

TP WebSockets

1 Explications

1.1 Socket

Ce sont des fonctions qui permettent à 2 programmes de communiquer à travers un réseau.

Écrite en C à l'origine, puis depuis, porté dans – presque - tout les langages.

La première API mettant en œuvre les socket a été développée par l'université de Berkeley pour leur BSD/Unix, dans les années 1980. C'est un des premiers truc open source de l'histoire.

Socket, veut dire « prise » en Français.

Un socket représente une prise par laquelle une application peut envoyer et recevoir des données. Cette prise permet à l'application de se brancher sur un réseau et communiquer avec d'autres applications qui y sont branchées. Les informations écrites sur une prise depuis une machine sont lues sur la prise d'une autre machine, et inversement.

1.2 WebSocket

Le protocole WebSocket permet d'ouvrir un canal de communication bidirectionnel (ou "full-duplex") sur un socket TCP pour les navigateurs et les serveurs web.

En clair, il permet :

- la notification au client d'un changement d'état du serveur,
- l'envoi de données en mode « pousser » (méthode Push) du serveur vers le client, sans que ce dernier ait à effectuer une requête.

Vous connaissez sûrement l'objet XMLHttpRequest utilisé via des requêtes HTTP avec un TTL long. Elles sont stockées par le serveur pour une réponse ultérieure au client via une fonction de callback.

C'est l'architectures Ajax.

WebSocket offre - pratiquement - le même service aux applications, mais présente l'intérêt de contourner les nombreux obstacles intermédiaires aux flux réseau (pare-feux etc.).

1.3 PyWebSocket

Renommé juste WebSockets depuis Python 3.

C'est l'implémentation Python que l'on vas utiliser.

2 Démo : Puissance 4

Ou « Connect 4» en anglais

Cette démo est en 3 partie.

- Partie 1 : On vas écrire un serveur et y connecter un navigateur.
 - Pour y jouer il faudra être sur le même navigateur
- Partie 2 : On vas connecter un second navigateur au serveur
 - Pour y jouer il faudra être sur le même réseau local
- Partie 3 : On vas mettre notre serveur sur le web.
 - Pour y jouer il faudra être sur la même planète (et connecté à Internet)



2.1 Partie 1 : En local

2.1.1 Démarrage propre

2.1.1.1 Créer un projet Git*

Créer un projet sur GitLab ; GitHub,

Y ajouter un `readme.md`

Le cloner en local sur votre PC

```
git clone git@git.arrobe.fr:Hugues/connect4.git
```

2.1.1.2 Installer Websockets

Créer un vEnv ; y installer websockets

Linux & Mac

```
cd connect4
```

```
python -m venv connect4
```

```
source connect4/bin/activate
```

```
(connect4) $ pip install websockets
```

Windows

```
cd connect4
```

```
python -m venv connect4
```

```
connect4/Scripts/activate
```

```
(connect4) $ pip install websockets
```

2.1.1.3 Vérifier les versions

```
python -v
```

Attendu ≥ 3.9

```
python -m websockets --version
```

Attendu = 10.4

2.1.1.4 Un .gitignore

Créez un fichier .gitignore à la racine et mettez

```
connect4/*  
__pycache__/*
```

Ça évite de monter sur le serveur Git des dossiers inutile

... Et on pousse le 1^{er} commit

2.1.2 Le kit de démarrage coté client

Récupérez les 4 fichiers suivants et mettez les à la racine du projet

(comme c'est pas du Python, je les ai commenté et vous fournis le code)

2.1.2.1 index.html

Le cadre HTML du client

2.1.2.2 main.js

Appelé par index.html

Déclenche la fonction `createBoard()` quand le DOM est chargé

2.1.2.3 connect4.js

Contient la fonction `createBoard()` qui remplit de `div class="board"` de index.html avec le tableau de jeu.

Injecte aussi `connect4.css` dans le DOM

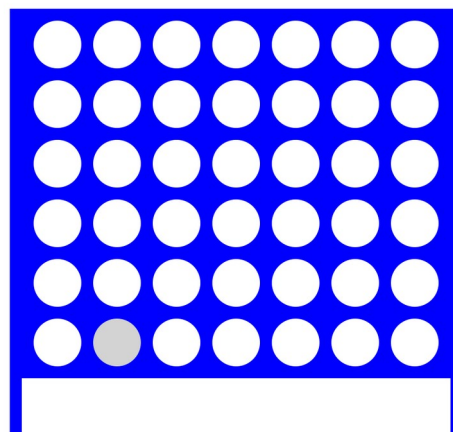
2.1.2.4 connect4.css

Un coup de peinture

Pour tester : Lancez un serveur http et allez à l'url

<http://127.0.0.1:8000>

Si ça affiche ça : C'est OK



2.1.3 Le kit de démarrage coté serveur

1 seul fichier : `app.py` (mettez le à la racine du projet)

2.1.3.1 `app.py`

Fichier principal du serveur

La fonction `main` qui crée un boucle d'écoute infinie et ouvre un websocket d'écoute grâce à la méthode `serve`. Ses 3 paramètres sont :

- `handler` est une coroutine (une fonction asynchrone) qui gère une connexion.
Quand un client se connecte, `websockets` appelle `handler` avec la connexion en paramètre.
Lorsque `handler` se termine, `websockets` ferme la connexion.
- Le deuxième paramètre définit les interfaces réseau auxquelles le serveur peut être joint. Avec une chaîne vide : Le serveur écoute toutes les interfaces.
- Le troisième paramètre est le port d'écoute du serveur

Pour tester :

