Tarea N°2:

Sockets

Cristian Ignacio Herrera Badilla
Cristóbal Tomás David Alberto Lagos Valtierra
Escuela de Ingeniería, Universidad de O'Higgins
18 de Noviembre del 2023

I. Introducción

En el vasto mundo de la programación de redes, los sockets se establecen como herramientas fundamentales para establecer comunicación entre dispositivos a través de Internet. En este contexto, la presente tarea nos ha planteado el desafío de desarrollar un chat online que no solo permita la interacción instantánea entre granjeros de Stardew Fridi, sino que también posibilite el intercambio de artefactos coleccionables, creando así un espacio virtual donde pueda realizarse una comunicación entre granjeros cuidando la integridad de los datos, así como el correcto funcionamiento del Cliente y el Servidor.

La importancia de los sockets en esta tarea radica en su capacidad para facilitar la comunicación bidireccional entre el cliente y el servidor, proporcionando el marco necesario para la transmisión de datos en tiempo real.

En el desarrollo de un chat online, es preciso abordar la cuestión de la estabilidad del servidor. Un servidor robusto y bien gestionado es esencial para mantener la continuidad del chat y garantizar que los granjeros puedan disfrutar de una comunicación sin interrupciones.

En este informe, exploraremos los desafíos enfrentados durante la implementación del chat para granjeros, destacando las decisiones de diseño que influyeron en la estabilidad, integridad de los datos y la eficacia de la comunicación.

II. Marco Teórico

A. Arquitectura Cliente-Servidor

La arquitectura cliente-servidor es un modelo fundamental en el diseño de sistemas distribuidos. Este modelo define roles específicos para los nodos de una red: el cliente, que solicita servicios o recursos, y el servidor, que proporciona esos servicios o recursos.

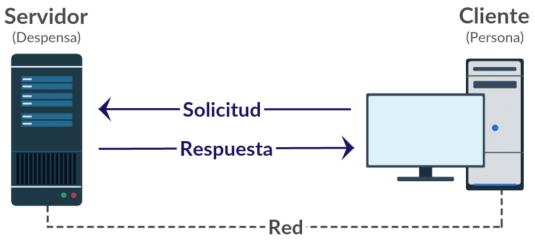

Fig. 1: Arquitectura Cliente-Servidor

La arquitectura cliente-servidor se basa en la comunicación distribuida a través de redes informáticas. En este modelo, los clientes y servidores intercambian datos utilizando protocolos específicos adaptados para la transmisión eficiente de información en entornos de red. Esto es esencial para la funcionalidad del modelo, ya que permite la interacción fluida entre distintos nodos de la red.

Se establece una jerarquía de roles claramente definida entre los nodos que participan en la comunicación. En esta estructura, los clientes, que son las entidades que solicitan servicios o recursos, se sitúan en un nivel jerárquico inferior respecto a los servidores, que poseen y proporcionan los servicios demandados. Esta relación jerárquica permite una distribución eficiente de tareas, ya que los servidores, al ser especializados en la gestión de recursos, tienen la capacidad de atender múltiples solicitudes simultáneas provenientes de diversos clientes, garantizando así una eficiencia operativa.

B. Socket

Un socket es una interfaz de programación de aplicaciones (API) que proporciona un conjunto de funciones para la comunicación entre procesos a través de una red, ya sea en un mismo dispositivo o entre varios. Un socket se puede ver como un extremo de una conexión bidireccional, permitiendo que los programas envíen y reciban datos entre sí. Se basan en el modelo de protocolo TCP/IP. Corresponden a una puerta que sirve principal como comunicación entre la capa de aplicación y la capa de transporte.

Podemos discriminar dos tipos principales de sockets: los sockets de servidor, que esperan conexiones entrantes, y los sockets de cliente, que inician la conexión. La interacción entre estos sockets permite la comunicación entre diferentes programas o sistemas, facilitando la transmisión de información de un extremo a otro.

