte
        if len(texts) < 1:
            tkDisplay.insert(tk.END, data) # Insertion du texte à la fin de la zone de texte
        else:
            tkDisplay.insert(tk.END, "\n\n" + data.decode("utf-8")) # Insertion du texte avec un saut de ligne
            tkDisplay.config(state=tk.DISABLED) # Désactivation de la zone de texte
            tkDisplay.see(tk.END) # Défilement automatique de la zone de texte vers le bas
            print("Recvd from server: "+data.decode("utf-8")) # Affichage du message reçu du serveur dans la console
        sock.close() # Fermeture de la connexion avec le serveur
        window.destroy() # Fermeture de la fenêtre de l'interface graphique
```

## 4. Envoi des messages au serveur :

- La fonction **send\_message\_to\_server** qui envoie les messages au serveur (**client.send**) est la partie du code qui correspond à cette fonctionnalité.

```
def send_message_to_server(msg):
    client.send(msg.encode("utf-8")) # Envoi du message encodé au serveur
    if msg == "exit": # Vérification si le message est "exit" pour fermer la connexion
        client.close() # Fermeture de la connexion avec le serveur
        window.destroy() # Fermeture de la fenêtre de l'interface graphique
    print("Sent message.") # Affichage dans la console pour indiquer que le message a été envoyé
```

### 5. Gestion de l'interface graphique :

- Les lignes de code qui activent ou désactivent les éléments de l'interface en fonction de l'état de la connexion, comme la désactivation du champ de nom après la connexion, correspondent à cette fonctionnalité.

```
entName.config(state=tk.DISABLED)
```

```
connectButton.config(state=tk.DISABLED)
```

```
tkMessage.config(state=tk.NORMAL)
```

### 6. Gestion des événements utilisateur :

- La liaison d'événement pour la touche "Entrée" qui appelle la fonction **getChatMessage** pour envoyer le message correspond à cette partie.