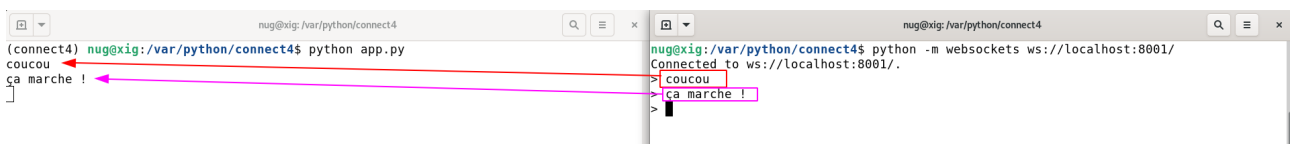
Démarrez le serveur dans le terminal :

```
python app.py
```

Puis ouvrez un 2nd terminal, et lancez un client en ligne de commande avec :

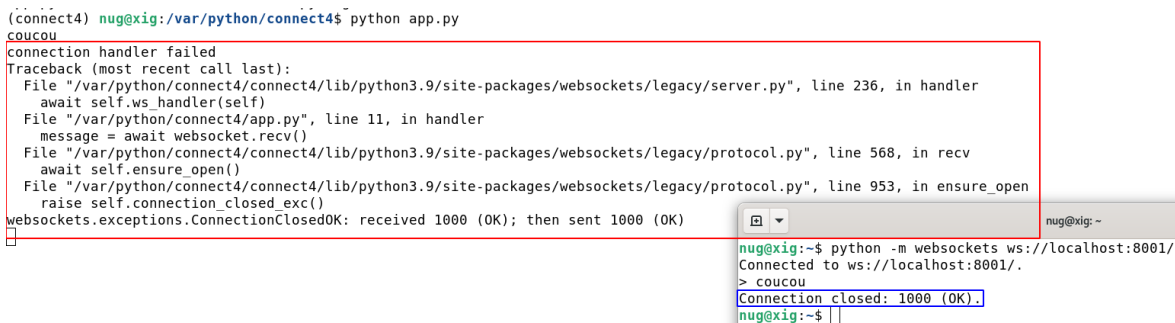
```
python -m websockets ws://localhost:8001/
```

Tout ce que vous tapez dans la 2eme console apparaît dans la 1er



Ça marche !

Par contre, quand le client quitte (Ctrl + C) ; le serveur morfle !

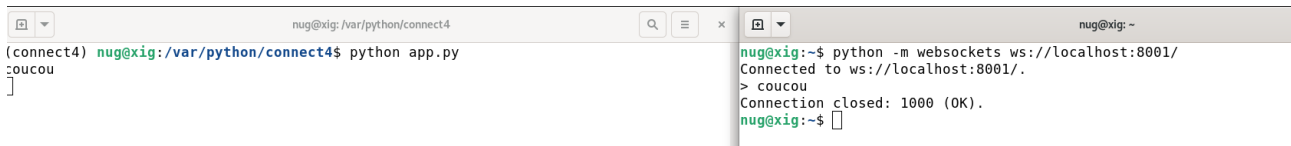


C'est parce que `recv()` c'est déconnecté brutalement que le serveur a levé une exception `ConnectionClosedOK`.

On corrige `app.py` pour éviter de polluer la log avec des « faux » messages d'erreur

```
async def handler(websocket):
    while True:
        try:
            message = await websocket.recv()
        except websockets.ConnectionClosedOK:
            break
        print(message)
```

Re-test ...



Ok, c'est bon.

2.1.4 Faire causer le navigateur avec le serveur

Comme le client (JavaScript) et le serveur (Python) utilisent 2 langages différents, il faut se mettre d'accord sur un format de messages. On va utiliser le plus classique : JSON

Les 3 commandes de base en JS :

Ouvrir une connexion WebSocket :

```
const websocket = new WebSocket("ws://localhost:8001/");
```

Créer un message JSON (ici, je joue sur la colonne 3)

```
const event = {type: "play", column: 3};
```

Envoyer ce message au serveur

```
websocket.send(JSON.stringify(event));
```

Concrètement :

Dans `main.js`, ajouter la fonction `sendMoves` :

```
function sendMoves(board, websocket) {
    // Fonction qui envoie le clic au serveur
    board.addEventListener("click", ({ target }) => {
        // Je récupère la colonne cliquée
        const column = target.dataset.column;
        if (column === undefined) {
            // J'ignore les clics hors des colonnes
            return;
        }
        const event = {
```

```

    type: "play",
    // je convertit le N° de colonne en un interger en base 10
    column: parseInt(column, 10),
  };
  // J'envoie le message "play" au serveur après l'avoir linéarisé
  websocket.send(JSON.stringify(event));
});
}

