Práctica 2. Servidor de STOP!

Diego Esclarín Fernández

May 9, 2024

Contenido

1 Introducción

La práctica de implementación del Servidor de STOP tiene como objetivo desarrollar una aplicación cliente/servidor en Python que permita a múltiples jugadores participar en el juego del STOP de manera concurrente. En este juego, los jugadores comparten un tablero formado por diversas categorías, donde cada jugador debe completar las categorías con palabras que comiencen con una letra aleatoria generada por el servidor. La implementación de esta práctica implica aspectos técnicos como la comunicación mediante sockets, la gestión de la concurrencia y sincronización entre los jugadores y el servidor, entre otros.

2 Descripción del Juego del STOP

El juego del STOP es una versión del clásico juego "Stop" donde un número determinado de jugadores comparten un tablero con diferentes categorías. Cada categoría representa un aspecto diferente, Marca, Lugar, Comida o Animal. Al inicio de la partida, el servidor genera una letra al azar y la envía a los jugadores. Luego, los jugadores deben completar cada categoría del tablero con palabras que comiencen con la letra generada. Es importante que la palabra escrita por los jugadores comience por mayúscula. Por ejemplo, si la letra es "A", los jugadores deberán encontrar una marca que comience con "A", una comida que comience con "A", y así sucesivamente.

El tablero es un recurso compartido entre los jugadores y el servidor, donde cada jugador puede ver las palabras escritas por los demás. El objetivo del juego es completar el máximo de categorías del tablero con palabras válidas antes de que termine el tiempo establecido o se complete el tablero.

3 Aspectos Técnicos

Para implementar el servidor del juego del STOP, se deben tener en cuenta varios aspectos técnicos:

1. **Comunicación mediante sockets**: El servidor y los clientes se comunicarán a través de sockets para enviar y recibir datos de manera bidireccional. Se utilizará la biblioteca 'socket' de Python para gestionar la comunicación.

2. **Generación de letras aleatorias**: El servidor generará una letra aleatoria al inicio de cada partida y la enviará a todos los jugadores. Se puede utilizar la biblioteca 'random' de Python para generar letras aleatorias.

3. **Gestión de múltiples partidas simultáneas**: El servidor debe ser capaz de atender varias partidas simultáneamente, permitiendo que múltiples jugadores se unan y jueguen al mismo tiempo sin interferir entre sí. Esto requerirá la implementación de un sistema de hilos o procesos para manejar cada partida de manera independiente.

4. **Control del tiempo de la partida**: El servidor debe controlar el tiempo de cada partida para determinar cuándo finaliza. Se puede utilizar la función 'time' de Python para medir el tiempo transcurrido desde el inicio de la partida.

5. **Sincronización entre jugadores**: Es necesario sincronizar la interacción entre los jugadores y el servidor para evitar condiciones de carrera y asegurar un acceso seguro al tablero compartido. Esto implicará el uso de mecanismos de sincronización como bloqueos o semáforos.

6. **Manejo de categorías y palabras válidas**: El servidor debe validar las palabras ingresadas por los jugadores para asegurarse de que cumplan con las reglas del juego, como comenzar con la letra asignada y estar en mayúscula. También debe controlar que cada categoría del tablero se complete correctamente antes de finalizar la partida.

4 Arquitectura del Servidor

La arquitectura del servidor del juego del STOP consta de los siguientes componentes principales:

1. **Servidor principal**: Este componente es responsable de escuchar las solicitudes de conexión de los clientes y crear un hilo o proceso dedicado para manejar cada nueva partida. Utilizará la biblioteca 'socket' de Python para establecer y gestionar las conexiones.

2. **Partida**: Cada instancia de partida está asociada a un grupo de jugadores que comparten un tablero de juego. El servidor crea y gestiona una instancia de partida para cada grupo de jugadores que se unen para jugar. La clase 'Partida' se encarga de administrar la lógica del juego, incluyendo la gestión del tablero, el temporizador, la validación de palabras, etc.

3. **Tablero compartido**: El tablero es un recurso compartido entre los jugadores y el servidor, donde se registran las palabras ingresadas por los jugadores en cada categoría. La clase 'Partida' contiene el estado del tablero y gestiona su actualización y sincronización entre los jugadores.

4. **Gestión de conexiones**: El servidor gestiona múltiples conexiones de clientes simultáneas utilizando hilos para cada conexión. Cada hilo se encarga de escuchar los movimientos de un jugador específico en su partida asociada.

