Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación

Ingeniería en Ciencias de la Computación

Materia: Programación Distribuida Aplicada

Practica 1

Profesor: Gustavo Emilio Mendoza Olguin

Alumnos:

Pérez Flores Ivonne 202141158

Primavera 2025

18 de enero de 2025

Introducción

Como practica uno, se plantea la realización de un programa servidor de nombres de recursos de archivos, el cual debe de cumplir con los siguientes requisitos:

- El proceso debe de obtener la lista de archivos de una carpeta de la computadora (especificada por el usuario) y crear una lista dinámica de objetos que contenga:
    - Nombre del archivo y extensión
    - Permitir al usuario definir que archivos pueden publicarse y cuales no y el tiempo de actualización (TTL) (Guardar un archivo de configuración).
    - Crear un hilo que actualice la lista de archivos cada 5 minutos, si hay un archivo que estaba en la lista anterior, entonces debe preguntar si se podrá publicar, si existe un archivo en la lista que ya no esta en la carpeta, entonces deberá eliminarse (los mensajes de cambio de la lista deben enviarse a un log)
    - El archivo de configuración debe actualizarse cada vez que inicie el proceso con los cambios en el sistema de archivos

Asimismo, se requiere de otro hilo que inicie un socket que escuche en el puerto 50000 UDP.

Desarrollo

Para comenzar con el desarrollo de esta práctica, se consideraron varios aspectos para la implementación, los cuales se enlistan a continuación:

- Para la recopilación de los archivos desde la ruta proporcionada por el usuario, se toman en cuenta solo aquellos que tienen una extensión en el nombre del archivo, excluyendo carpetas del listado.
- El almacenamiento de un archivo de configuración se realiza mediante un archivo json para facilitar el acceso a los datos del listado.
- Los mensajes de cambio se almacenan en un archivo de extensión .log.
- El lenguaje de programación utilizado para esta práctica es Python.

Una vez considerando los pasados puntos, vamos a desglosar cada parte del código para comprender su funcionamiento.

Importación de librerías de Python

```
1   import threading
2   import time
3   import os
4   import json
5   import socket
6   import logging
```

Estas librerías cumplen con los siguientes aspectos:

- **Threading.** Librería para gestión y creación de hilos.
- **Time.** Gestión de tiempo en Python, nos permitirá suspender o poner a dormir un hilo el tiempo necesario.
- **Os.** Permite interactuar con el sistema operativo, se utilizará para las consultas de los listados de archivos en el sistema.
- **Json.** Librería para gestionar archivos json.
- **Socket.** Creación y gestión de sockets en Python.
- **Logging.** Librería para generar mensajes de registro para archivos log.

Clase server

Constructor

```python
def __init__(self):
    self.ruta_salida = "config.json"
    # Configuración del logger
    logging.basicConfig(
        filename="historial.log",
        level=logging.INFO,
        format="%(asctime)s - %(levelname)s - %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S"
    )
    self.lock = threading.Lock()
```

Definimos valores globales, tales como la ruta del archivo de configuración “config.json”, el formato de salida para el archivo de extensión .log y el mecanismo de sincronización para establecer una zona critica.

Método para iniciar conexión de socket

```python
def iniciar_socket_udp(self):
    #Inicializacion de socket
    udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    host=socket.gethostname()
    port=50000
    print(f"Iniciando socket UDP en el puerto {port}...")
    udp_socket.bind(("0.0.0.0", port))
    clientes_conectados = set()
    while True:
        data, addr = udp_socket.recvfrom(2048)  # Incrementar tamaño si es necesario
        mensaje = json.loads(data.decode("utf-8"))
        print(mensaje)
        if addr not in clientes_conectados:
            clientes_conectados.add(addr)
        print(f"Nuevo cliente conectado desde {addr[0]}:{addr[1]}")

        if "addr" in mensaje:  # Guardar datos de archivos
            self.guardarEnJsonCliente(mensaje)
            threading.Thread(target=self.guardarEnJsonCliente, args=(mensaje,)).start()
        elif "ruta" in mensaje:  # Validar ruta
            ruta = mensaje["ruta"]
            validacion = self.validarRuta(ruta, addr)
            udp_socket.sendto(json.dumps(validacion).encode("utf-8"), addr)
        time.sleep(300)
```