```
tkMessage.bind("<Return>", (lambda event: getChatMessage(tkMessage.get("1.0", tk.END)))) # Liaison de la touche "Entrée" pour envoyer l
```

Ces parties du code client contribuent ensemble à l'architecture d'un chat TCP en fournissant une interface utilisateur interactive, en gérant la connexion au serveur, en recevant et affichant les messages, et en permettant à l'utilisateur d'envoyer des messages au serveur.

## 3. Défis et Solutions :

J'ai utilisé Ubuntu sur Windows en utilisant un environnement de virtualisation, le WSL (Windows Subsystem for Linux). Cela m'a permis de bénéficier des fonctionnalités et de l'écosystème d'Ubuntu tout en conservant Windows comme système d'exploitation principal (ce choix n'est pas volontaire mais plutôt forcé par le fait que je n'ai pas d'espace suffisant dans mon ordinateur pour une machine virtuelle).

L'un des défis auxquels j'ai été confronté était l'affichage des interfaces graphiques (GUI) des applications Ubuntu sur mon environnement Windows. Par défaut, Ubuntu sur Windows ne prend pas en charge l'affichage graphique des applications Linux.

Pour résoudre ce problème, j'ai utilisé Xming, un serveur X Window System pour Windows qui

permet d'afficher les applications graphiques Linux sur un système Windows. Voici comment j'ai procédé :

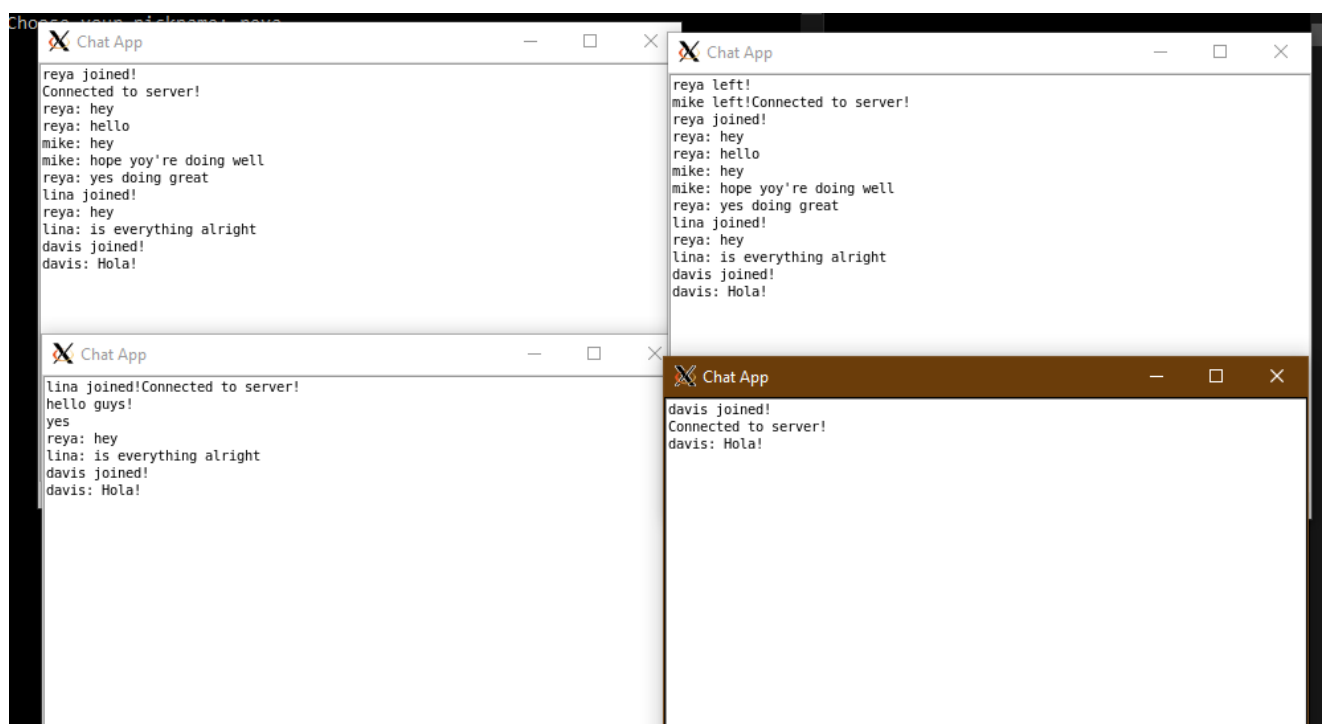
Installation de Xming : J'ai téléchargé et installé Xming sur mon système Windows. Xming crée un serveur X sur Windows qui permet à Ubuntu (ou toute autre distribution Linux) de transférer les interfaces graphiques vers l'environnement Windows.

Configuration de l'export DISPLAY : Une fois Xming installé, j'ai utilisé la commande `export DISPLAY=192.168.118.50:0.0` dans mon terminal Ubuntu pour spécifier à quel serveur X il devait envoyer les interfaces graphiques. Ici, 192.168.118.50 est l'adresse IP de mon système Windows où Xming est exécuté.

Exécution des applications graphiques : Après avoir configuré l'export DISPLAY, j'ai pu lancer des applications graphiques Ubuntu depuis mon terminal. Par exemple, si je voulais exécuter l'éditeur de texte Gedit, je tapais simplement `gedit` dans le terminal, et l'interface graphique de Gedit s'ouvrait sur mon système Windows via Xming.

Ainsi, en utilisant Xming et en configurant l'export DISPLAY correctement, j'ai pu résoudre le problème d'affichage des interfaces graphiques des applications Ubuntu sur mon système Windows. Cela m'a permis de bénéficier pleinement des fonctionnalités de l'environnement Ubuntu tout en travaillant sur Windows.

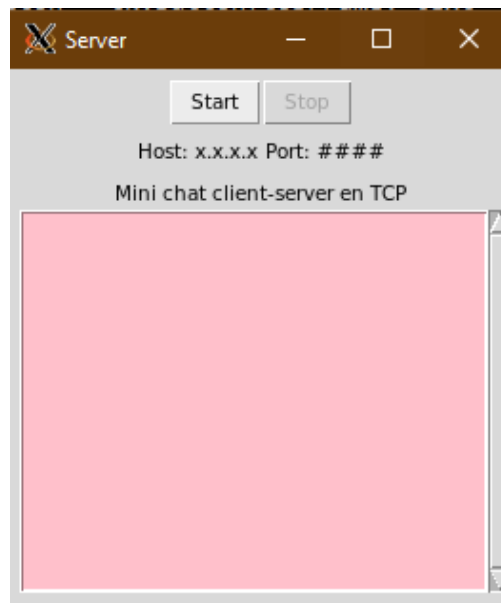
Au début j'ai réalisé cette interface graphique sur Xming que j'ai amélioré ultérieurement car je la trouvais très simple et pas conviviale :



## 4. Compilation et test du code final

- Pour démarrer le serveur de chat, on ouvre le terminal et on tape "**python3 serverchatfin.py**" :