```

Dans `main.js`, ajouter un second événement au `addEventListener` :

```

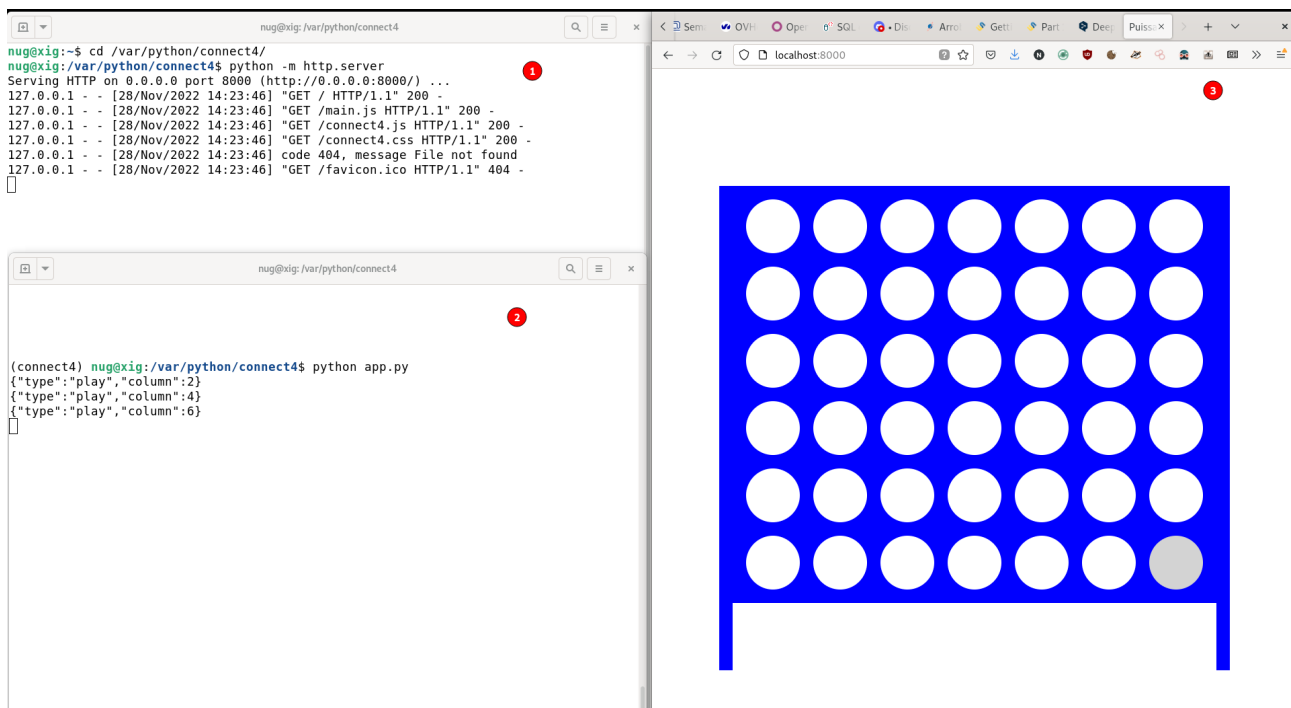
// Ouvrir la connexion WebSocket & enregistrer l'EventListener.
const websocket = new WebSocket("ws://localhost:8001/");
sendMoves(board, websocket);

```

Pour tester :

- **1** Dans un nouveau terminal : Démarrez un serveur http python
- **2** Démarrer le serveur applicatif
- **3** Dans votre navigateur, lancez le client ... et cliquez

Si vous voyez les messages s'afficher dans **2** ... c'est gagné



2.1.5 Faire causer le serveur avec le navigateur

Pour l'instant, on clique ... mais rien ne se passe ! (en tout cas coté navigateur)

... c'est parce-que le serveur ne réponds pas.

Au boulot !

En JavaScript, on reçoit les messages WebSocket en faisant

```
websocket.addEventListener("message", ({ data }) => {  
    const event = JSON.parse(data);  
    // Faire quelque chose  
});
```

Le serveur vas envoyer 3 types de messages :

- `{type: "play", player: "red", column: 3, row: 0}`
- `{type: "win", player: "red"}`
- `{type: "error", message: "Cette colonne est complète."}`

Ajoutons 2 fonctions à `main.js` :

```
function showMessage(message) {  
    window.setTimeout(() => window.alert(message), 50);  
}  
  
function receiveMoves(board, websocket) {  
    websocket.addEventListener("message", ({ data }) => {  
        const event = JSON.parse(data);  
        switch (event.type) {  
            case "play":  
                // Afficher le coup joué avec playMove de connect4.js.  
                playMove(board, event.player, event.column, event.row);  
                break;  
            case "win":  
                showMessage(`Joueur ${event.player} à gagné !`);  
                // Game over ! On ferme la connexion.  
                websocket.close(1000);  
                break;  
            case "error":  
                // Afficher le message envoyé par le serveur  
                showMessage(event.message);  
                break;  
            default:  
                throw new Error(`Unsupported event type: ${event.type}.`);  
        }  
    });  
}
```

Ainsi qu'un troisième événement au `addEventListener` :

```
    receiveMoves(board, websocket);  
(juste avant le sendMoves )
```

Explications :

La fonction `showMessage` qui sert à afficher un message à l'utilisateur utilise une grosse ruse à 2 balles.

Pour éviter, par exemple, dans le cas d'une victoire que le message « Rouge à gagné » ne s'affiche avant que le pion soit visible ... j'ai mis un délai de 50 millisecondes

... oui, je sait, c'est mal

Dans du « vrai » code, on ferai autrement (déjà, on n'utiliserai pas `window.alert` ...). Mais pour un TP c'est bien suffisant

La fonction `receiveMoves` n'est pas compliquée. Elle se contente de trier les messages reçus et de les faire suivre à `showMessage` pour affichage ou `playMove` pour « voir » le coup joué.

Il faut encore faire quelques évolutions coté serveur :

2.1.5.1 connect4.py

C'est ce fichier qui vas contenir la logique de jeu ... plus tard

Pour l'instant, créez un fichier `connect4.py` à la racine et mettez

```
# Défini 2 constantes
PLAYER1, PLAYER2 = "red", "yellow"
```

