

## Table des matières

Introduction .....	4
1. Liste des Use cases réalisés.....	5
2. USE CASE 1 : Client Node JS / Serveur Node JS avec TCP/IP .....	5
2.1. Fonctionnalités implémentées .....	5
2.2. Fonctionnalités non implémentées .....	6
2.3. Définition du protocole (format des messages client/serveur) .....	6
2.3.1. Message de bienvenue du serveur.....	6
2.3.2. Envoi de messages privés.....	6
2.3.3. Envoi de messages broadcast .....	7
2.3.4. Lister les utilisateurs connectés.....	7
2.3.5. Quitter le chat (se déconnecter) proprement .....	8
2.3.6. Quitter le chat (se déconnecter) de façon forcée (CTRL + C) .....	8
2.3.7. Notification lorsque quelqu'un se connecte .....	8
2.3.8. Notification lorsque quelqu'un se déconnecte .....	8
2.3.9. Créer un groupe.....	9
2.3.10. Rejoindre un groupe .....	9
2.3.11. Envoi de message dans un groupe .....	9
2.3.12. Lister les membres d'un groupe .....	10
2.3.13. Lister les messages d'un groupe.....	10
2.3.14. Lister tous les groupes existants .....	11
2.3.15. Quitter un groupe.....	11
2.3.16. Ajouter quelqu'un dans un groupe .....	11
2.3.17. Retirer quelqu'un d'un groupe .....	12
2.3.18. Bannir quelqu'un d'un groupe.....	12
2.3.19. Débannir quelqu'un d'un groupe.....	13
2.3.20. Lister tous les événements qui se sont produits dans un groupe .....	13

2.3.21.	Télécharger sa session .....	13
2.3.22.	Authentification par mot de passe .....	14
2.4.	Description de la base de données .....	14
2.4.1.	La table users.....	14
2.4.2.	La table groups .....	15
2.4.3.	La table groups_users.....	15
2.4.4.	La table bans.....	15
2.4.5.	La table sentMessages .....	16
2.5.	Explication du fonctionnement.....	16
2.6.	Quelques illustrations .....	17
2.6.1.	Illustration de quelques commandes (1).....	17
2.6.2.	Illustration de quelques commandes (2).....	18
2.6.3.	Illustration de quelques commandes (3).....	20
2.6.4.	Illustration de l'authentification .....	21
3.	USE CASE 2 : Client Node JS / Serveur Node JS avec websocket .....	21
3.1.	Fonctionnalités implémentées .....	21
3.2.	Fonctionnalités non implémentées .....	21
3.3.	Définition du protocole (format des messages client/serveur) .....	21
3.4.	Description de la base de données .....	21
3.5.	Explication du fonctionnement.....	21
4.	USE CASE 3 : Client Node JS / Serveur ESP32 avec TCP/IP.....	22
4.1.	Fonctionnalités implémentées .....	22
4.2.	Définition du protocole (format des messages client/serveur) .....	23
4.3.	Explication du fonctionnement.....	23
5.	USE CASE 4 : Client ESP32 / Serveur Node JS avec TCP/IP.....	23
5.1.	Fonctionnalités implémentées .....	23
5.2.	Définition du protocole (format des messages client/serveur) .....	23

5.3. Explication du fonctionnement..... 23

5.4. Quelques illustrations ..... 24

Conclusion..... 26

## Introduction

**L'Internet des objets** ou **Internet of Things (IoT)** décrit le réseau de terminaux physiques, les « **objets** », qui intègrent des capteurs, des logiciels et d'autres technologies en vue de se connecter à d'autres terminaux et systèmes sur Internet et d'échanger des données avec eux. Cette interconnexion et ces échanges de données s'effectuent via un protocole ou environnement **client-serveur** qui désigne un mode de transaction (souvent à travers un réseau) entre plusieurs programmes ou processus : l'un, qualifié de **client**, envoie des requêtes ; l'autre, qualifié de **serveur**, attend les requêtes des clients et y répond. Ainsi, dans le cadre d'un projet de l'UE Electronique et Interfaçage, il nous a été demandé d'implémenter cette architecture client-serveur dans une application de Chat basée sur les protocoles **TCP/IP** et **websocket**. Ce projet se décompose en des **use cases** qui se distinguent soit par le protocole utilisé, soit par l'équipement (ordinateur ou ESP32), tant du côté client que du côté serveur. Dans la suite de notre travail, nous allons présenter les **use cases** que nous avons implémentés.

## 1. Liste des Use cases réalisés

Nous avons réalisé les **quatre (04)** use case suivants :

- Client Node JS / Serveur Node JS avec TCP/IP
- Client Node JS / Serveur Node JS avec websocket
- Client Node JS/ Serveur ESP32 avec TCP/IP
- Client ESP32 / Serveur Node JS avec TCP/IP

Pour chacun des use cases, nous présenterons les fonctionnalités implémentées, les fonctionnalités non implémentées, l'explication du fonctionnement ainsi que les formats des messages envoyés par le client et le serveur.