El método iniciar_socket_udp configura y gestiona un servidor UDP para recibir y procesar mensajes de clientes. Inicializa el socket, lo vincula al puerto 50000 y escucha constantemente los datos entrantes, recibidos en formato json. Al recibir un mensaje, verifica si el cliente es nuevo y lo registra. Según el contenido del mensaje, puede guardar información en un archivo JSON o validar una ruta específica, enviando la respuesta correspondiente al cliente. Cada entrada de cliente es definida por un hilo para seguir su comportamiento.

Validar ruta

```
validarRuta(self, ruta, direccion):
respuesta = {"valida": False, "archivos": [], "leer": True}  # Estructura de respuesta inicial

if not os.path.isdir(ruta):  # Verifica si la ruta es un directorio válido
    logging.error(f"Ruta no válida proporcionada: {ruta} por {direccion}")
    print("La ruta proporcionada no es válida. Intente nuevamente.\n")
    return respuesta
else:
    try:
        # Obtener contenido del directorio y filtrar archivos
        contenido_bruto = os.listdir(ruta)
        archivos = [archivo for archivo in contenido_bruto if '.' in archivo]

        # Verifica si la ruta ya está en el archivo JSON
        if archivos:
            with self.lock:
                # Comprobar si la ruta ya está en el JSON
                with open(self.ruta_salida, "r", encoding="utf-8") as archivo_json:
                    datos_existentes = json.load(archivo_json)

            ruta_existente = next((item for item in datos_existentes if item["ruta"] == ruta), None)

            if ruta_existente:
                archivos_existentes = {archivo["nombre"] for archivo in ruta_existente["archivos"]}
                archivos_nuevos = [archivo for archivo in archivos if archivo not in archivos_existente
```

```
            if not archivos_nuevos:
                # Si no hay archivos nuevos, añadir "leer": False para indicar que el clier
                respuesta["leer"] = False
            else:
                # Si hay archivos nuevos, agregar solo esos archivos a la respuesta
                respuesta["archivos"] = archivos_nuevos
                logging.info(f"Archivos obtenidos desde la ruta: {ruta} por {direccion}")
        else:
            # Si no existe en el archivo JSON, agregar todos los archivos encontrados
            respuesta["archivos"] = archivos
            logging.info(f"Archivos obtenidos desde la ruta: {ruta} por {direccion}")

        respuesta["valida"] = True  # Ruta válida
    else:
        logging.info(f"No se encontraron archivos en la ruta: {ruta} por {direccion}")

    return respuesta
except PermissionError:
    print("No tiene permiso para acceder a esta ruta.\n")
    logging.error(f"Permiso denegado para acceder a la ruta: {ruta} por {direccion}")
    return respuesta
except Exception as e:
    print(f"Error inesperado al intentar acceder a la ruta: {e}\n")
    logging.error(f"Error inesperado al intentar acceder a la ruta: {e} por {direccion}")
    return respuesta
```

El método validarRuta comprueba si la ruta proporcionada es válida y devuelve información sobre los archivos disponibles en ella. Inicializa una respuesta predeterminada indicando que la ruta no es válida y que no se deben procesar archivos. Si la ruta no corresponde a un directorio válido, se registra un error y se devuelve la respuesta sin cambios. En caso de que la ruta sea válida, intenta obtener los archivos presentes en el directorio, filtrando aquellos que contienen un punto en el nombre. Luego, verifica si la ruta ya está registrada en el archivo JSON. Si está registrada, compara los archivos existentes con los nuevos, actualizando la respuesta según corresponda. Si no está registrada, agrega todos los archivos encontrados y marca la ruta como válida. El método gestiona errores como permisos denegados o fallos inesperados, registrando los eventos en un log para garantizar un seguimiento adecuado.