2.1.5.2 app.py

Ajoutez 2 imports :

```
import json
from connect4 import PLAYER1, PLAYER2
```

Réécrire la fonction `handler` comme suit :

```
async def handler(websocket):
    redTurn = True
    while True:

        # Lecture du message envoyé par le navigateur
        async for message in websocket:

            # A chaque tour on change de joueur
            player = PLAYER1 if redTurn else PLAYER2
            redTurn = not redTurn

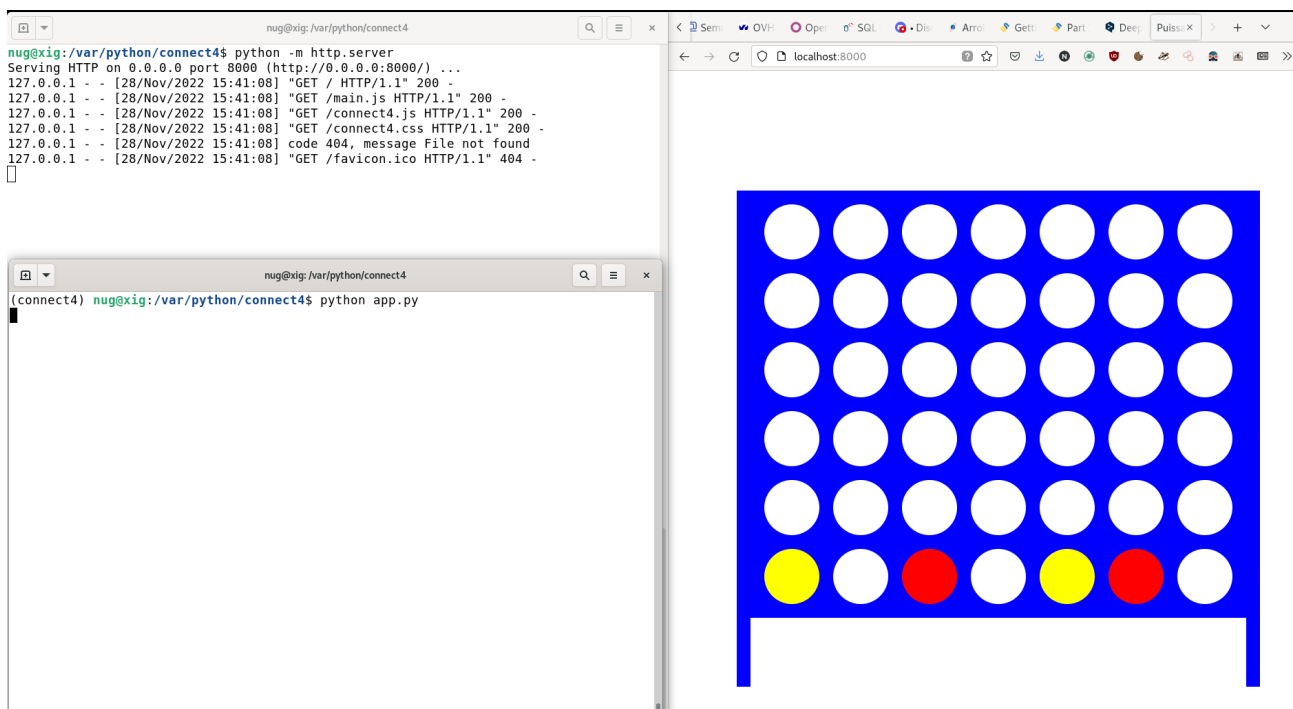
            event = json.loads(message)
            assert event["type"] == "play"
            column = event["column"]
```



```
# On forge un objet JSON event
event = {
    "type": "play",
    "player": player,
    "column": column,
    "row": 0, # Pour l'instant on ne sait jouer que sur la ligne du bas
}
# On l'envoi au navigateur pour affichage
await websocket.send(json.dumps(event))
```

Pour tester, relancez les 2 serveurs + refresh de la page du navigateur

⚠ Pour l'instant il n'y a pas de règles du jeu et on fait nawak (et que sur la ligne du bas)



2.1.6 Le jeu avec des règles

Remplacer le fichier `connect4.py` par celui fourni

Et réécrire une dernière fois la fonction `handler` :

```
async def handler(websocket):
    game = Connect4() # On instancie la classe Connect4
    redTurn = True # C'est le rouge qui commence

    # Lecture du message envoyé par le navigateur
    async for message in websocket:

        # A chaque tour on change de joueur
        player = PLAYER1 if redTurn else PLAYER2
        redTurn = not redTurn

        # On lit un evenement "play" depuis le navigateur
        event = json.loads(message)
```

```

assert event["type"] == "play"
column = event["column"]

try:
    # On joue le coup reçu
    row = game.play(player, column)
except RuntimeError as exc:
    # On a reçu un événement "error" pour coup illégal
    event = {
        "type": "error",
        "message": str(exc),
    }
    await websocket.send(json.dumps(event))
    continue

# On forge un événement "play" pour afficher sur le navigateur
event = {
    "type": "play",
    "player": player,
    "column": column,
    "row": row,
}
await websocket.send(json.dumps(event))

# Si ce dernier coup est gagnant : On envoie un événement "win"
if game.winner is not None:
    event = {
        "type": "win",
        "player": game.winner,
    }
    await websocket.send(json.dumps(event))

```

Et ça marche ! 🎉

On a un jeu fonctionnel ... mais sur un même navigateur 😞

2.2 Partie 2 : En réseau local

2.2.1 Principe

Dans cette version, le serveur va devoir communiquer avec 2 navigateurs différents.

Il va devoir en identifier un comme « Joueur 1 » (Le 1^{er} arrivé)

Puis l'appairer avec le prochain qui arrive (et qui sera « Joueur 2 »)

Pour ça on va gérer un 4ème événement : « init »

Lorsque le premier joueur va déclencher « init » : On lui donnera un identifiant.

Ensuite, on communiquera l'identifiant au deuxième joueur.

Lorsque le second joueur rejoint le jeu (« init »), on pourra les appairer à l'aide de l'identifiant.

2.2.2 Premier essai

Assez causé. Codons !