```
rey@DESKTOP-4Q0TJVR:~$ python3 serverchatfin.py
```



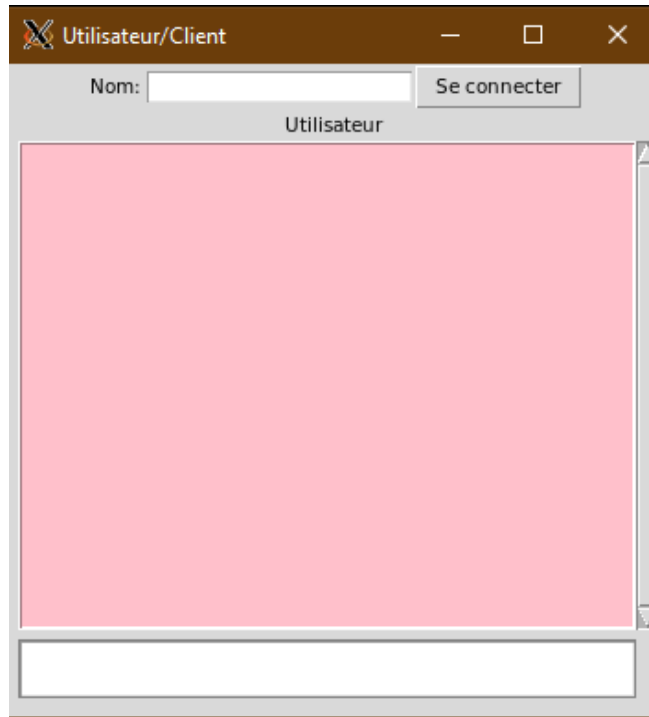
- On appuie ensuite sur le bouton "Start" pour lancer le serveur :



```
rey@DESKTOP-4Q0TJVR:~$ python3 serverchatfin.py
Socket serveur créé/Created server socket
Le serveur écoute/Server is listening
Il faut passer à la fonction d'acceptation des clients./got to accept clients function
```

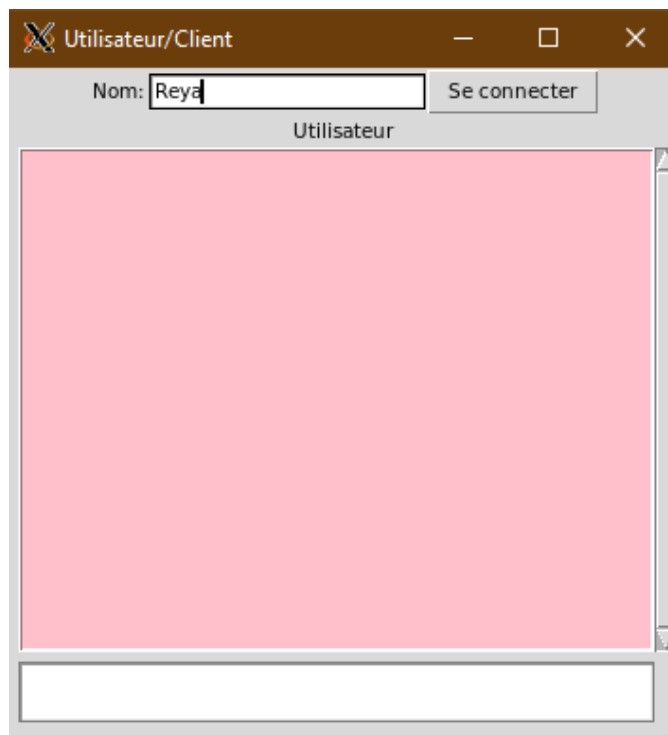
- Pour commencer à discuter, on ouvre un autre terminal et on tape "**python3 clientchatfin.py**" :