La interacción entre estos componentes permite la ejecución concurrente de múltiples partidas del juego del STOP, garantizando la sincronización adecuada entre los jugadores y el tablero compartido.

5 Implementación del Cliente/Servidor

5.1 Api

En esta parte se presenta el código modificado de la api que permite a los jugadores crear partidas de juego.

```
from bottle import Bottle, run, response, request
import datetime
import subprocess
import string
import random
import socket
import json
import threading

app = Bottle()

@app.route('/hi')
def hello():
    now = datetime.datetime.now()
    return f'Hola, hoy es {now.strftime("%d/%m/%Y")} y son las {now.strftime("%H:%M:%S")}'

@app.route('/status')
def status():
    process = subprocess.Popen(['systemctl', 'list-units', '--type=service', '--state=running'
], stdout=subprocess.PIPE)
    output, error = process.communicate()
    lines = output.decode('utf-8').split('\n')
    services = []
    for line in lines:
        parts = line.split()
        if len(parts) >= 4:
            services.append({
                'unit': parts[0],
                'load': parts[1],
                'active': parts[2],
```

```
                    'sub': parts[3],
                    'description': ' '.join(parts[4:])
                })

        # Genera una tabla HTML
        html = '<table border="1">'
        html += '<tr><th>Unit</th><th>Load</th><th>Active</th><th>Sub</th><th>Description</th></tr>'
        for service in services:
            html += f'<tr><td>{service["unit"]}</td><td>{service["load"]}</td><td>{service["active"]}</td><td>{service["sub"]}</td><td>{service["description"]}</td></tr>'
        html += '</table>'

        response.content_type = 'text/html'
        return html


    @app.route('/stop/new')
    def nueva_partida():
        id = random.randint(4250, 4500)
        return str(id)


    @app.route('/stop/<id>')
    def partida_existente(id: str):
        return id


    if __name__ == '__main__':
        run(app, host='0.0.0.0', port=8080)
```

Listing 1: Código Python

5.2 Supervisor: stop.conf

En esta parte se presenta el código que mantiene en ejecución continua Supervisor, lo que permite jugar al Stop en cualquier momento sin tener que lanzar manualmente el código del *stop_server.py*. Este archivo está guardado en el directorio */etc/supervisor/conf.d/* del servidor.

```
[program:stop_server]
command=/usr/bin/python3 /home/scripts/stop_server.py  # Ruta absoluta al script stop_server.py
directory=/home/scripts  # Directorio donde se encuentra stop_server.py
autostart=true
autorestart=true
stderr_logfile=/var/log/stop_server.err.log
stdout_logfile=/var/log/stop_server.out.log
```

Listing 2: Código Python

Hay pequeños cambios en el siguiente código respecto al original por motivos de errores en LaTex (Estos cambios no afectan al rendimiento del código)