➡ Dans `index.html` : on ajoute un lien pour rejoindre une partie

```
<div class="actions">
  <a class="action join" href="">Join</a>
</div>
```

➡ Dans `main.js` : On ajoute une fonction `initGame` qui enverra le nouvel événement « init »

```
function initGame(websocket) {
  websocket.addEventListener("open", () => {
    // Envoie un événement "init" au 1er joueur
    const event = { type: "init" };
    websocket.send(JSON.stringify(event));
  });
}
```

➡ Dans `main.js` : On ajoute l'appel à `initGame` dans le `window.addEventListener`
`initGame(websocket);`

➡ Dans `main.js` : On ajoute un case au switch (`event.type`) :

```
case "init":
  // Créer un lien pour inviter le second joueur
  document.querySelector(".join").href = "?join=" + event.join;
  break;
```

➡ Dans `app.py` : On ajoute un import : (pour générer des chaînes aléatoires)

```
import secrets
```

➡ Dans `app.py` : On ajoute une variable globale (pour stocker un tableau de parties) :

```
JOIN = {}
```

➡ Dans `app.py` : On ajoute une fonction `start`

```
async def start(websocket):
    game = Connect4()
    connected = {websocket}

    # On génère une clé de 12 chars de long, compatible avec une URL
    join_key = secrets.token_urlsafe(12)
    # On stocke le tuple Partie / Connexion dans la case "clé"
    JOIN[join_key] = game, connected

    try:
        # Envoyez la clé d'accès au navigateur du premier joueur,
        # ou elle sera utilisé pour faire un lien "Join"
        event = {
            "type": "init",
            "join": join_key,
```

```

    }
    await websocket.send(json.dumps(event))

    # Temporaire pour tests
    print("Premier joueur commence partie ", id(game))
    async for message in websocket:
        print("Premier joueur envoie ", message)

    finally:
        del JOIN[join_key]

```

➡ Dans `app.py` : on réécrit (encore 😊) `handler` :

```

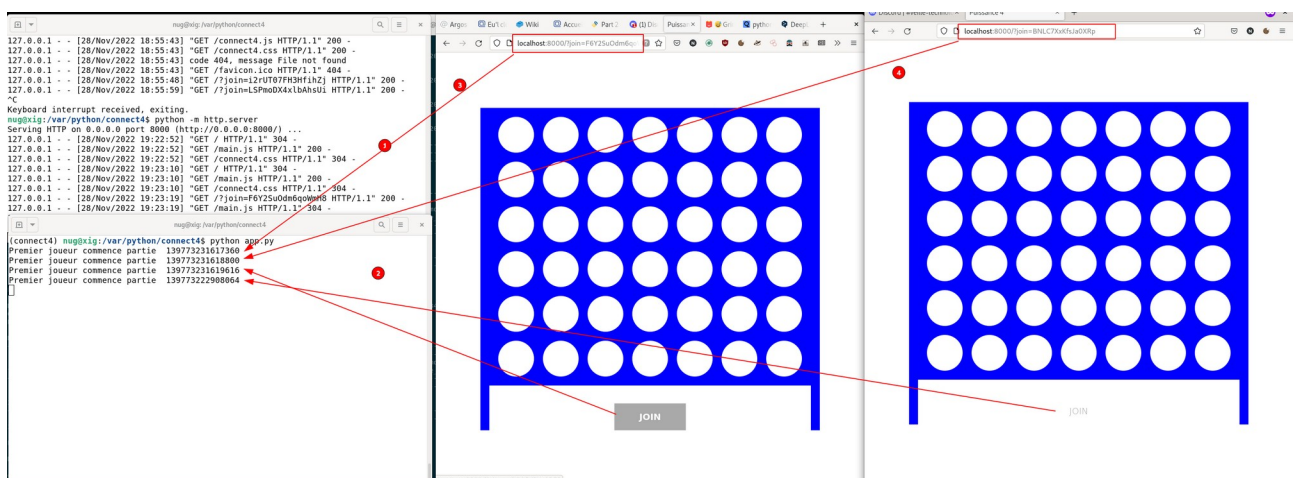
async def handler(websocket):
    # Lire & traiter les événements "init" depuis le navigateur
    message = await websocket.recv()
    event = json.loads(message)
    assert event["type"] == "init"

    # Le premier joueur commence une partie
    await start(websocket)

```

Et on teste

J'ai ouvert 2 navigateurs différents (le 2eme est en session privée)



On voit que les 2 navigateurs ont une URL différente : ✓

Par contre les 4 messages ... ont tous des clés différentes : ✗

2.2.3 Joindre une partie

➡ Dans `main.js` : Améliorer la fonction `initGame` pour tenir compte du contexte :

```

function initGame(websocket) {
    websocket.addEventListener("open", () => {
        // Envoie un événement "init" en fonction de la personne qui se
        connecte.

        const params = new URLSearchParams(window.location.search);
        let event = { type: "init" };
        if (params.has("join")) {

```

```

        // Un second joueur rejoint une partie existante
        event.join = params.get("join");
    } else {
        // Le 1er joueur crée la partie
    }
    websocket.send(JSON.stringify(event));
});
}

```

➡ Dans `app.py` : Ajouter une fonction `error` (qui devient obligatoire vu le bordel le nombre d'événements à gérer) :

```

async def error(websocket, message):
    event = {
        "type": "error",
        "message": message,
    }
    await websocket.send(json.dumps(event))

```

➡ Dans `app.py` : Ajouter une fonction `join` (qui sera utilisé par le 2eme joueur pour se connecter)

```

async def join(websocket, join_key):
    # Trouver le "bon" Connect4
    try:
        game, connected = JOIN[join_key]
    except KeyError:
        await error(websocket, "Partie non trouvée.")
        return

    # Se connecter à la partie
    connected.add(websocket)
    try:

        # Temporaire pour tests
        print("Second joueur à rejoint ", id(game))
        async for message in websocket:
            print("Second joueur envoie ", message)

    finally:
        connected.remove(websocket)

```

➡ Dans `app.py` : Et, pour respecter la tradition, on modifie `handler` :

```

# Le premier joueur commence une partie
await start(websocket)

```

devient

```

if "join" in event:
    # Le second joueur rejoint une partie existante
    await join(websocket, event["join"])
else:
    # Le premier joueur commence une partie

```

```
await start(websocket)
```

Protocole de test :

- Ouvrir un 1^{er} navigateur
- Copier le lien « Join »
- Ouvrir un 2eme navigateur en session privée
- Coller

```
(connect4) nug@xig:/var/python/connect4$ python app.py
Premier joueur commence partie 139917976716624
Second joueur à rejoint 139917976716624
]
```

YES !