```
rey@DESKTOP-4Q0TJVR:~$ python3 clientchatfin.py
```



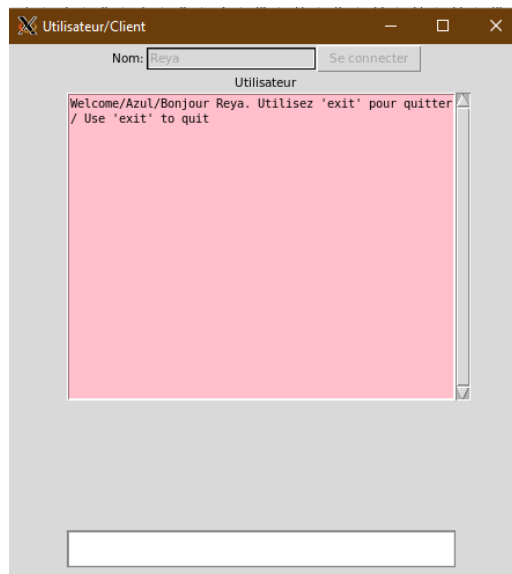
- On doit ensuite entrer un nom ou un pseudonyme et appuyer sur "Se connecter" pour nous

Connecter au serveur :



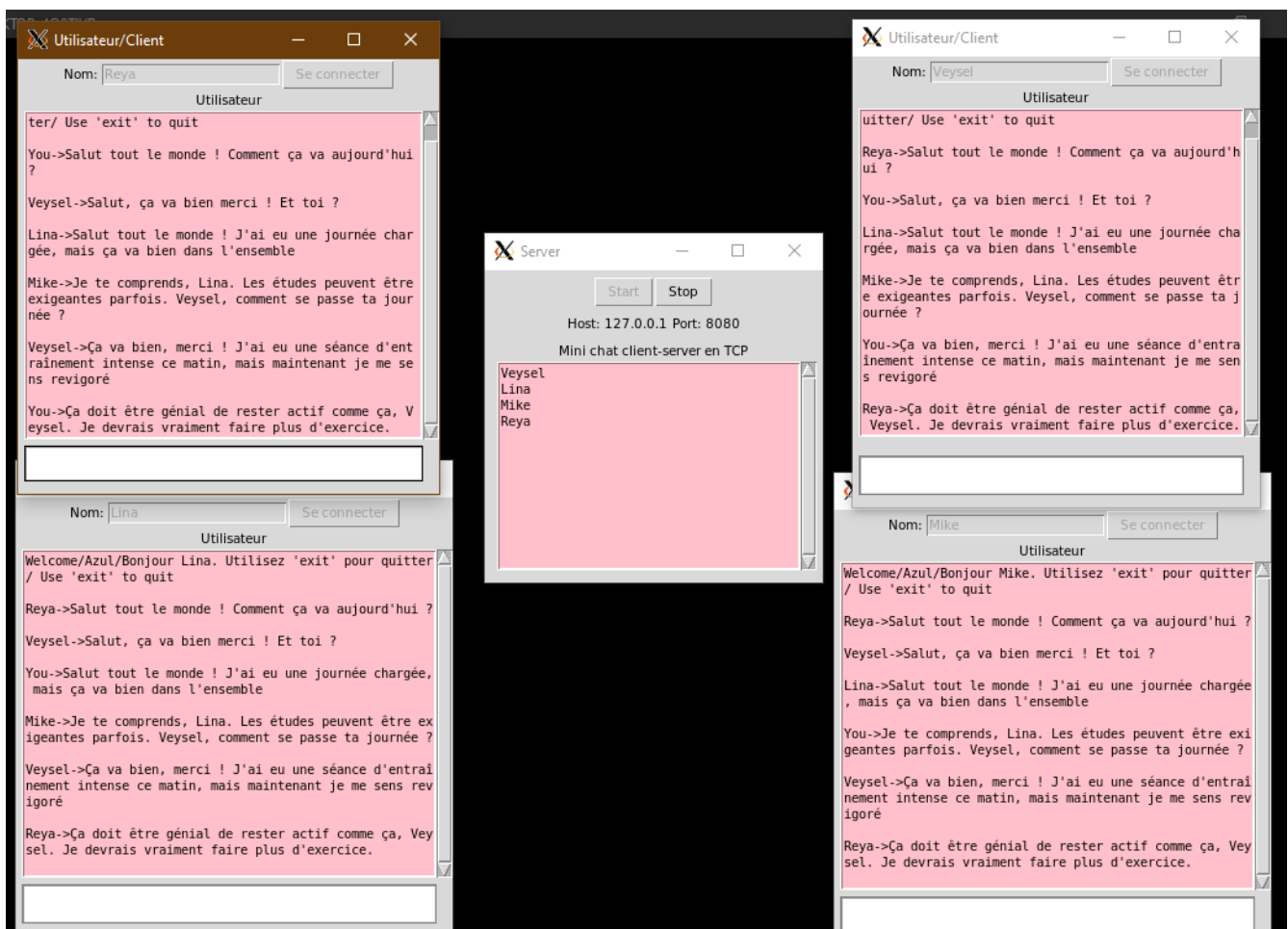
## Développement du mini chat client-serveur en TCP

- On reçoit un message de bienvenue confirmant notre connexion :