Método para guardar y gestionar el uso de archivos de configuración y logs

```python
def guardarEnJsonCliente(self, mensaje_completo):
    with self.lock:
        self.getContenido(mensaje_completo["ruta"])
        if os.path.exists(self.ruta_salida):
                try:
                    # Abrir el archivo JSON para leer los datos existentes
                    with open(self.ruta_salida, "r", encoding="utf-8") as archivo_json:
                        datos_existentes = json.load(archivo_json)

                    # Verificar si la ruta ya existe en el archivo JSON
                    ruta_existente = next((item for item in datos_existentes if item["ruta"] == mensaje
                    print(ruta_existente)

                    if ruta_existente:
                        # Si la ruta existe, verificar si hay archivos en el mensaje
                        if mensaje_completo["archivos"]:  # Solo continuar si hay archivos en el mensaj
                            archivos_existentes = {archivo["nombre"] for archivo in ruta_existente["arc

                            datos_a_guardar = []

                            # Identificar archivos nuevos o modificados
                            for archivo in mensaje_completo["archivos"]:
                                if archivo["nombre"] not in archivos_existentes:
                                    datos_a_guardar.append(archivo)
```

```python
            if datos_a_guardar:
                for nuevo_dato in datos_a_guardar:
                    if not any(dato["nombre"] == nuevo_dato["nombre"] for dato in ruta_existente["archiv
                        ruta_existente["archivos"].append(nuevo_dato)
                logging.info(f"Archivos agregados: {datos_a_guardar}")
                print("Se agregaron nuevos archivos en la ruta especificada")

        else:
            logging.info(f"No se encontraron archivos nuevos para la ruta: {mensaje_completo['ruta']}")
            print(f"No se encontraron archivos nuevos para la ruta: {mensaje_completo['ruta']}")

    else:
        # Si la ruta no existe, agregarla como nueva
        nuevo_dato = {
            "ruta": mensaje_completo["ruta"],
            "archivos": mensaje_completo["archivos"]
        }
        datos_existentes.append(nuevo_dato)
        logging.info(f"Nueva ruta agregada: {mensaje_completo['ruta']}")
        print(f"Se agregó una nueva ruta: {mensaje_completo['ruta']}")

    # Escribir los cambios en el archivo JSON
    with open(self.ruta_salida, "w", encoding="utf-8") as archivo_json:
        json.dump(datos_existentes, archivo_json, ensure_ascii=False, indent=4)
```

```
            except Exception as e:
                print(f"Error al leer o actualizar el archivo JSON: {e}")
                logging.error(f"Error al actualizar el archivo JSON: {e}")
    else:
        # Si el archivo JSON no existe, crear uno nuevo con los datos recibidos
        try:
            with open(self.ruta_salida, "w", encoding="utf-8") as archivo_json:
                json.dump([mensaje_completo], archivo_json, ensure_ascii=False, indent=4)
            logging.info(f"Archivo JSON creado con los datos iniciales: {mensaje_completo}")
            print("Se guardaron los datos en el archivo JSON")

        except Exception as e:
            print(f"Error al crear el archivo JSON: {e}")
            logging.error(f"Error al crear el archivo JSON: {e}")

    print("Esperando 5 minutos para la próxima actualización...")
```

El método guardarEnJsonCliente se encarga de gestionar la actualización de un archivo JSON con la información recibida de los clientes, utilizando un bloqueo (self.lock) para definir una zona crítica y asegurar que múltiples hilos puedan interactuar de manera segura con el archivo.

Si el archivo JSON ya existe, el método lo abre y verifica si la ruta especificada en el mensaje ya está registrada. Si la ruta existe, compara los archivos del mensaje con los ya almacenados, agregando únicamente los nuevos para mantener actualizada la información sin duplicados. En caso de que la ruta no esté registrada, la agrega como una nueva entrada junto con los archivos asociados.

En los casos en que el archivo JSON no existe, el método crea un nuevo archivo e inicializa su contenido con los datos del mensaje recibido. Cualquier evento relevante, como la adición de nuevas rutas o archivos, se registra en un log para facilitar el seguimiento. Asimismo, se maneja cualquier error que pueda ocurrir durante la lectura o escritura del archivo.

Por último, el método incluye un retardo de cinco minutos antes de realizar nuevas actualizaciones, lo que permite optimizar los recursos y reducir la frecuencia de acceso al archivo en entornos concurrentes.