2.2.4 Joindre une partie ... et pouvoir jouer

On y est presque ...

➡ Dans `app.py` : Dans `start` ;

remplacer le bloc

```
# Temporaire pour tests
par
```

```
await play(websocket, game, PLAYER1, connected)
```

➡ Dans `app.py` : Dans `join` ;

remplacer le bloc

```
# Temporaire pour tests
par
```

```
await play(websocket, game, PLAYER2, connected)
```

➡ Dans `app.py` : Ajouter une fonction `play` qui contient la logique de jeu :

```
async def play(websocket, game, player, connected):
    """
    Recoit et traite les coups des différents joueurs
    """
    async for message in websocket:
        # Lit un événement "play" reçu d'un navigateur
        event = json.loads(message)
        assert event["type"] == "play"
        column = event["column"]
    try:
        # Joue le coup demandé
        row = game.play(player, column)
    except RuntimeError as exc:
        # Envoie une erreur si le coup est irrégulier
        await error(websocket, str(exc))
    continue
```

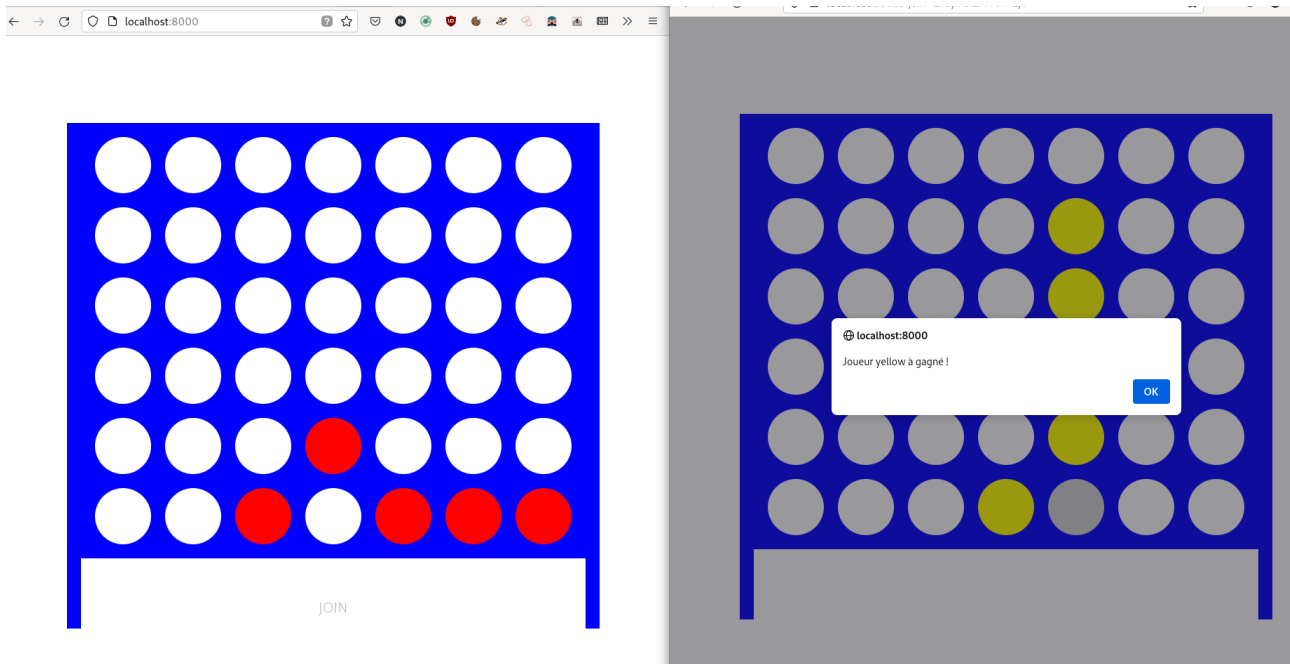
```

# Forge un événement "play" pour mettre à jour le navigateur
event = {
    "type": "play",
    "player": player,
    "column": column,
    "row": row,
}
await websocket.send(json.dumps(event))

# Si le coup est gagnant : Envoie un événement "win"
if game.winner is not None:
    event = {
        "type": "win",
        "player": game.winner,
    }
    await websocket.send(json.dumps(event))

```

On teste (avec le coup du copié / collé de l'URL)



Mwouais ... c'est pas tip top

2.2.5 Joindre une partie ... et pouvoir jouer correctement

Le problème est que les messages « mettre un pion » et « gagné » ne sont envoyés que au navigateur que les à déclenchés.