```
rey@DESKTOP-4Q0TJVR:~$ python3 clientchatfin.py
Reçu du serveur :/Recvd from server: Welcome/Azul/Bonjour Reya. Utilisez 'exit' pour quitter/ Use 'exit' to quit
```

- Une fois connecté, nous pouvons commencer à envoyer des messages en saisissant du texte dans la zone de texte en bas de la fenêtre de chat.



## 5. Conclusion

Le produit final du mini chat client-serveur, avec clients multiples en se basant sur la connexion TCP en mode graphique représente une solution robuste et conviviale pour la communication en temps réel. En combinant la puissance de Python pour la gestion des connexions réseau et la facilité d'utilisation de Tkinter pour l'interface utilisateur, le chat TCP offre une plateforme interactive et multiplateforme où les utilisateurs peuvent échanger des messages en toute simplicité. Malgré les défis rencontrés, tels que l'intégration d'Ubuntu sur Windows pour l'affichage graphique, l'utilisation de solutions telles que Xming a permis de surmonter ces obstacles et de fournir une expérience fluide aux utilisateurs. En conclusion, le produit final du chat TCP démontre l'efficacité et la flexibilité de Python avec Tkinter pour le développement d'applications réseau interactives et fonctionnelles.

---

# **ANNEXE**

---



## Code du Serveur TCP :

```

rey@DESKTOP-4Q0TJVR: ~
GNU nano 6.2 serverchatfin.py
import tkinter as tk # Importation de la bibliothèque tkinter pour créer une interface graphique
import socket # Importation de la bibliothèque socket pour la communication réseau
import threading # Importation de threading pour gérer les connexions concurrentes

# Création de la fenêtre principale de l'interface graphique
window = tk.Tk()
window.title("Server") # Définition du titre de la fenêtre

# Définition de l'adresse IP et du port du serveur
HOST = "127.0.0.1"
PORT = 8080

# Initialisation des variables pour le nom du client, la liste des clients et la liste des noms des clients
client_name = ""
clients = []
client_names = []

# Fonction pour envoyer et recevoir des messages des clients
def send_rcv_client_msg(client_connection, client_ip_addr):
    global server, client_name, clients, clients_addr # Déclaration des variables globales utilisées dans la fonction
    client_msg = "" # Initialisation du message du client
    client_name = client_connection.recv(4096) # Réception du nom du client
    client_name = client_name.decode("utf-8") # Décodage du nom du client
    print(type(client_name))
    welcome_message = "Welcome/Azul/Bonjour " + client_name + ". Utilisez 'exit' pour quitter/ Use 'exit' to quit" # Message de bienvenue
    client_connection.send(welcome_message.encode("utf-8")) # Envoi du message de bienvenue au client
    client_names.append(client_name) # Ajout du nom du client à la liste des noms de clients