5.3 Servidor

Aquí se incluye el código del servidor que se encarga de gestionar las conexiones entrantes de los clientes, manejar las partidas simultáneas, y actualizar el estado del juego según las acciones de los jugadores.

```
import threading
import socket
import json
import random
import string
import time


class Partida:
    def __init__(self) -> None:
        # Inicializacion del tablero con palabras vacias y categorias desbloqueadas
        self.tablero = {
            'Marca': {'word': '', 'locked': False, 'filled_by': ''},
            'Lugar': {'word': '', 'locked': False, 'filled_by': ''},
            'Comida': {'word': '', 'locked': False, 'filled_by': ''},
            'Animal': {'word': '', 'locked': False, 'filled_by': ''}
        }

        # Eleccion aleatoria de una letra del alfabeto mayuscula
        self.letra = random.choice(string.ascii_uppercase)

        # Inicializacion de los mutex para cada categoria del tablero
        self.mutex = {
                'Marca': threading.Lock(),
                'Lugar': threading.Lock(),
                'Comida': threading.Lock(),
                'Animal': threading.Lock()
        }

        # Inicializacion del diccionario de jugadores vacio
        self.jugadores = {}

        # Inicializacion del temporizador para el juego con una duracion de 300 segundos (5
    minutos)
        self.temporizador = threading.Timer(300, self.temporizador_expirado)

        # Bandera que indica si el juego ha terminado
        self.endgame = False

    def tablero_completo(self) -> bool:
        # Itera sobre cada categoria del tablero
        for _, word in self.tablero.items():
            w = word['word']                    # Obtiene la palabra de la categoria
            if w == '':                         # Si la palabra esta vacia
                return False                    # El tablero no esta completo
        return True                             # Si todas las palabras estan llenas, el tablero
    esta completo

    def enviar_cambio(self, msg: str = '') -> None:
        # Envia los datos actualizados del tablero y la letra a todos los jugadores
        for _, conn_socket in self.jugadores.items():
            conn_socket.send(json.dumps((self.tablero, self.letra, msg)).encode())

        # Verifica si el tablero esta completo
        if self.tablero_completo():
```

```
            self.endgame = True  # Si el tablero esta completo, establece la bandera de fin de juego

    def bloq_categ(self, categ: str) -> None:
        self.mutex[categ].acquire()             # Adquiere el cierre para la categoria especificada
        self.tablero[categ]['locked'] = True    # Establece el estado de bloqueo de la categoria en True
        self.enviar_cambio()                    # Envia una actualizacion a todos los jugadores
        time.sleep(8)                           # Espera 8 segundos durante el tiempo de bloqueo
        self.mutex[categ].release()             # Libera el cierre de la categoria
        self.tablero[categ]['locked'] = False   # Restablece el estado de bloqueo de la categoria en False
        self.enviar_cambio()                    # Envia una actualizacion a todos los jugadores

    def iniciar_temporizador(self) -> None:
        # Verifica si el temporizador esta inactivo
        if not self.temporizador.is_alive():
            self.temporizador.start()   # Inicia el temporizador

    def temporizador_expirado(self) -> None:
        self.enviar_cambio(msg='tiempo_agotado')    # Envia un mensaje de tiempo agotado a todos los jugadores
        self.endgame = True                         # Establece la bandera de fin de juego en True


reg_partidas = {}


def escuchar(connex: socket, partida: Partida, reg_partida: dict, id_partida: str) -> None:
    partida.iniciar_temporizador()      # Comienza el temporizador de la partida
    connex.send(json.dumps((partida.tablero, partida.letra, '')).encode())    # Envia los datos iniciales del tablero y la letra al jugador recien conectado

    while not partida.endgame:          # Mientras el juego no haya terminado
        categ = connex.recv(1024).decode()    # Recibe la categoria seleccionada por el jugador
        if partida.endgame:                 # Si el juego ha terminado, rompe el bucle
            break

        threading.Thread(target=partida.bloq_categ, args=(categ,)).start()  # Inicia un hilo para bloquear la categoria seleccionada por el jugador
        p_n = connex.recv(1024).decode()                                    # Recibe la palabra y el nombre del jugador que la eligio

        if partida.endgame:  # Si el juego ha terminado, rompe el bucle
            break

        palabra, name = json.loads(p_n)
        partida.tablero[categ]['word'] = palabra    # Actualiza el tablero con la palabra
        partida.tablero[categ]['filled_by'] = name   # Actualiza el tablero con el nombre del jugador
        partida.enviar_cambio()                     # Envia una actualizacion del tablero a todos los jugadores

    if id_partida in reg_partida.keys():   # Si la partida esta registrada, elimina la partida del registro al finalizar
        del(reg_partida[id_partida])
        print(f"Se elimina la partida {id_partida}\n{reg_partida}")


if __name__ == '__main__':
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)   # Creacion de un socket TCP/IP
    s.bind(('0.0.0.0', 9090))                               # Vinculacion del socket a la direccion y puerto deseados
    s.listen()                                              # Escucha de conexiones entrantes
```

```
107
108     while True:
109         conn, addr = s.accept()          # Aceptacion de una nueva conexion
110         print(f'Conexion de {addr}.')
111
112         nom = conn.recv(10).decode()     # Recepcion del nombre del jugador
113
114         id_partidas = conn.recv(4).decode()  # Recepcion del identificador de la partida
115
116         if id_partidas not in reg_partidas.keys():   # Verificacion y gestion de la partida en el
    registro de partidas
117             reg_partidas[id_partidas] = Partida()
118
119         p = reg_partidas[id_partidas]    # Obtencion de la partida correspondiente
120         p.jugadores[nom] = conn          # Asignacion del socket del jugador a la partida
121         print(p.jugadores)
122
123         threading.Thread(target=escuchar, args=(conn, p, reg_partidas, id_partidas,)).start()    #
     Inicio de un hilo para escuchar los movimientos del jugador en la partida
```