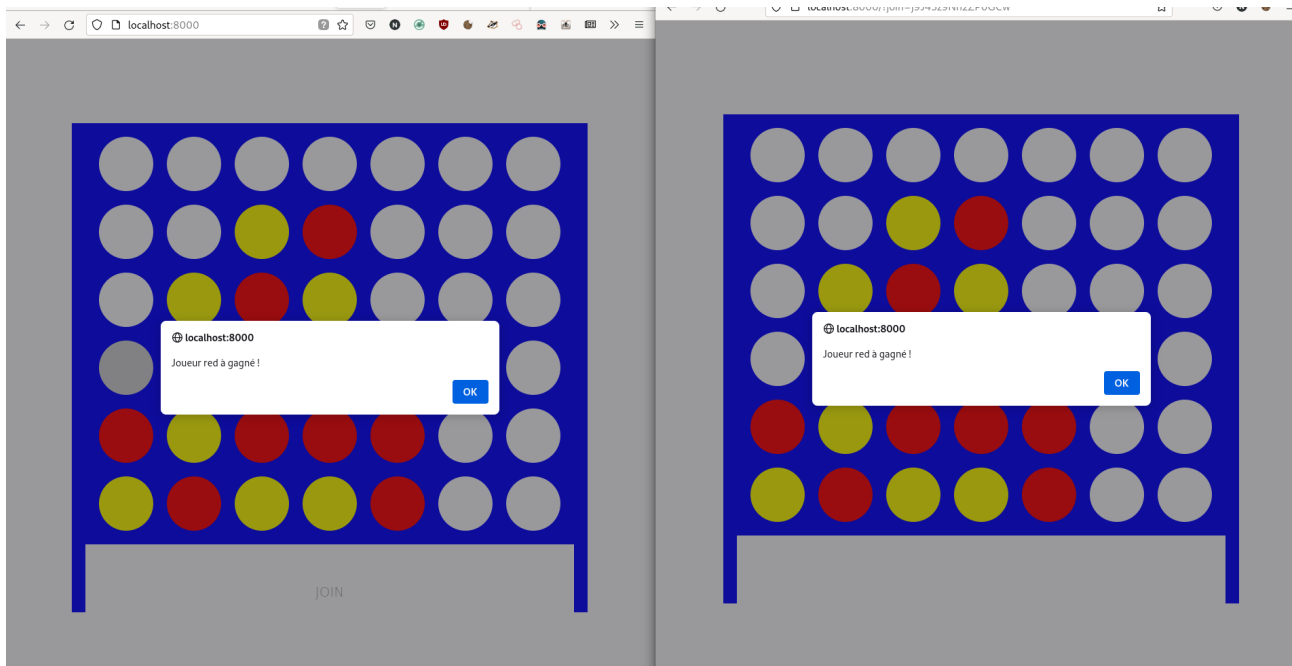
Il suffit d'utiliser la fonction websocket Broadcast pour diffuser le message à tout les connectés !

On va remplacer les 2 `await websocket.send(json.dumps(event))` de play
par des `websockets.broadcast(connected, json.dumps(event))`

Ou `connected` vaut la liste des connectés

i pas besoin de `await` car c'est une fonction synchrone

Re – test ...



Étape 2 : ✓

2.3 Exercice

Ajouter, un bouton « Watch » qui permet de générer une URL pour des spectateurs

TODO list :

- ➡ Déclarer une variable globale WATCH similaire à JOIN
- ➡ Générer une clé « watch » lors de la création d'une partie ; elle doit être différente de la clé join, sinon un spectateur pourrait détourner une partie en modifiant l'url
- ➡ Inclure la clé « watch » dans l'événement « init » envoyé au premier joueur
- ➡ Générer un lien WATCH dans l'interface utilisateur avec un paramètre de requête watch
- ➡ Mettre à jour la fonction `initGame()` pour gérer de tels liens
- ➡ Mettre à jour la fonction `handler()` pour appeler la fonction `watch()` pour les spectateurs
- ➡ Empêcher `sendMoves()` d'envoyer des événements « play » pour les spectateurs

Facultatif

- ➡ Afficher un pion dans le coin supérieur gauche du navigateur pour rappeler le statut (Joueur 1 / pion rouge | Joueur 2 / pion jaune | Spectateur / pion gris)

2.4 Partie 3 : Sur Internet

2.4.1 Prérequis

Pour faire cette partie, il faut disposer d'un serveur web avec un nom de domaine / sous domaine. On pourrait passer par des hébergeurs gratuit style Heroku ou une autre plateforme IaaS ... mais nan : Chui vieux et j' préfère quand je gère tout moi même !

2.4.2 Adaptation du code

Avant d'attaquer la partie serveur, on vas adapter un minimum le code

Créer un fichier `requirement.txt` qui vas contenir la loooongue liste des spécificités de notre projet :

`websockets`

Dans `main.js` on a une ligne ... qui vas causer problème

```
// Ouvrir la connexion WebSocket & enregistrer les événements
const websocket = new WebSocket("ws://localhost:8001/");
```

Ajoutez une fonction

```
function getWebSocketServer() {
  if (window.location.host === "connect4.arrobe.fr") {
    return "wss://connect4.arrobe.fr/ws";
  } else if (window.location.host === "localhost:8000") {
    return "ws://localhost:8001/";
  } else {
    throw new Error(`Hote inconnu: ${window.location.host}`);
  }
}
```

Et modifiez la ligne « WebSocket »

```
const websocket = new WebSocket(getWebSocketServer());
```

2.4.3 Coté hébergeur :

J'ai crée un sous domaine que je fais pointer vers l'IP d'un serveur « à moi » :

Ajouter une entrée à la zone DNS Étape 3 sur 3

Vous allez ajouter l'entrée suivante dans votre zone DNS :

Type de champ	A
Domaine	connect4.arrobe.fr.
Cible	46.105.103.40

L'ajout sera immédiat dans la zone DNS, mais veuillez prendre en compte le temps de propagation (maximum 24h).

AnnulerPrécédentValider

2.4.4 Coté serveur :

⚠ Si votre serveur n'a pas un python 3.9 il faut l'installer.