Clase client

Constructor

```python
class Client:
    def __init__(self):
        self.s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.host=socket.gethostname()
        self.port=50000
```

El constructor de la clase Client configura un cliente para la comunicación mediante UDP. Inicializa un socket, obtiene el nombre del host local y define el puerto a utilizar para las conexiones.

Establecer conexión con el servidor

```python
def establecerConexion(self):
    try:
        self.s.connect((self.host, self.port))  # Establecer conexión con el servidor
        print("Conexión establecida con el servidor.")
        print("Inserte la ruta del directorio a conocer su listado:")
        ruta = input()
        while True:
            mensaje = {"ruta": ruta}
            self.s.sendto(json.dumps(mensaje).encode("utf-8"), (self.host, self.port))
            try:
                data, _ = self.s.recvfrom(2048)
                respuesta = json.loads(data.decode("utf-8"))

                if (respuesta["valida"]):
                    if(not respuesta["leer"]):
                        print("No hay archivos nuevos")
                    else:
                        archivos = respuesta["archivos"]
                        if(archivos):
                            datos_archivos = []
                            for archivo in archivos:
                                ttl = input(f"Ingrese el TTL para '{archivo}': ")
                                try:
                                    ttl = int(ttl)
                                except ValueError:
                                    print("Valor inválido. Se asignará 3600 por defecto.")
                                    ttl = 3600
```

```
                        publish = input(f"¿Publicar '{archivo}'? (S/N): ").strip().lower() == 's'
                        datos_archivos.append({
                            "nombre": archivo,
                            "ttl": ttl,
                            "publish": publish
                        })

                    mensaje_completo = {
                        "addr": self.s.getpeername()[0],  # Dirección del cliente
                        "ruta": ruta,  # Ruta proporcionada
                        "archivos": datos_archivos  # Datos de los archivos
                    }

                    self.s.sendto(json.dumps(mensaje_completo).encode("utf-8"), (self.host, self.p
                else:
                    print("No hay archivos almacenado en esa ruta")
                #print("Ruta validada correctamente")
            else:
                print("Ruta invalida, inserte nuevamente una ruta valida")
                ruta = input()
            #print(f"Respuesta del servidor: {data.decode('utf-8')}")
        except socket.timeout:
            print("El servidor no respondió. Intente nuevamente.")
        #time.sleep(10)
except ConnectionRefusedError:
    print("No se pudo establecer conexión con el servidor. Verifique si está en ejecución.")
```

El método establecerConexion establece la conexión con el servidor y gestiona la interacción con el usuario para obtener información sobre una ruta de directorio. Primero, intenta conectar con el servidor usando el socket y, si la conexión es exitosa, pide al usuario que ingrese la ruta del directorio a consultar.

Luego, envía la ruta al servidor y espera la respuesta. Si la ruta es válida, verifica si hay archivos nuevos. En caso afirmativo, solicita al usuario el TTL para cada archivo y si desea publicarlos. Los datos se envían de vuelta al servidor. Si la ruta no es válida, solicita una nueva ruta.

El método maneja posibles errores como la falta de conexión con el servidor o la ausencia de respuesta, mostrando un mensaje adecuado al usuario en cada caso.

Ejecución

Servidor

```
Iniciando socket UDP en el puerto 50000...
{'ruta': 'C:\\Users\\Bobe\\Pictures\\Camera Roll'}
Nuevo cliente conectado desde 192.168.0.12:55841
{'addr': '192.168.0.11', 'ruta': 'C:\\Users\\Bobe\\Pictures\\Camera Roll', 'archivos': [{'nombre': 'desktop.
ini', 'ttl': 200, 'publish': True}, {'nombre': 'WIN_20240816_20_31_40_Pro.jpg', 'ttl': 100, 'publish': False
}, {'nombre': 'WIN_20240816_20_31_49_Pro.jpg', 'ttl': 100, 'publish': True}]}
Nuevo cliente conectado desde 192.168.0.12:55841
```

```
Se agregó una nueva ruta: C:\Users\Bobe\Pictures
Esperando 5 minutos para la próxima actualización...
{'ruta': 'C:\\Users\\Bobe\\Pictures', 'archivos': [{'nombre': 'desktop.ini', 'ttl': 357, 'publish': True},
'nombre': 'logo.png', 'ttl': 8763, 'publish': False}]}
Esperando 5 minutos para la próxima actualización...
```