## 2. USE CASE 1 : Client Node JS / Serveur Node JS avec TCP/IP

### 2.1. Fonctionnalités implémentées

1. Utilisation des variables d'environnement pour configurer le 'host' et le 'port'
2. Utilisation d'un système de debug en lieu et place de 'console.log'
3. Utilisation de la librairie readline à la place de process.stdin
4. Utilisation de la librairie yargs pour gérer les arguments du programme
5. Message de bienvenue du serveur
6. Envoi de messages privés
7. Envoi de messages broadcast
8. Lister les utilisateurs connectés
9. Quitter le chat (se déconnecter)
10. Notifications (avertir l'utilisateur quand une personne se connecte ou se déconnecte)
11. Amélioration du code avec l'utilisation des modules
12. Affichage des couleurs dans la console
13. Créer un groupe
14. Rejoindre un groupe
15. Envoi de message dans un groupe
16. Lister les membres d'un groupe
17. Lister les messages d'un groupe
18. Lister tous les groupes existants
19. Quitter un groupe

20. Ajouter quelqu'un dans un groupe
21. Retirer quelqu'un d'un groupe
22. Bannir quelqu'un d'un groupe
23. Débannir quelqu'un d'un groupe
24. Lister tous les événements qui se sont produits dans un groupe
25. Utilisation d'une base de données pour stocker les données relatives au bon fonctionnement du tchat (groupes, utilisateurs, messages, événements...)
26. Restaurer la session de l'utilisateur lorsqu'il se reconnecte.
27. Télécharger sa session
28. Authentification par mot de passe
29. Chiffrement des mots de passe avec bcrypt

## 2.2. Fonctionnalités non implémentées

- Gestion des groupes privés
- Supprimer sa session
- Chiffrement des messages par l'algorithme de Diffie-Hellman

## 2.3. Définition du protocole (format des messages client/serveur)

### 2.3.1. Message de bienvenue du serveur

Format du message du client	<code>{from: sender_name ,action:'client-hello'}</code>
Format de réponse du serveur	<code>{from: sender_name,action:'server-hello', msg: message_content}</code>
Description	<code>sender_name : nom du client</code>

### 2.3.2. Envoi de messages privés

Syntaxe de la commande	<code>s;receiver_name;message_content</code>
Format du message du client	<code>{from: sender_name, to: receiver_name ,msg: message_content ,action:'client-send'}</code>

Format de réponse du serveur	<code>{from: sender_name, to: receiver_name ,msg: message_content ,action:'server-send'}</code>
Description	<p><code>sender_name</code> : nom de l'expéditeur du message</p> <p><code>receiver_name</code> : nom du destinataire</p> <p><code>message_content</code> : contenu du message</p>

### 2.3.3. Envoi de messages broadcast

Syntaxe de la commande	<b><code>b;message_content</code></b>
Format du message du client	<code>{from: sender_name ,msg: message_content ,action:'client-broadcast'}</code>
Format de réponse du serveur	<code>{from: sender_name ,msg: message_content ,action:'server-broadcast'}</code>
Description	<code>message_content</code> : contenu du message

### 2.3.4. Lister les utilisateurs connectés

Syntaxe de la commande	<b><code>ls;</code></b>
Format du message du client	<code>{from: sender_name ,action:'client-list-clients'}</code>
Format de réponse du serveur	<code>{from: sender_name ,action:'client-list-clients', list : client-list}</code>
Description	<p><code>sender_name</code> : nom du client qui a tapé la commande</p> <p><code>client-list</code> : un tableau contenant les noms des utilisateurs connectés</p>

**2.3.5. Quitter le chat (se déconnecter) proprement**

<b>Syntaxe de la commande</b>	<b>q;</b>
Format du message du client	{from: sender_name ,action:'client-quit'}
Format de réponse du serveur	{from: sender_name ,action:'server-quit'}
Description	sender_name : nom du client qui a tapé la commande

**2.3.6. Quitter le chat (se déconnecter) de façon forcée (CTRL + C)**

Format du message du client	{from: sender_name, code: error_code ,msg: error_msg, action:'client-error'}
Format de réponse du serveur	{from: sender_name , code: error_code ,msg: error_msg, action:'server-error'}
Description	sender_name : nom du client qui a tapé la commande error_code : le code d'erreur error_msg : le message d'erreur

**2.3.7. Notification lorsque quelqu'un se connecte**

Le serveur reçoit un message de type 'client-hello' venant du client qui se connecte ; ensuite il envoie un message aux autres utilisateurs.

Format de message du server	{from: sender_name, action: 'server-someone-arrived'}
Description	sender_name : le nom du client qui s'est connecté

**2.3.8. Notification lorsque quelqu'un se déconnecte**



Le serveur reçoit un message de type 'client-quit' ou 'client-error' venant du client qui se déconnecte ; ensuite il envoie un message aux autres utilisateurs

Format de message du serveur	<code>{from: sender_name, type: deconnection_type, action: 'server-someone-left'}</code>
Description	<p><code>sender_name</code> : le nom du client qui s'est déconnecté</p> <p><code>deconnection_type</code> : le type de déconnexion. Il vaut soit 'normal' soit 'forced'</p>

### 2.3.9. Créer un groupe

Syntaxe de la commande	<b><code>cg;group_name</code></b>
Format du message du client	<code>{from: sender_name, group: group_name, action: 'cgroup'}</code>
Format de réponse du serveur	<code>{from: sender_name, group: group_name, action: 'cgroup'}</code>
Description	<p><code>sender_name</code> : le nom du client</p> <p><code>group_name</code> : le nom du groupe qu'il veut créer</p>

### 2.3.10. Rejoindre un groupe

Syntaxe de la commande	<b><code>j;group_name</code></b>
Format du message du client	<code>{from: sender_name, group: group_name, action: 'join'}</code>
Format de réponse du serveur	<code>{from: sender_name, group: group_name, action: 'join'}</code>
Description	<p><code>sender_name</code> : le nom du client</p> <p><code>group_name</code> : le nom du groupe qu'il veut rejoindre</p>