Sous Debian, il faut le compiler depuis les sources

Tuto : <https://computingforgeeks.com/how-to-install-python-on-debian-linux/>

Remplacez 3.10.0 par la dernière release de la branche : 3.9.9

```
# Se connecter en ssh au serveur
ssh hugues@arwen.arrobe.fr

# Aller dans le dossier des sites web
cd /var/www

# Cloner le projet
git clone git@gitlab.com:arrobe/connect4.git connect4.arrobe.fr

# Aller dans le dossier
cd connect4.arrobe.fr

# Créer un venv
python3.9 -m venv connect4

# Activer
source connect4/bin/activate

# Installer les prérequis
pip install -r requirements.txt

# Créer fichier de conf. Apache
nano /etc/apache2/sites-available/connect4.arrobe.fr.conf

<VirtualHost *:80>
    ServerAdmin webmaster@arrobe.fr
    ServerName connect4.arrobe.fr

    DocumentRoot /var/www/connect4.arrobe.fr

    <Directory /var/www/connect4.arrobe.fr>
        Require all granted
        AllowOverride All
    </Directory>

    ErrorLog /var/log/apache2/connect4.arrobe.fr.error.log
    LogLevel warn
    CustomLog /var/log/apache2/connect4.arrobe.fr.access.log combined
</VirtualHost>
```

```
# Activer le site
a2ensite connect4.arrobe.fr.error.log
# Tester la config
apachectl configtest
# Recharger apache
systemctl reload apache2
# Activer le https
certbot
# Modifier le fichier apache / https pour faire un reverse proxy
nano /etc/apache2/sites-available/connect4.arrobe.fr-le-ssl.conf

et ajouter

    ProxyRequests Off
    ProxyPreserveHost On
    ProxyPass /ws ws://localhost:8001
    ProxyPassReverse /ws ws://localhost:8001

# Lancer le serveur Python
python app.py
```

Enjoy !

3 Le TP (enfin!)

3.1 Cahier des charges

En utilisant les websockets entre Python (serveur) et JavaScript (client) vous devez réaliser un jeu en ligne multijoueur.

... exactement ce qu'on vient de faire ensemble, quoi !

Mais :

- Binômes autorisés
- Jeu en temps réel (et non tour par tour)

Pour passer du tour par tour au temps réel :

Pour le temps réel il suffit, coté client d'envoyer des messages websockets tous les 1/10^{eme}¹ de secondes (position de la raquette, de la balle, du joueur, ...)

¹ Ou plus rapide encore, si vous voyez que ça reste fluide

Le serveur est déjà en temps réel

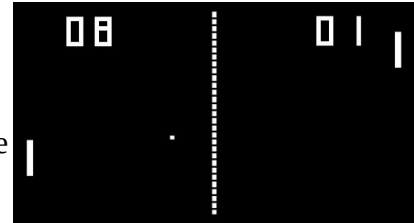
3.2 Des idées de jeu :

🔊 Ce ne sont que des suggestions : Codez votre propre jeu !

3.2.1 Pong

Avec comme améliorations possibles

- Pouvoir jouer au clavier, à la souris ou au doigt sur un téléphone
- Des bonus / malus qui popent au milieu du terrain (à prendre en passant la balle dessus) et qui augmentent la taille de ta raquette, rapetissent celle de l'adversaire ...
- Pouvoir déplacer sa raquette en 2 dimensions pour « monter au filet » (et accélérer la balle)
- Pouvoir jouer à 4 joueurs (Un sur chaque côté du carré)



3.2.2 Tétris

En multijoueur : Toutes les 2 lignes réussies, vous ajoutez une ligne (par le bas) à vos adversaires.

(ligne qui comporte 8 cases sur 10)

Le dernier survivant à gagné

Avec comme améliorations possibles :

- 3 joueurs, 4 joueurs, ...



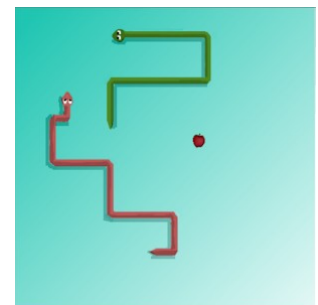
3.2.3 Snake

Celui qui mange la pomme fait grandir son serpent

Tout choc à la tête tue le serpent

Avec comme améliorations possibles :

- 3 joueurs, 4 joueurs, ... paper.io



3.2.4 Blobby volley

Chaque joueur peut se déplacer gauche-droite et peut sauter

Il ne renvoie la balle que si il la touche pendant le saut

Avec comme améliorations possibles

- Multijoueur (1 vs 2 ; 2 vs 2 ; 3 vs 2 ...)

3.2.5 Ton jeu !

Tant que c'est multijoueur en ligne en temps réel



avec client en JS et serveur en Python

... c'est OK

3.3 Contraintes

Pour la partie HTML/CSS :

La partie esthétique du jeu n'est pas notée

Pas de panique si vous êtes une quiche en CSS : Vous pouvez vous contenter de bouger des carrés

Pour la partie JS :

Même remarque : La partie esthétique du jeu n'est pas notée

N'utilisez pas de librairies « de jeu » style Phaser.io

Autant ça vous facilitera la partie JS, autant ça vas vous pourrir la partie websockets

Jquery autorisé

C'est tout !

3.4 Serveur Web

Si vous n'avez pas de serveur web / si vous ne voulez pas l'utiliser : Je vous héberge

Je met à votre disposition un serveur web et un sous-domaine

Attention : C'est un serveur sur lequel il y a des « vrai » sites de prod qui tournent

Je serai très moyennement compréhensif si vous me plantez mon serveur exprès !

A la demande :

- Je crée le sous domaine `xxx.arrobe.fr`
- Je crée un user ssh xxx chrooté dans un dossier `/home/xxx`
- Je crée le fichier de conf apache qui pointe sur `/home/xxx/TP`

3.5 Barème

Qualité du code : 10 points

Optimisé, commenté, Facile à lire & à comprendre ...

ou

Codé avec le cul, contient du code mort, des variables nommés à la Yolo ...

Richesse fonctionnelle : 10 points

- Le jeu de base fonctionne ?
- Le multijoueur fonctionne ?
- C'est fluide ?
- Y'a des options ? Des trucs en plus ?

3.6 Livrables

Un dépôt git + une démo en ligne

Attention : Assurez vous que tout les membres du binôme ont poussé du code sur le dépôt

Si l'un des 2 à commité 95% du code l'autre aura une note ... proportionnelle 😊

Toutes les explications / consignes d'installation ... doivent être dans le README.md à la racine du projet

Date de rendu :

Avant le dimanche 12 février 2023

A rendre par [mail](#) ou par MP Discord (git + démo)

N'oubliez pas de me rappeler vos noms & prénoms ; les pseudos passent mal sur le bulletin de note officiel.