    # Mise à jour de l'affichage des noms de clients sur l'interface graphique
    update_client_names_display(client_names)

while True:
    print("En attente de recevoir un message d'un client./Waiting to receive message from a client")
    data = client_connection.recv(4096) # Réception des données du client
    if not data: break # Sortir de la boucle si aucune donnée n'est reçue
    if data == "exit": break # Sortir de la boucle si le client envoie "exit"

    client_msg = data.decode("utf-8") # Décodage des données du client
    idx = get_client_index(clients, client_connection) # Obtention de l'indice du client dans la liste des clients
    sending_client_name = client_names[idx] # Obtention du nom du client envoyant le message

    for client in clients:
        if client != client_connection:
            message_to_other_clients = sending_client_name + "->" + client_msg # Création du message à envoyer aux autres clients
            client.send(message_to_other_clients.encode("utf-8")) # Envoi du message aux autres clients

    # Suppression du client de la liste
    idx = get_client_index(clients, client_connection)
    del client_names[idx]
    del clients[idx]
    client_connection.close() # Fermeture de la connexion avec le client

    # Mise à jour de l'affichage des noms de clients sur l'interface graphique
    update_client_names_display(client_names)

# Fonction pour obtenir l'indice d'un client dans la liste des clients
def get_client_index(client_list, curr_client):
    for i in range(0, len(client_list)):
        if client_list[i] == curr_client:
            return i
    return -1

# Fonction pour mettre à jour l'affichage des noms de clients sur l'interface graphique
def update_client_names_display(name_list):
    tkDisplay.config(state=tk.NORMAL) # Activation de la zone de texte
    tkDisplay.delete('1.0', tk.END) # Effacement de tout le texte précédent
    for client in name_list:
        tkDisplay.insert(tk.END, client+"\n") # Insertion des noms de clients dans la zone de texte
    tkDisplay.config(state=tk.DISABLED) # Désactivation de la zone de texte

# Fonction pour accepter les connexions des clients
def accept_clients(some_server):
    print("Il faut passer à la fonction d'acceptation des clients./got to accept clients function")
    while True:
        client, addr = some_server.accept() # Acceptation de la connexion d'un client
        print("Le serveur a accepté un client./Server accepted a client")
        clients.append(client) # Ajout du client à la liste des clients
        threading.start_new_thread(send_rcv_client_msg, (client,addr),) # Démarrage d'un thread pour gérer la communication avec le cl

# Fonction pour démarrer le serveur

```

```

# Fonction pour démarrer le serveur
def start_server():
    startButton.config(state=tk.DISABLED) # Désactivation du bouton de démarrage
    stopButton.config(state=tk.NORMAL) # Activation du bouton d'arrêt

    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Création du socket serveur
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # Configuration du socket
    print("Socket serveur créé/Created server socket")
    server.bind((HOST, PORT)) # Liaison du socket à l'adresse et au port spécifiés
    server.listen(5) # Attente de connexions entrantes
    print("Le serveur écoute/Server is listening")
    threading._start_new_thread(accept_clients, (server,)) # Démarrage d'un thread pour accepter les connexions des clients

    hostLabel["text"] = "Host: " + HOST # Mise à jour du label d'adresse IP
    portLabel["text"] = "Port: " + str(PORT) # Mise à jour du label de port

# Fonction pour arrêter le serveur
def stop_server():
    stopButton.config(state=tk.DISABLED) # Désactivation du bouton d'arrêt
    startButton.config(state=tk.NORMAL) # Activation du bouton de démarrage

# Création de la partie supérieure de l'interface graphique avec les boutons de démarrage et d'arrêt du serveur
topFrame = tk.Frame(window)

startButton = tk.Button(topFrame, text="Start", command=start_server)
startButton.pack(side=tk.LEFT)

stopButton = tk.Button(topFrame, text="Stop", command=stop_server, state=tk.DISABLED)
stopButton.pack()
topFrame.pack(side=tk.TOP, pady=(5,0))

# Création de la partie centrale de l'interface graphique avec les labels d'adresse IP et de port
centerFrame = tk.Frame(window)
hostLabel = tk.Label(centerFrame, text="Host: x.x.x.x")
hostLabel.pack(side=tk.LEFT)
portLabel = tk.Label(centerFrame, text="Port: ###")
portLabel.pack()
centerFrame.pack(side=tk.TOP, pady=(5,0))

