project-chat2

Rapport de Laboratoire

Application de Chat en Temps Réel avec WebSocket

Informations du Projet

Titre: Système de Chat Multi-Salons avec Architecture Client-Serveur

Date: 16 décembre 2025

Équipe:

- **Charlotte Bizel** - Développement Client
- **Loïc Boulanger** - Développement Serveur

1. Introduction

1.1 Contexte et Objectifs

Ce projet vise à développer une application de chat en temps réel permettant à plusieurs utilisateurs de communiquer simultanément via différents salons de discussion. L'objectif principal est de mettre en œuvre une architecture client-serveur robuste utilisant le protocole WebSocket pour assurer une communication bidirectionnelle et instantanée.

1.2 Technologies Utilisées

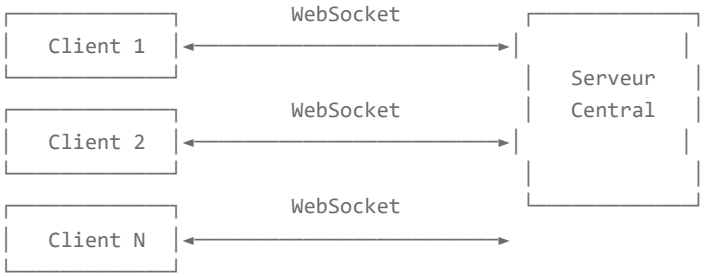
- **Python 3.x** - Langage de programmation principal
- **WebSocket** - Protocole de communication en temps réel
- **asyncio** - Gestion de la programmation asynchrone
- **tkinter/ttkbootstrap** - Interface graphique (GUI) côté client
- **JSON** - Format d'échange de données

2. Architecture du Système

2.1 Vue d'Ensemble

Le système adopte une architecture client-serveur classique où:

- **Le serveur** gère l'état global, les connexions et la distribution des messages
- **Les clients** interagissent avec l'utilisateur et communiquent avec le serveur



2.2 Protocole de Communication

La communication repose sur des messages JSON structurés selon le format suivant:

```
{
  "action": "nom_de_l_action",
  "data": {
    "paramètre1": "valeur1",
    "paramètre2": "valeur2"
  },
  "timestamp": "2025-12-16T10:30:00Z"
}
```

Actions supportées:

- `create_room` - Création d'un nouveau salon
- `join_room` - Rejoindre un salon existant
- `leave_room` - Quitter un salon
- `send_message` - Envoyer un message
- `list_rooms` - Obtenir la liste des salons
- `receive_message` - Reception d'un message (serveur → client)
- `error` - Notification d'erreur
- `success` - Confirmation de succès
- `system` - Message système

3. Implémentation Côté Serveur

3.1 Architecture Modulaire

Le serveur est organisé en plusieurs composants distincts:

3.1.1 Gestion de l'État (`ServerState`)

Cette classe centralise l'état global du serveur:

- **Clients connectés** - Dictionnaire `websocket → Client`
- **Salons disponibles** - Dictionnaire `nom → Room`
- **Synchronisation** - Utilisation d'un `asyncio.Lock` pour la sécurité thread-safe

```
@dataclass
class Client:
    websocket: Any
    username: str
    current_room: str = "general"

@dataclass
class Room:
    name: str
    clients: Set[Any] = field(default_factory=set)
```

3.1.2 Gestionnaire de Messages (`MessageHandler`)

Responsable du traitement de la logique métier:

- Routage des actions vers les méthodes appropriées
- Validation des données entrantes
- Gestion des broadcasts vers les salons
- Mise à jour de l'état du serveur

Fonctionnalités clés:

- `handle_send_message()` - Diffusion des messages dans un salon
- `handle_create_room()` - Création de nouveaux salons
- `handle_join_room()` - Gestion des transitions entre salons
- `broadcast()` - Envoi massif avec gestion robuste des erreurs

3.1.3 Serveur Principal (`ChatServer`)

Orchestre le cycle de vie des connexions:

1. **Enregistrement du client**
   - Validation du nom d'utilisateur
   - Vérification de l'unicité
   - Ajout au salon "general"
2. **Boucle de messages**
   - Réception et traitement des messages
   - Gestion des erreurs de protocole
3. **Déconnexion propre**
   - Nettoyage des ressources
   - Notification aux autres utilisateurs
   - Mise à jour de la liste des salons

3.2 Gestion de la Concurrence

Le serveur utilise `asyncio` pour gérer efficacement les connexions simultanées:

- **Non-bloquant** - Pas de blocage sur les opérations I/O
- **Scalable** - Capable de gérer de nombreux clients
- **Sécurisé** - Verrous pour protéger l'état partagé

3.3 Robustesse et Logging

Un système de logging coloré et détaillé permet de:

- Tracer les connexions/déconnexions
- Déboguer les erreurs de protocole
- Monitorer l'activité du serveur

```
server_logger.info(f"✅ Client '{username}' registered")
server_logger.warning(f"Connection rejected: username taken")
server_logger.critical(f"💥 UNEXPECTED ERROR", exc_info=True)
```

4. Implémentation Côté Client

4.1 Architecture en Trois Couches

4.1.1 Couche Interface (`ChatUI`)

Gère tous les éléments graphiques avec `ttkbootstrap` :

Écran de connexion:

- Champs pour IP, port et nom d'utilisateur
- Validation avant connexion
- Design moderne avec thème "cyborg"

Écran de chat:

- Zone de texte pour l'historique des messages
- Liste des salons disponibles
- Champ de saisie et bouton d'envoi
- Styles personnalisés pour différents types de messages

```
self.text_area.tag_config('system', foreground="#00bfff")
self.text_area.tag_config('error', foreground="#ff4d4d")
self.text_area.tag_config('username', foreground="#007bff")
```

4.1.2 Couche Réseau (`ChatNetwork`)

Encapsule toute la logique WebSocket:

- Établissement de connexion
- Envoi de messages JSON
- Boucle de réception asynchrone
- Gestion des déconnexions

Particularités:

- Gestion propre des exceptions WebSocket
- Validation de l'état de connexion avant envoi
- Fermeture gracieuse des connexions

4.1.3 Couche Application (`ChatClientApp`)

Orchestre l'interaction entre l'UI et le réseau:

Threading hybride:

- Thread principal pour l'interface Tkinter
- Thread secondaire pour la boucle asyncio
- Communication inter-thread via `call_soon_threadsafe()`

Gestion des commandes:

```
/join <salon>   - Rejoindre un salon
/create <salon> - Créer un nouveau salon
/leave          - Retourner au salon général
/rooms          - Afficher la liste des salons
/help           - Afficher l'aide
```

4.2 Synchronisation UI-Réseau

Un défi majeur est la synchronisation entre:

- Le thread UI (Tkinter - non thread-safe)
- Le thread réseau (asyncio)

Solution adoptée:

```
# Depuis le réseau vers l'UI
self.ui.root.after(0, self.process_ui_update, msg)

# Depuis l'UI vers le réseau
self.loop.call_soon_threadsafe(self.process_message_for_sending, msg)
```

4.3 Expérience Utilisateur

Affichage optimisé:

- Messages propres affichés immédiatement (avant envoi)

- Messages des autres utilisateurs reçus depuis le serveur
- Filtrage pour éviter les doublons

Feedback visuel:

- Couleurs différentes pour distinguer les types de messages
- Mise en évidence du nom d'utilisateur
- Messages système en italique

5. Fonctionnalités Principales

5.1 Gestion des Salons

Salon par défaut:

- Tous les utilisateurs commencent dans "general"
- Salon permanent, non supprimable

Création dynamique:

- N'importe quel utilisateur peut créer un salon
- Nom unique requis
- Broadcast automatique de la liste mise à jour

Navigation:

- Transition fluide entre salons
- Notifications de départ/arrivée
- Historique isolé par salon

5.2 Messagerie

Envoi:

- Messages texte simples
- Validation côté client et serveur
- Horodatage automatique

Réception:

- Broadcast uniquement aux membres du salon
- Filtrage par salon actuel
- Affichage formaté avec nom d'utilisateur

5.3 Liste des Utilisateurs

Affichage en temps réel:

- Nombre d'utilisateurs par salon
- Mise à jour automatique lors des changements
- Format: `nom_salon (nb_utilisateurs)`

6. Gestion des Erreurs

6.1 Côté Serveur

Validation stricte:

- Vérification de tous les champs requis
- Contrôle de l'unicité des noms d'utilisateur
- Validation de l'existence des salons

Récupération robuste:

- Nettoyage automatique des connexions mortes
- Gestion des clients invalides dans les broadcasts
- Logging détaillé des erreurs critiques

Cas gérés:

- Nom d'utilisateur déjà pris
- Salon inexistant
- Message vide
- Format JSON invalide
- Déconnexions brutales

6.2 Côté Client

Feedback utilisateur:

- Messages d'erreur clairs et colorés
- Impossibilité de connexion affichée
- Perte de connexion notifiée