### 2.3.11. Envoi de message dans un groupe

<b>Syntaxe de la commande</b>	<b>bg;group_name;message_content</b>
Format du message du client	{from: sender_name ,group: group_name, msg: message_content ,action:'gbroadcast'}
Format de réponse du serveur	{from: sender_name ,group: group_name, msg: message_content ,action:'gbroadcast'}
Description	sender_name : le nom du client qui envoie group_name : le nom du groupe message_content : le message envoyé

### 2.3.12. Lister les membres d'un groupe

<b>Syntaxe de la commande</b>	<b>members;group_name</b>
Format du message du client	{from: sender_name ,group: 'group_name', action:'members'}
Format de réponse du serveur	{from: sender_name ,group: 'group_name', action:'members', list : members-list}
Description	sender_name : le nom du client group_name : le nom du groupe members-list: un tableau contenant les noms de tous les membres du groupe.

### 2.3.13. Lister les messages d'un groupe

<b>Syntaxe de la commande</b>	<b>messages;group_name</b>
Format du message du client	{from: sender_name ,group: 'group_name', action:'msgs'}
Format de réponse du serveur	{from: sender_name ,group: 'group_name', action:'msgs', list : messages-list}
Description	sender_name : le nom du client group_name : le nom du groupe

	messages-list: un tableau contenant les messages du groupe.
--	---

#### 2.3.14. Lister tous les groupes existants

<b>Syntaxe de la commande</b>	<b>groups;</b>
Format du message du client	{from: sender_name ,action:'groups'}
Format de réponse du serveur	{from: sender_name ,action:'groups', list : groups-list}
Description	sender_name : le nom du client groups-list: un tableau contenant les noms de tous les groupes.

#### 2.3.15. Quitter un groupe

<b>Syntaxe de la commande</b>	<b>leave;group_name</b>
Format du message du client	{from: sender_name, group: group_name ,action:'leave'}
Format de réponse du serveur	{from: sender_name, group: group_name ,action:'leave'}
Description	sender_name : le nom du client group_name : le nom du groupe

NB : lorsque quelqu'un quitte un groupe, on notifie les autres membres du groupe.

#### 2.3.16. Ajouter quelqu'un dans un groupe

<b>Syntaxe de la commande</b>	<b>invite;group_name;dest</b>
Format du message du client	{from: sender_name , group: 'group_name', dest: receiver_name, action:'invite'}
Format de réponse du serveur	{from: sender_name , group: 'group_name', dest: receiver_name, action:'invite'}

Description	<p>sender_name : le nom du client (celui qui a tapé la commande)</p> <p>group_name : le nom du groupe</p> <p>dest (receiver_name) : le nom de celui qu'on veut ajouter dans le groupe</p>
-------------	---

NB : lorsqu'on ajoute quelqu'un dans un groupe, les autres membres du groupe sont notifiés.

### 2.3.17. Retirer quelqu'un d'un groupe

Syntaxe de la commande	<b>kick;group_name;dest;reason</b>
Format du message du client	{from: sender_name , group: group_name, dest: receiver_name, reason: reason, action:'kick'}
Format de réponse du serveur	{from: sender_name , group: group_name, dest: receiver_name, reason: reason, action:'kick'}
Description	<p>sender_name : le nom du client (celui qui a tapé la commande)</p> <p>group_name : le nom du groupe</p> <p>dest (receiver_name) : le nom de celui qu'on veut retirer du groupe</p> <p>reason : la raison pour laquelle on le retire du groupe</p>

### 2.3.18. Bannir quelqu'un d'un groupe

Syntaxe de la commande	<b>ban;group_name;dest;reason</b>
Format du message du client	{from: sender_name , group: group_name, dest: receiver_name, reason: reason, action:'ban'}
Format de réponse du serveur	{from: sender_name , group: group_name, dest: receiver_name, reason: reason, action:'ban'}
Description	<p>sender_name : le nom du client (celui qui a tapé la commande)</p> <p>group_name : le nom du groupe</p> <p>dest (receiver_name) : le nom de celui qu'on veut bannir du groupe</p>

	reason : la raison pour laquelle on le bannit du groupe
--	---

NB : lorsqu'on bannit quelqu'un d'un groupe, les autres membres du groupe sont notifiés.

### 2.3.19. Débannir quelqu'un d'un groupe

<b>Syntaxe de la commande</b>	<b>unban;group_name;dest</b>
Format du message du client	{from: sender_name , group: 'group_name', dest: receiver_name, action:'unban'}
Format de réponse du serveur	{from: sender_name , group: 'group_name', dest: receiver_name, action:'unban'}
Description	sender_name : le nom du client (celui qui a tapé la commande) group_name : le nom du groupe dest (receiver_name) : le nom de celui qu'on veut débannir du groupe

### 2.3.20. Lister tous les événements qui se sont produits dans un groupe

<b>Syntaxe de la commande</b>	<b>states;group_name</b>
Format du message du client	{from: sender_name , group: group_name, action:'states'}
Format de réponse du serveur	{from: sender_name , group: group_name, action:'states', list : events-list}
Description	sender_name : le nom du client (celui qui a tapé la commande) group_name : le nom du groupe events-list : la liste(tableau) de tous les événements survenus dans le groupe

### 2.3.21. Télécharger sa session

L'utilisateur peut télécharger sa session dans un fichier texte. Le nom du fichier contient le nom de l'utilisateur et la date de téléchargement, afin de pouvoir mieux se repérer.