Clientes

```
C:\Users\dayan\OneDrive\Documentos\ProgramacionDistribuida\Practica1>py client.py
Conexión establecida con el servidor.
Inserte la ruta del directorio a conocer su listado:
C:\Users\Bobe\Pictures\Camera Roll
{'ruta': 'C:\\Users\\Bobe\\Pictures\\Camera Roll'}
Ingrese el TTL para 'desktop.ini': 200
¿Publicar 'desktop.ini'? (S/N): s
Ingrese el TTL para 'WIN_20240816_20_31_40_Pro.jpg': 100
¿Publicar 'WIN_20240816_20_31_40_Pro.jpg'? (S/N): n
Ingrese el TTL para 'WIN_20240816_20_31_49_Pro.jpg': 100
¿Publicar 'WIN_20240816_20_31_49_Pro.jpg'? (S/N): s
{'ruta': 'C:\\Users\\Bobe\\Pictures\\Camera Roll'}
```

```
PROBLEMS 89  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SPELL CHECKER 88  COMMENTS
PS C:\Users\ivonn\OneDrive\Documentos\PDA\dis> py .\client.py
Conexión establecida con el servidor.
Inserte la ruta del directorio a conocer su listado:
C:\Users\Bobe\Pictures
Ingrese el TTL para 'desktop.ini': 357
¿Publicar 'desktop.ini'? (S/N): s
Ingrese el TTL para 'logo.png': 8763
¿Publicar 'logo.png'? (S/N): n
```

Archivos de configuración

```
{
    "ruta": "C:\\Users\\Bobe\\Pictures\\Camera Roll",
    "archivos": [
        {
            "nombre": "desktop.ini",
            "ttl": 200,
            "publish": true
        },
        {
            "nombre": "WIN_20240816_20_31_40_Pro.jpg",
            "ttl": 100,
            "publish": false
        },
        {
            "nombre": "WIN_20240816_20_31_49_Pro.jpg",
            "ttl": 100,
            "publish": true
        }
    ]
},
```

```
{
    "ruta": "C:\\Users\\Bobe\\Pictures",
    "archivos": [
        {
            "nombre": "desktop.ini",
            "ttl": 357,
            "publish": true
        },
        {
            "nombre": "logo.png",
            "ttl": 8763,
            "publish": false
        }
    ]
}
```

Archivo de historial .log

```
2025-01-23 18:34:04 - INFO - Archivos obtenidos desde la ruta: C:\Users\Bobe\Pictures\Camera Roll por (
2025-01-23 18:34:34 - INFO - Nueva ruta agregada: C:\Users\Bobe\Pictures\Camera Roll
2025-01-23 18:36:04 - INFO - Archivos obtenidos desde la ruta: C:\Users\Bobe\Pictures por ('192.168.0.1
2025-01-23 18:37:04 - INFO - Nueva ruta agregada: C:\Users\Bobe\Pictures
```

Conclusión

En conclusión, el programa establece una comunicación eficiente entre el cliente y el servidor mediante el uso de sockets UDP, permitiendo al cliente enviar solicitudes de validación de rutas y obtener información sobre los archivos disponibles. La interacción se maneja de manera dinámica, permitiendo al usuario ingresar rutas, recibir información sobre archivos y decidir sobre su publicación. Además, el programa está diseñado para manejar errores de conexión y validar la entrada del usuario, asegurando que el proceso sea robusto y estable.

La implementación de mecanismos de sincronización y control de errores proporciona una experiencia de usuario fluida y segura, garantizando que el intercambio de datos entre cliente y servidor se realice sin problemas. Dado el alcance actual, se considera que se han logrado los objetivos planteados inicialmente. En trabajos posteriores, se implementarán mecanismos para interactuar como servidor DNS y las estrategias necesarias para su implementación.