# Création de la partie inférieure de l'interface graphique pour afficher les noms des clients connectés

# Création de la partie inférieure de l'interface graphique pour afficher les noms des clients connectés
clientsFrame = tk.Frame(window)
clientsLabel = tk.Label(clientsFrame, text="Mini chat client-server en TCP")
clientsLabel.pack() # Ajout du label pour indiquer les clients connectés
scrollBar = tk.Scrollbar(clientsFrame) # Création d'une barre de défilement
scrollBar.pack(side=tk.RIGHT, fill=tk.Y) # Placement de la barre de défilement à droite
tkDisplay = tk.Text(clientsFrame, height=15, width=40) # Création d'une zone de texte pour afficher les noms des clients
tkDisplay.pack(side=tk.LEFT, fill=tk.Y, padx=(5,0)) # Placement de la zone de texte à gauche
scrollBar.config(command=tkDisplay.yview) # Configuration de la barre de défilement pour fonctionner avec la zone de texte
tkDisplay.config(yscrollcommand=scrollBar.set, background="#FFC0CB", highlightbackground="white", state="disabled") # Configuration de la zone de texte
clientsFrame.pack(side=tk.BOTTOM, pady=(5, 10)) # Placement du cadre des clients en bas de la fenêtre

window.mainloop() # Lancement de la boucle principale pour exécuter l'interface graphique

```

## Code du Client TCP :

```

rey@DESKTOP-4QQTIVR: ~
GNU nano 6.2 clientchatfin.py
import tkinter as tk # Importation de la bibliothèque tkinter pour créer une interface graphique
from tkinter import messagebox # Importation de la fonction messagebox de tkinter pour afficher des messages
import socket # Importation de la bibliothèque socket pour la communication réseau
import threading # Importation de threading pour gérer les connexions concurrentes

client = None # Initialisation de la variable client à None
HOST_ADDR = "127.0.0.1" # Définition de l'adresse IP du serveur
HOST_PORT = 8080 # Définition du port du serveur

window = tk.Tk() # Création de la fenêtre principale de l'interface graphique
window.title("Utilisateur/Client") # Définition du titre de la fenêtre

def connect():
    global username, client # Déclaration des variables globales username et client
    if len(entName.get()) < 1: # Vérification si le champ de nom est vide
        tk.messagebox.showerror(title="Erreur/Error!", message="Veuillez entrer votre prénom, <exemple, Reyai>")
    else:
        username = entName.get() # Récupération du nom entré par l'utilisateur
        connect_to_server(username) # Appel à la fonction pour se connecter au serveur

def connect_to_server(name):
    global client, HOST_PORT, HOST_ADDR # Déclaration des variables globales client, HOST_PORT et HOST_ADDR
    try:
        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Création du socket client TCP
        client.connect((HOST_ADDR, HOST_PORT)) # Connexion au serveur avec l'adresse et le port spécifiés
        client.send(name.encode("utf-8")) # Envoi du nom d'utilisateur encodé au serveur

        entName.config(state=tk.DISABLED) # Désactivation du champ de nom
        connectButton.config(state=tk.DISABLED) # Désactivation du bouton de connexion
        tkMessage.config(state=tk.NORMAL) # Activation de la zone de message pour envoyer des messages au serveur

        threading.start_new_thread(recv_msg, (client,)) # Démarrage d'un thread pour recevoir les messages du serveur

    except socket.error as e:
        print(e) # Affichage de l'erreur en cas d'échec de connexion

def recv_msg(sock):
    while True:
        data = sock.recv(4096) # Réception des données/messages du serveur
        if not data: # Vérification si aucune donnée n'est reçue
            break
        # Activation de la zone d'affichage et insertion du texte reçu

        # Activation de la zone d'affichage et insertion du texte reçu
        texts = tkDisplay.get("1.0", tk.END).strip()
        tkDisplay.config(state=tk.NORMAL) # Activation de la zone de texte
        if len(texts) < 1:
            tkDisplay.insert(tk.END, data) # Insertion du texte à la fin de la zone de texte
        else:
            tkDisplay.insert(tk.END, "\n\n" + data.decode("utf-8")) # Insertion du texte avec un saut de ligne
        tkDisplay.config(state=tk.DISABLED) # Désactivation de la zone de texte
        tkDisplay.see(tk.END) # Défilement automatique de la zone de texte vers le bas
        print("Recu du serveur :/Recvd from server: "+data.decode("utf-8")) # Affichage du message reçu du serveur dans la console
    sock.close() # Fermeture de la connexion avec le serveur
    window.destroy() # Fermeture de la fenêtre de l'interface graphique