<b>Syntaxe de la commande</b>	<b>sdownload;</b>
Format du message du client	{from: sender_name , action:'session-download'}
Format de réponse du serveur	{ from: sender_name , action:'session-download', list : messages-list}
Description	sender_name : le nom du client (celui qui a tapé la commande) messages-list : la liste (tableau) contenant tous les messages et événements de la session du client

### 2.3.22. Authentification par mot de passe

L'utilisateur lance l'application client en spécifiant son nom et son mot de passe. Exemple :

```
\client>node client --name Robert --pass xxxxxxxx
```

Format du message du client	{from: sender_name, pass : password, action:'client-hello'}
Format de réponse du serveur	<i>En cas de réussite</i> {from: sender_name, action:'server-hello', msg: message_content}
	<i>En cas d'échec</i> {from: sender_name, action:'unknown-user'}
Description	sender_name : nom du client password : mot de passe

## 2.4. Description de la base de données

On utilise une base de données **sqlite** pour stocker les données relatives au bon fonctionnement du tchat (groupes, utilisateurs, messages, événements...). Cette base de données contient 5 tables :

### 2.4.1. La table users

Elle contient les utilisateurs du tchat. Son schéma est le suivant :

```
CREATE TABLE IF NOT EXISTS "users" (  
    "id"      INTEGER NOT NULL,  
    "username" TEXT NOT NULL UNIQUE,  
    "password" TEXT NOT NULL DEFAULT '',  
    PRIMARY KEY ("id" AUTOINCREMENT)  
);
```

#### 2.4.2. La table groups

Elle contient les groupes. Son schéma est le suivant :

```
CREATE TABLE IF NOT EXISTS "groups" (  
    "id"      INTEGER NOT NULL,  
    "groupname" TEXT NOT NULL UNIQUE,  
    PRIMARY KEY ("id" AUTOINCREMENT)  
);
```

#### 2.4.3. La table groups\_users

Elle contient les associations utilisateur-groupe (permet de savoir quel client est dans quel groupe). Son schéma est le suivant :

```
CREATE TABLE IF NOT EXISTS "groups_users" (  
    "id"      INTEGER NOT NULL,  
    "groupname" TEXT NOT NULL,  
    "username" TEXT NOT NULL,  
    CONSTRAINT "fk_groupname" FOREIGN KEY("groupname") REFERENCES  
"groups"("groupname"),  
    CONSTRAINT "fk_username" FOREIGN KEY("username") REFERENCES  
"users"("username"),  
    PRIMARY KEY ("id" AUTOINCREMENT),  
    UNIQUE("groupname", "username")  
);
```

#### 2.4.4. La table bans

Elle permet de savoir quels utilisateurs sont bannis de quels groupes. Son schéma est le suivant :

```
CREATE TABLE IF NOT EXISTS "bans" (  
    "id"      INTEGER NOT NULL,  
    "groupname" TEXT NOT NULL,  
    "username" TEXT NOT NULL,
```

```

    CONSTRAINT "fk_grouptime" FOREIGN KEY("grouptime") REFERENCES
"groups"("grouptime"),
    CONSTRAINT "fk_username" FOREIGN KEY("username") REFERENCES
"users"("username"),
    PRIMARY KEY ("id" AUTOINCREMENT),
    UNIQUE("grouptime", "username")
);

```

#### 2.4.5. La table sentMessages

Elle contient les messages (de tous types) échangés entre le serveur et les clients. L'attribut « action » nous permet de savoir de quel type de message il s'agit. L'attribut « time » permet de stocker le moment (date et heure) auquel le message a été émis. Ci-après le schéma de la table sentMessages.

```

CREATE TABLE IF NOT EXISTS "sentMessages" (
    "id"      INTEGER NOT NULL,
    "from"    TEXT,
    "to"      TEXT,
    "dest"    TEXT,
    "group"   TEXT,
    "msg"     TEXT,
    "action"   TEXT,
    "reason"   TEXT,
    "time"    TEXT,
    PRIMARY KEY ("id" AUTOINCREMENT)
);

```

### 2.5. Explication du fonctionnement

Le principe général de fonctionnement de l'application est le suivant :

- L'utilisateur se connecte grâce à son nom et son mot de passe. S'il s'agit de sa première connexion alors ses informations sont enregistrées dans la base de données. Sinon, on vérifie que les informations (nom et mot de passe) sont correctes.
- L'utilisateur saisit une commande
- La commande est analysée grâce à des expressions régulières pour déterminer l'action à effectuer. L'ensemble des expressions régulières utilisées ont été mises dans un module nommé *messagesRegex.js* dont voici le contenu



```

const messagesTypes = {
  MSG_SEND_TO_ONE : /^s;(.*?);(.*?)$/i,
  MSG_SEND_TO_ALL : /^b;(.*?)$/i,
  MSG_LIST_USERS : /^ls;$/i,
  MSG_QUIT : /^q;$/i,

  MSG_CREATE_GROUP : /^cg;(.*?)$/i,
  MSG_JOIN_GROUP : /^j;(.*?)$/i,
  MSG_BROADCAST_GROUP : /^bg;(.*?);(.*?)$/i,
  MSG_LIST_GROUP_MEMBERS : /^members;(.*?)$/i,
  MSG_LIST_GROUP_MESSAGES : /^messages;(.*?)$/i,
  MSG_LIST_GROUPS : /^groups;$/i,
  MSG_LEAVE_GROUP : /^leave;(.*?)$/i,
  MSG_INVITE : /^invite;(.*?);(.*?)$/i,
  MSG_KICK : /^kick;(.*?);(.*?);(.*?)$/i,
  MSG_BAN : /^ban;(.*?);(.*?);(.*?)$/i,
  MSG_UNBAN : /^unban;(.*?);(.*?)$/i,
  MSG_LIST_GROUP_EVENTS : /^states;(.*?)$/i,
  MSG_SESSION_DOWNLOAD : /^sdownload;$/i,
}

module.exports = messagesTypes

```

- Une fois que l'action à effectuer a été déterminée, un message au format JSON est envoyé au serveur.
- Le serveur analyse ce message, effectue les opérations nécessaires et renvoie une réponse au client, toujours au format JSON.
- Le client analyse la réponse et affiche une information dans la console.