Listing 3: Código Python

5.4 Cliente

En esta parte se presenta el código del cliente que permite a los jugadores conectarse al servidor, enviar sus movimientos y recibir actualizaciones sobre el estado del juego.

```
1   import threading    # Importar el modulo threading para manejar multiples hilos de ejecucion
2   import socket       # Importar el modulo socket para la comunicacion de red
3   import requests     # Importar el modulo requests para realizar solicitudes HTTP
4   import json
5   import time
6   from collections import Counter     # Importar la clase Counter para contar elementos
7
8
9   def mostrar_tablero(table: dict) -> None:
10      print('\n')  # Salto de linea para separar el tablero
11      for categ, word in table.items():
12          w = word['word']         # Palabra de la categoria
13          b = word['locked']       # Estado de bloqueo de la categoria
14          f = word['filled_by']    # Quien lleno la categoria
15          if b:
16              print(f'{categ} => *{f}* (BLOQUEADA)')             # Si la categoria esta
    bloqueada
17          else:
18              print(f'{categ} => *{f}* (DESBLOQUEADA) {w} ')     # Si la categoria no esta
    bloqueada
19
20
21  def tablero_completo(table: dict) -> bool:
22      for _, w in table.items():  # Iterar sobre cada categoria en el tablero
23          if not w['word']:         # Si la palabra asociada a una categoria esta vacia
24              return False          # Entonces el tablero no esta completo
25      return True                   # Si todas las palabras estan llenas, el tablero esta
    completo
26
27
28  def mostrar_resultados(table: dict) -> None:
29      r = []                          # Lista para almacenar quien lleno cada categoria
30      for _, w in table.items():    # Iterar sobre cada categoria en el tablero
31          r.append(w['filled_by'])        # Agregar quien lleno la categoria a la lista
32      conteo = Counter(r)             # Contar cuantas veces aparece cada jugador en la lista
33      print('\n\n')                   # Salto de linea para separar los resultados
34      for j, n in conteo.items():     # Iterar sobre los resultados del conteo
```

```
            if j == '':                         # Si no hay jugador asociado
                print(f'Hay {n} palabras que nadie ha logrado.')    # Mostrar el numero de palabras sin jugador asociado
            else:
                print(f'{j} ha logrado {n} palabras.')              # Mostrar el numero de palabras logrado por el jugador


    def leer() -> None:
        # Declarar variables globales
        global s
        global tablero
        global letra
        global endgame

        endgame = False                                     # Inicializar la variable de fin de juego como False
        while not endgame:                                  # Bucle para leer continuamente datos hasta que el juego termine
            tablero, letra, msg = json.loads(s.recv(1024).decode())     # Leer datos recibidos y decodificar JSON
            if tablero_completo(tablero):                               # Verificar si el tablero esta completo
                print('\n\nPARTIDA FINALIZADA, TABLERO COMPLETO')
                endgame = True                                          # Establecer el fin del juego como True
                mostrar_resultados(tablero)                             # Mostrar resultados finales del juego
            else:
                if not msg:                                             # Si no hay mensaje adicional
                    mostrar_tablero(tablero)                                # Mostrar el tablero actual
                    print(f'Letra aleatoria: {letra}')
                elif msg == 'tiempo_agotado':                       # Si el mensaje indica que el tiempo se ha agotado
                    print('\n\nPARTIDA FINALIZADA, TIEMPO AGOTADO')      # Mostrar mensaje de finalizacion por tiempo agotado
                    endgame = True                                      # Establecer el fin del juego como True
                    mostrar_resultados(tablero)                         # Mostrar resultados finales del juego


    def escribir(name: str) -> None:
        # Declarar variables globales
        global s
        global tablero
        global letra
        global endgame

        while not endgame:      # Bucle para esperar las respuestas del jugador hasta que el juego termine
            time.sleep(.5)          # Esperar medio segundo antes de solicitar la entrada del jugador
            categ = input('Introduce el nombre de una categoria: ')     # Solicitar al jugador el nombre de una categoria
            if endgame:                                                 # Si el juego ha terminado
                break
            while tablero[categ]['locked']:                         # Si la categoria esta bloqueada en el tablero
                categ = input('Esa categoria esta bloqueada, escoge otra: ')    # Solicitar al jugador otra categoria
                if endgame:                                                     # Si el juego ha terminado
```

```
                    break
            if endgame:                                             # Si el juego ha terminado
                break

            s.send(categ.encode())  # Enviar el nombre de la categoria al servidor

            palabra = input('Introduce la palabra: ')   # Solicitar al jugador que introduzca una palabra
            if endgame:                                 # Si el juego ha terminado
                break
            while palabra[0] != letra:                  # Si la palabra no comienza con la letra correcta
                palabra = input(f'La palabra debe comenzar por {letra}: ')  # Solicitar al jugador otra palabra
                if endgame:                                                 # Si el juego ha terminado
                    break
            if endgame:     # Si el juego ha terminado
                break
            s.send(json.dumps((palabra, name)).encode())    # Enviar la palabra y el nombre del jugador al servidor


    if __name__ == '__main__':
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)   # Crear un objeto de socket TCP/IP
        s.connect(("13.53.94.147", 9090))                       # Conectar el socket al servidor y puerto especificados

        nom = input('Introduce tu nombre: ')  # Solicitar al jugador que introduzca su nombre

        s.send(nom.encode())  # Enviar el nombre del jugador al servidor

        id_partida = input('Introduce el ID de la partida o deja en blanco para crear una nueva: ')  # Solicitar al jugador que introduzca el ID de la partida

        if not id_partida:  # Si no se proporciona un ID de partida
            id_partida = requests.get('http://13.53.94.147:8080/stop/new').text  # Obtener nuevo ID de partida del servidor
        else:
            id_partida = requests.get(
                f'http://13.53.94.147:8080/stop/{id_partida}').text     # Obtener ID de partida del servidor con el ID dado

        s.send(id_partida.encode())  # Enviar el ID de partida al servidor

        print(f'Te conectaste a la partida {id_partida}!')

        # Iniciar hilos para la lectura y escritura de datos en paralelo
        hilo_r = threading.Thread(target=leer).start()
        hilo_w = threading.Thread(target=escribir, args=(nom,)).start()
```