def getChatMessage(msg):
    msg = msg.replace('\n', '') # Suppression des sauts de ligne dans le message
    texts = tkDisplay.get("1.0", tk.END).strip() # Récupération du texte de la zone de texte
    tkDisplay.config(state=tk.NORMAL) # Activation de la zone de texte
    if len(texts) < 1:
        tkDisplay.insert(tk.END, "You->" + msg, "tag_your_message") # Insertion du message dans la zone de texte
    else:
        tkDisplay.insert(tk.END, "\n\n"+"You->" + msg, "tag_your_message") # Insertion du message avec un saut de ligne
    tkDisplay.config(state=tk.DISABLED) # Désactivation de la zone de texte
    send_message_to_server(msg) # Appel à la fonction pour envoyer le message au serveur
    tkDisplay.see(tk.END) # Défilement automatique de la zone de texte vers le bas
    tkMessage.delete("1.0", tk.END) # Effacement du champ de message après l'envoi

def send_message_to_server(msg):
    client.send(msg.encode("utf-8")) # Envoi du message encodé au serveur
    if msg == "exit": # Vérification si le message est "exit" pour fermer la connexion
        client.close() # Fermeture de la connexion avec le serveur
        window.destroy() # Fermeture de la fenêtre de l'interface graphique
    print("Envoyé un message / Sent message.") # Affichage dans la console pour indiquer que le message a été envoyé

# Création de la partie supérieure de l'interface graphique avec le champ de nom et le bouton de connexion
topFrame = tk.Frame(window)

nameLabel = tk.Label(topFrame, text="Nom:") # Label pour le champ de nom
nameLabel.pack(side=tk.LEFT) # Placement du label à gauche
entName = tk.Entry(topFrame) # Champ de texte pour saisir le nom
entName.pack(side=tk.LEFT) # Placement du champ de texte à gauche
connectButton = tk.Button(topFrame, text="Se connecter", command=connect) # Bouton de connexion
connectButton.pack(side=tk.LEFT) # Placement du bouton à gauche

```

```

topFrame.pack(side=tk.TOP) # Placement de la partie supérieure en haut de la fenêtre

# Création de la partie centrale de l'interface graphique pour afficher les messages du serveur
displayFrame = tk.Frame(window)

headingLine = tk.Label(displayFrame, text="Utilisateur") # Ligne de séparation
headingLine.pack() # Placement de la ligne de séparation
scrollBar = tk.Scrollbar(displayFrame) # Barre de défilement
scrollBar.pack(side=tk.RIGHT, fill=tk.Y) # Placement de la barre de défilement à droite
tkDisplay = tk.Text(displayFrame, height=20, width=55) # Zone de texte pour afficher les messages
tkDisplay.pack(side=tk.LEFT, fill=tk.Y, padx=(5,0)) # Placement de la zone de texte à gauche
scrollBar.config(command=tkDisplay.yview) # Configuration de la barre de défilement pour fonctionner avec la zone de texte
tkDisplay.config(yscrollcommand=scrollBar.set, background="#FFC0CB", highlightbackground="white", state="disabled") # Configuration de la zone de texte
displayFrame.pack(side=tk.TOP) # Placement de la partie centrale en haut de la fenêtre

# Création de la partie inférieure de l'interface graphique pour taper et envoyer des messages
bottomFrame = tk.Frame(window)

tkMessage = tk.Text(bottomFrame, height=2, width=55) # Zone de texte pour taper le message
tkMessage.pack(side=tk.LEFT, padx=(5, 13), pady=(5, 10)) # Placement de la zone de texte à gauche
tkMessage.config(highlightbackground="grey", state="disabled") # Configuration de la zone de texte
tkMessage.bind("<Return>", (lambda event: getChatMessage(tkMessage.get("1.0", tk.END)))) # Liaison de la touche "Entrée" pour envoyer le message
bottomFrame.pack(side=tk.BOTTOM) # Placement de la partie inférieure en bas de la fenêtre

window.mainloop() # Lancement de la boucle principale pour exécuter l'interface graphique

```