## 2.6. Quelques illustrations

Dans l'application client,

- Les noms des groupes sont soulignés
- Les noms d'utilisateurs sont **en vert**

### 2.6.1. Illustration de quelques commandes (1)

```

E:\Informatique\fichiers web\Projet Web pour IoT\project-aurel-steve\td5\client>node client --name Aurel
You are connected to the server
>server> Welcome Aurel !
>Restauration de votre session...
.Aurel> Bonjour à tous!
.Alfred> Salut M. Aurel
.Aurel> Bonjour jeune homme
.reseau>Aurel> Bonjour le groupe!
.reseau>Sigala> Bonjour ici.
.Aurel> Bonjour Scammer!
.reseau>Aurel> Bonjour ldsjdj/dqkfh
.reseau>Aurel> Hi
.maison>Aurel> Bonjour dans le groupe!
.maison>Alfred> Salut à toi boss;
.reseau>Aurel> Bonjour dans le groupe réseau
.reseau>Aurel> ça va ici?
.server> You successfully removed Alfred from the group reseau because Pour rien
.server> Bertrand joined the group reseau
.reseau>Bertrand> Salut Je viens d'arriver
.reseau>Bertrand> Je m'appelle Bertrand
.server> Robert joined the group reseau
.server> Robert joined the group maison
Session restaurée.
>server> Alfred arrived.
>members;reseau
>server> Membres du groupe reseau (1): [ 'Aurel' ]
>invite;reseau;Alfred
>server> You successfully added Alfred to the group reseau
>reseau>Alfred> Bonjour et merci pour l'ajout.
>bg;reseau;Je t'en prie
>sdownload;
>Sauvegarde réussie dans le fichier Aurel-save2022-11-16 17-24-08.txt
>

```

La session de l'utilisateur est restaurée lorsqu'il se reconnecte

Illustration des commandes **members;** **invite;** et **bg;**

L'utilisateur télécharge sa session

Figure 1 : Illustration n°1

### 2.6.2. Illustration de quelques commandes (2)

```

E:\Informatique\fichiers web\Projet Web pour IoT\project-aurel-steve\td4\client>no
ient --name Robert
You are connected to the server
>server> Welcome Robert !
>groups;
>server> Liste des groupes (1): [ 'reseau' ]
>j;reseau
>server> You successfully joined the group reseau
>server> Aurel removed Francis from the group reseau because a resaon
>server> Aurel added Francis to the group reseau
>states;reseau
>reseau> Liste de tous les événements du groupe:
server> Francis created the group reseau
server> Aurel joined the group reseau
server> Bertrand joined the group reseau
reseau>Bertrand> Bonjour dans ce groupe.
reseau>Aurel> Bjr
reseau>Francis> Bertrand, je vais de retirer.
server> Francis removed Bertrand from the group reseau because some reason
server> Junior joined the group reseau
server> David joined the group reseau
reseau>David> Salut à tous.
server> You successfully joined the group reseau
server> Aurel removed Francis from the group reseau because a resaon
server> Aurel added Francis to the group reseau
reseau> Fin de la liste
>members;reseau
>server> Membres du groupe reseau: [ 'Aurel', 'Francis', 'Robert' ]
>s;Aurel;Bonjour Aurel. Comment tu vas?
>s;Bertrand;Hello Bertrand. How are you?
>