Listing 4: Código Python

6 Sincronización y Concurrencia

En la versión actual del juego STOP!, los participantes seleccionan la categoría que desean completar. Al hacerlo, dicha categoría queda inaccesible para los demás jugadores por un lapso de 8 segundos, periodo durante el cual el jugador tiene la oportunidad de anotar su palabra. Finalizado este intervalo, la categoría se desbloquea y está disponible para todos nuevamente. Cada partida del juego se almacena en una estructura de diccionario, donde cada llave corresponde a un elemento específico: el tablero de juego, la lista de jugadores activos y el puerto de conexión.

7 Pruebas y Resultados

Adjuntas como un vídeo explicativo.

8 Conclusiones

Completar esta práctica ha sido un verdadero desafío, lleno de pruebas y errores, pero sobre todo, de aprendizajes significativos. Terminar esta práctica me ha generado una satisfacción enorme por los logros obtenidos.

Se han cumplido los requisitos técnicos, incluyendo la comunicación a través de sockets y la gestión eficaz de múltiples partidas y la sincronización entre jugadores y servidor.

Los logros más notables incluyen la implementación de la lógica del juego del STOP!, con una estructura que soporta la gestión de la concurrencia y sincronización mediante hilos y 'Locks', asegurando la integridad del juego. Además, la comunicación cliente/servidor se ha optimizado para una interacción fluida y en tiempo real.

Para futuras mejoras, se podría crear una interfaz de usuario más amigable y la adición de nuevas funcionalidades que enriquezcan la experiencia del juego, un sistema de puntuación y un chat entre jugadores. En resumen, el desarrollo de este servidor para el juego STOP! no solo ha sido un gran reto sino también una valiosa oportunidad de aprendizaje y un avance significativo en el ámbito de los servidores y concurrencia de procesos.

9 Referencias

1. **Material del Aula Virtual**: incluye documentos y presentaciones proporcionados por el profesorado. Así como páginas web de referencia.
2. **Apuntes personales**: notas tomadas durante las clases y el estudio individual.
3. **Clases específicas**: lecciones impartidas que se centran en aspectos clave del proyecto como por ejemplo la implementación de los hilos entre otras.
4. **Interacción con compañeros**: discusiones y colaboraciones que aportan diferentes perspectivas y resoluciones de problemas específicos.
5. **Vídeos de YouTube**: recursos audiovisuales que complementan la información y ofrecen explicaciones adicionales o ejemplos prácticos.