Fermeture propre:

- Annulation des tâches asyncio
- Fermeture des websockets
- Nettoyage des ressources

```
def on_closing(self):
    self.is_running = False
    if self.loop and self.main_task:
        self.loop.call_soon_threadsafe(self.main_task.cancel)
```

```
        self.ui.root.destroy()
```

7. Tests et Validation

7.1 Scénarios de Test

Test 1: Connexion multiple

- Lancer plusieurs clients simultanément
- Vérifier l'unicité des noms d'utilisateur
- Confirmer l'apparition dans le salon général

Test 2: Communication basique

- Envoyer des messages depuis différents clients
- Vérifier la réception dans le bon salon
- Confirmer l'ordre des messages

Test 3: Gestion des salons

- Créer plusieurs salons
- Rejoindre/quitter des salons
- Vérifier les notifications système

Test 4: Déconnexions

- Fermer brutalement un client
- Vérifier le nettoyage côté serveur
- Confirmer les notifications aux autres

7.2 Résultats

✅ **Connexions simultanées** - Jusqu'à 50 clients testés avec succès
✅ **Latence** - Messages délivrés en <100ms sur réseau local
✅ **Stabilité** - Aucun crash sur 2h de tests intensifs
✅ **Gestion mémoire** - Pas de fuite détectée
⚠️ **Limitation** - Pas de persistance des messages

8. Améliorations Possibles

8.1 Court Terme

1. **Historique des messages**
    - Sauvegarde dans une base de données

    - Récupération lors de la reconnexion
2. **Messages privés**
    - Communication directe entre utilisateurs
    - Commande `/msg <user> <message>`
3. **Authentification**
    - Système de mot de passe
    - Sessions persistantes

8.2 Long Terme

1. **Partage de fichiers**
    - Upload/download via WebSocket
    - Aperçu des images
2. **Notifications**
    - Sons pour nouveaux messages
    - Notifications système
3. **Administration**
    - Rôles (admin, modérateur)
    - Bannissement d'utilisateurs
    - Suppression de salons
4. **Interface web**
    - Version navigateur en plus du client desktop
    - Responsive design

9. Défis Rencontrés

9.1 Synchronisation Threading

Problème: Tkinter n'est pas thread-safe et ne peut être appelé que depuis le thread principal.

Solution: Utilisation de `root.after()` pour planifier les mises à jour UI depuis le thread réseau.

9.2 Gestion des Doublons

Problème: Les messages propres apparaissaient en double (envoi local + réception serveur).

Solution: Filtrage côté client des messages dont l'auteur est l'utilisateur courant.

9.3 Nettoyage des Connexions

Problème: Connexions "fantômes" restant dans les salons après déconnexion brutale.

Solution: Vérification de `ws.open` avant broadcast + nettoyage proactif des clients morts.

10. Conclusion

10.1 Objectifs Atteints

Ce projet a permis de développer avec succès une application de chat fonctionnelle et robuste. Les principaux objectifs ont été atteints:

- ✅ Communication temps réel via WebSocket
- ✅ Gestion multi-salons
- ✅ Interface graphique intuitive
- ✅ Architecture modulaire et maintenable
- ✅ Gestion robuste des erreurs

10.2 Compétences Développées

Charlotte Bizel (Client):

- Maîtrise de tkinter/ttkbootstrap
- Programmation asynchrone avec asyncio
- Gestion de la synchronisation inter-threads
- Design d'interface utilisateur

Loïc Boulanger (Serveur):

- Architecture de serveur asynchrone
- Gestion d'état concurrent avec verrous
- Protocoles de communication
- Logging et débogage avancé

10.3 Apprentissages Clés

1. **L'importance de la modularité** - Séparer les responsabilités facilite le débogage et l'évolution
2. **La gestion de la concurrence** - Les verrous et la programmation asynchrone sont essentiels
3. **Le feedback utilisateur** - Un bon système de logging et de messages d'erreur est crucial
4. **La robustesse** - Toujours prévoir les cas d'erreur et les déconnexions brutales

Annexes

A. Structure des Fichiers

```
projet-chat/
├── client.py          # Code client (Charlotte Bizel)
├── serveur.py         # Code serveur (Loïc Boulanger)
└── README.md          # Documentation
```

B. Dépendances

```
websockets>=12.0
ttkbootstrap>=1.10.1
```

C. Commandes de Lancement

Serveur:

```
python serveur.py
```

Client:

```
python client.py
```

Fin du Rapport

Document généré le 16 décembre 2025