```

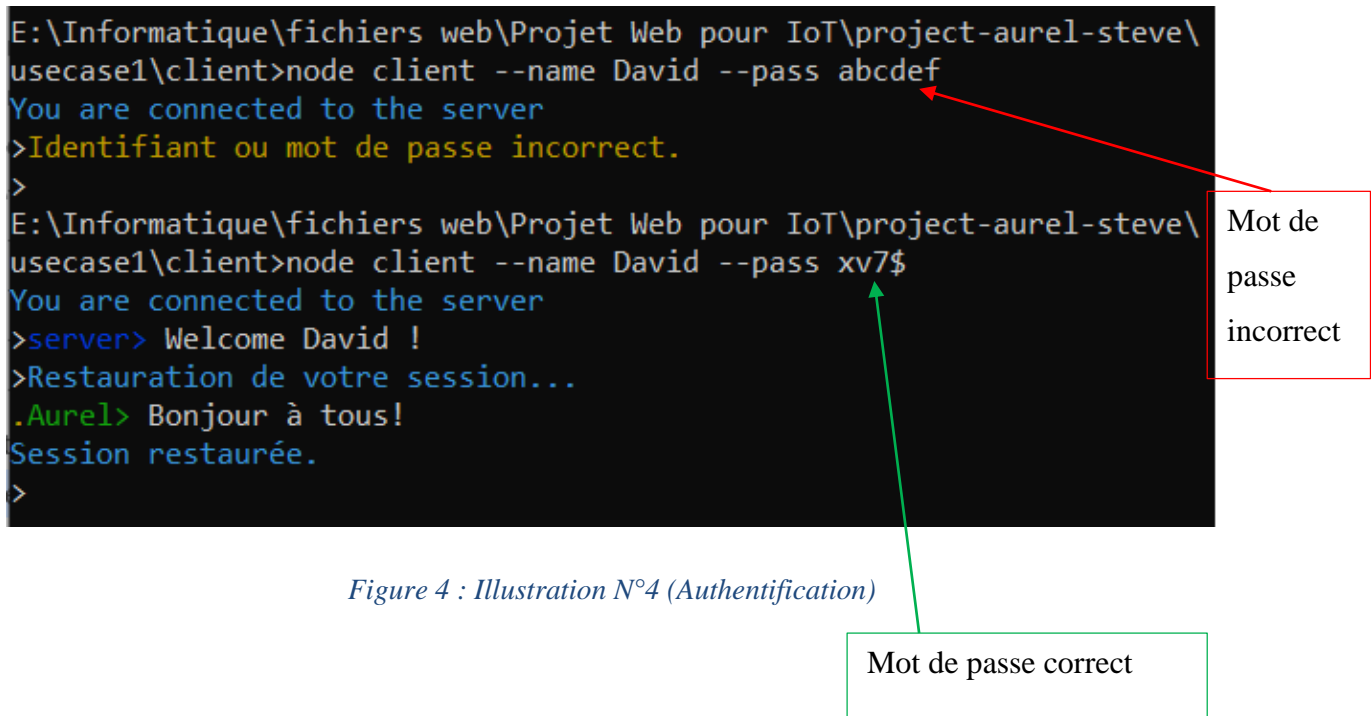
Figure 2 : Illustration N°2 (commandes groups; j; states; members; s;)

### 2.6.3. Illustration de quelques commandes (3)

```
>server> Welcome Aurel !
>server> Bertrand arrived.
>b;Bonjour à tout le monde
>server> Francis arrived.
>server> Francis created the group reseau
>j;reseau
>server> You successfully joined the group reseau
>server> Bertrand joined the group reseau
>reseau>Bertrand> Bonjour dans ce groupe.
>bg;reseau;Bjr
>reseau>Francis> Bertrand, je vais de retirer.
server> Francis removed Bertrand from the group reseau because some reason
>server> Junior arrived.
>server> Junior joined the group reseau
>server> Junior left.
>server> David arrived.
>server> David joined the group reseau
>reseau>David> Salut à tous.
>server> David left.
>server> Robert arrived.
>server> Robert joined the group reseau
>invite;reseau;Francis
>members;reseau
>server> Membres du groupe reseau: [ 'Aurel', 'Francis', 'Robert' ]
```

Figure 3 : Illustration N°3 (diverses notifications, commandes j; bg; invite; b; et members;)

#### 2.6.4. Illustration de l'authentification



The image shows a terminal window with a black background and white text. The terminal output is as follows:

```
E:\Informatique\fichiers web\Projet Web pour IoT\project-aurel-steve\
usecase1\client>node client --name David --pass abcdef
You are connected to the server
>Identifiant ou mot de passe incorrect.
>
E:\Informatique\fichiers web\Projet Web pour IoT\project-aurel-steve\
usecase1\client>node client --name David --pass xv7$
You are connected to the server
>server> Welcome David !
>Restauration de votre session...
.Aurel> Bonjour à tous!
Session restaurée.
>
```

Two annotations are present:

- A red arrow points from a red-bordered box labeled "Mot de passe incorrect" to the command `--pass abcdef`.
- A green arrow points from a green-bordered box labeled "Mot de passe correct" to the command `--pass xv7$`.

Figure 4 : Illustration N°4 (Authentification)

### 3. USE CASE 2 : Client Node JS / Serveur Node JS avec websocket

#### 3.1. Fonctionnalités implémentées

Dans ce use case, nous avons implémenté exactement les mêmes fonctionnalités que dans le premier.

#### 3.2. Fonctionnalités non implémentées

Idem que dans le premier use case.

#### 3.3. Définition du protocole (format des messages client/serveur)

Idem que dans le premier use case.

#### 3.4. Description de la base de données

Idem que dans le premier use case. Nous avons utilisé la même base de données.

#### 3.5. Explication du fonctionnement

Le principe de fonctionnement est le même que dans le premier use case. C'est juste le protocole de communication qui change car ici, on communique via des websockets, grâce à la bibliothèque socket.io. Cette dernière a deux variantes :

- Une variante pour serveur, nommée **socket.io**
- Une variante pour client, nommée **socket.io-client**

#### 4. USE CASE 3 : Client Node JS / Serveur ESP32 avec TCP/IP

Ici, le serveur est un module ESP32. Nous avons implémenté son fonctionnement en langage C++ **via l'IDE Arduino**. Pour ce qui est du client, son code reste le même que dans le premier use case.

Du fait de la complexité du C++ et la lenteur de compilation/téléversement du code dans l'ESP32, nous avons implémenté moins de fonctionnalités que dans les use case précédents.

##### 4.1. Fonctionnalités implémentées

1. Message de bienvenue du serveur
2. Envoi de messages privés
3. Envoi de messages broadcast
4. Lister les utilisateurs connectés
5. Quitter le chat (se déconnecter)
6. Notifications (avertir l'utilisateur quand une personne se connecte ou se déconnecte)
7. Créer un groupe
8. Rejoindre un groupe
9. Envoi de message dans un groupe
10. Lister les membres d'un groupe
11. Lister tous les groupes existants
12. Quitter un groupe
13. Ajouter quelqu'un dans un groupe
14. Retirer quelqu'un d'un groupe

#### 4.2. Définition du protocole (format des messages client/serveur)

Les formats des messages échangés restent les mêmes que dans les précédents use case.

#### 4.3. Explication du fonctionnement

Le principe de fonctionnement est le même que dans le premier use case. Les bibliothèques utilisées côté serveur (ESP32) sont :

- **Wifi.h** : pour la connexion au wifi et la création du serveur
- **ArduinoJson.h** : pour la manipulation d'objets JSON

### 5. USE CASE 4 : Client ESP32 / Serveur Node JS avec TCP/IP

Ici, le client est un module ESP32. Nous avons implémenté son fonctionnement en langage **C++ via l'IDE Arduino**. Pour ce qui est du serveur, son code reste le même que dans le premier use case.

#### 5.1. Fonctionnalités implémentées

1. Message de bienvenue du serveur
2. Envoi de messages privés
3. Envoi de messages broadcast
4. Lister les utilisateurs connectés
5. Quitter le chat (se déconnecter)
6. Notifications (avertir l'utilisateur quand une personne se connecte ou se déconnecte)

#### 5.2. Définition du protocole (format des messages client/serveur)

Les formats des messages échangés restent les mêmes que dans les précédents use case.

#### 5.3. Explication du fonctionnement

Le principe de fonctionnement est le même que dans le premier use case. Les bibliothèques utilisées côté client (ESP32) sont :



- **Wifi.h** : pour la connexion au wifi et la connexion au serveur
- **regex** : pour la manipulation des expressions régulières
- **ArduinoJson.h** : pour la manipulation d'objets JSON

L'interaction de l'ESP32 en tant que client du chat se fait via le moniteur série. Ce dernier dispose d'une zone de texte qui permet à l'utilisateur d'entrer ses commandes, et d'une console qui affiche les messages (voir figures suivantes).

#### 5.4. Quelques illustrations

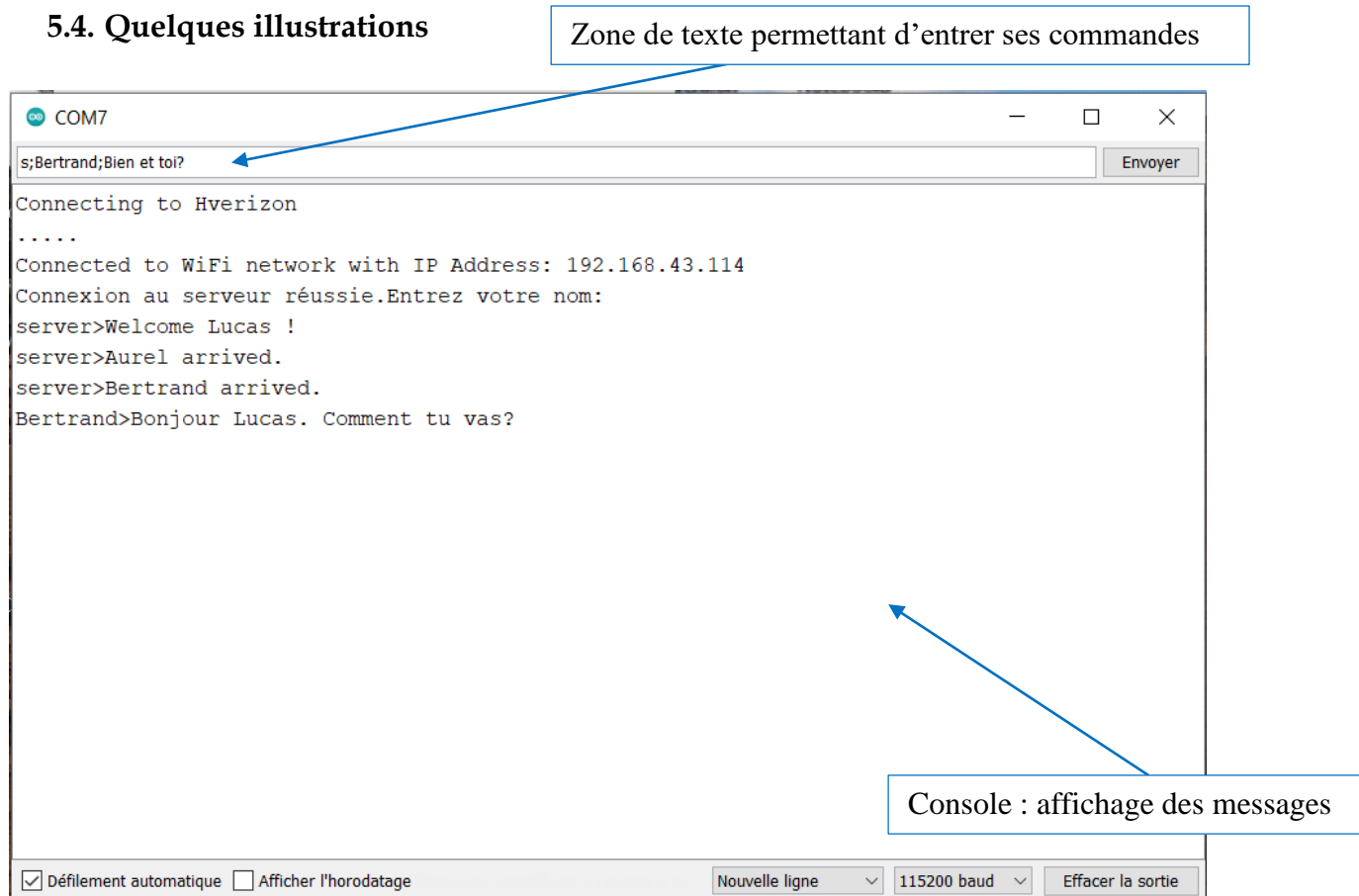


Figure 5 : Moniteur série du client ESP32

Ci-après, nous illustrons la communication entre deux clients :

- Un client ESP32, nommé Lucas
- Un client Node JS, nommé Aurel.



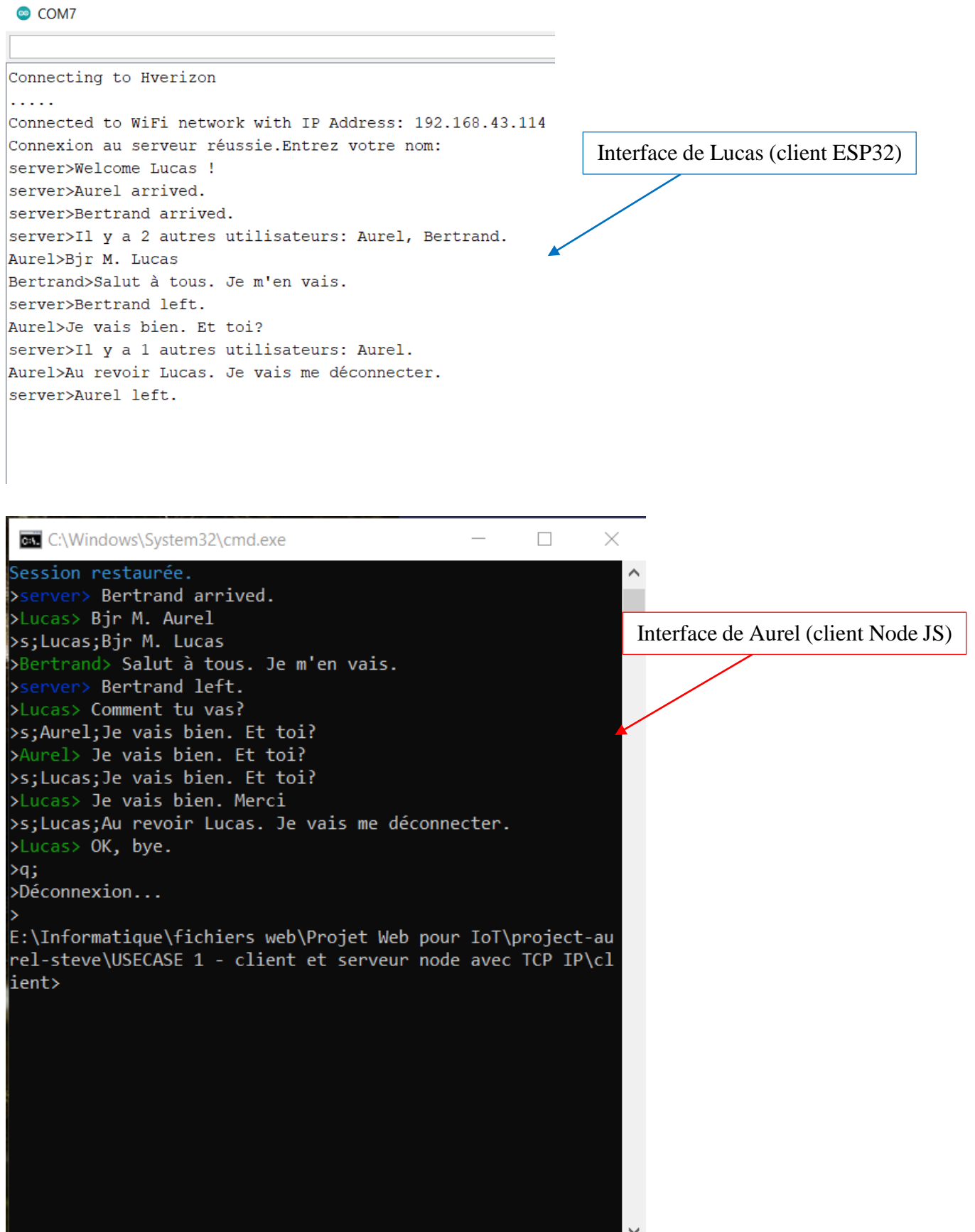


Figure 6 : Communication entre un client ESP32 et un client Node JS

## Conclusion

Parvenus au terme de ce projet qui portait sur la réalisation d'une application de chat incluant des pcs et des esp32, il en ressort que, certains protocoles sont plus adaptés que d'autres en fonction de l'équipement choisi, et qu'il faut être cohérent dans le choix du protocole de communication et du langage de programmation. Nous avons implémenté **04 use cases** qui sont : Client et Serveur node.js avec tcp/ip ; Client et Serveur node.js avec websocket ; Client node.js et Serveur ESP32 avec tcp/ip ; et enfin Client ESP32 et Serveur node.js avec tcp/ip. Ce projet nous a été très bénéfique puisqu'il nous a permis de comprendre le fonctionnement de la communication client-serveur et de découvrir le monde des microcontrôleurs et leur programmation.