El funcionamiento de un socket, en cuanto a su implementación y funcionamiento, se puede entender de la siguiente forma:

1) *Creación del Socket:* Tanto en el cliente como en el servidor, se crea un socket utilizando las funciones proporcionadas por la API de sockets. En sistemas basados en TCP/IP, esto implica la creación de un socket utilizando las funciones socket().
2) *Configuración del Socket:* Se configuran las propiedades del socket, como el tipo de conexión (TCP o UDP) y la dirección IP y el número de puerto asociados. En el caso del servidor, enlaza el socket a una dirección IP y un puerto específicos, para finalmente entrar en un

estado de escucha. Cuando un cliente intenta conectarse, el servidor acepta la conexión entrante y crea un socket dedicado a esa conexión. En el caso del cliente, intenta establecer una conexión utilizando la dirección ip y el puerto al que desea establecer conexión.

3) *Transferencia de datos:* Una vez que la conexión está establecida, ambos extremos del socket pueden enviar y recibir datos.
4) *Cierre de conexión:* Al finalizar la comunicación, ambos extremos cierran sus respectivos sockets.

En cuanto al socket utilizado en la implementación en particular es el "socket.SOCK_STREAM" el cual se trata de un socket de flujo. Estos son adecuados para la comunicación de tipo TCP, el cual está orientado a proporcionar una comunicación confiable y ordenada entre aplicaciones. Son varias las características que hacen de TCP el protocolo más utilizado cuando se requiere comunicación confiable, las cuales se detallan a continuación:

1) *Orientado a la Conexión:* Antes de que las aplicaciones puedan intercambiar datos, deben establecer una conexión. Esta conexión es bidireccional y garantiza la entrega de datos en orden y sin pérdida.
2) *Comunicación Confiable:* Utiliza mecanismos de confirmación y retransmisión para asegurarse de que los datos lleguen correctamente al destino. Si se detecta pérdida de datos, se vuelven a enviar para garantizar la integridad de la comunicación.
3) *Control de Flujo:* Implementa un mecanismo de control de flujo para evitar que un extremo sobrecargue al otro con datos. Esto asegura que el receptor pueda procesar los datos de manera eficiente, evitando la pérdida de paquetes debido a la congestión.
4) *Control de Congestión:* Incorpora mecanismos de control de congestión para gestionar la cantidad de datos que se envían en la red. Esto ayuda a evitar la saturación de la red y garantiza un rendimiento óptimo.
5) *Orden de Datos:* TCP garantiza que los datos enviados desde un extremo se reciban en el mismo orden en el que fueron enviados. Esto es crucial para aplicaciones que dependen de la secuencia correcta de los datos, como la transmisión de objetos o la comunicación en tiempo real.
6) *Detección y Recuperación de Errores:* TCP utiliza sumas de verificación y números de secuencia para detectar y corregir errores en la transmisión de datos. Si se identifica un error, se solicita la retransmisión de los datos afectados.

C. Threads

Son la unidad más pequeña de un proceso que pueden ser programados para ser ejecutados por el sistema operativo. Un proceso puede tener varios hilos, cada uno de los cuales ejecuta una tarea independiente, pero comparte los mismos recursos, como la memoria y los descriptores de archivos. En Python, el módulo *threading* proporciona una interfaz para trabajar con hilos, la cual cuenta con las siguientes características:

1) *Concurrencia:* Los hilos permiten que varias tareas se ejecuten simultáneamente en un programa, lo que puede mejorar la eficiencia y la capacidad de respuesta.
2) *GIL (Global Interpreter Lock):* Python utiliza el GIL para garantizar que solo un hilo ejecute el código de Python a la vez en un proceso.
3) *Módulo threading:* El módulo *threading* facilita la creación y gestión de hilos.
4) *Comunicación entre hilos:* Los hilos pueden compartir datos utilizando estructuras de datos como listas o colas.
5) *Sincronización:* Proporciona varios mecanismos de sincronización, como bloqueos (*Locks*) y semáforos (*Semaphores*), que ayudan a controlar el acceso a recursos compartidos entre hilos.

III. Metodología

A. Servidor

A continuación, veremos la implementación por parte del Servidor del código necesario para poder gestionar la comunicación entre usuarios, intercambio de artefactos, envío de emoticonos, conexión y desconexión de usuarios, entre otros. Para añadir estas funcionalidades se tuvo que cuidar aspectos como la integridad de datos debido a la concurrencia de envío de artefactos y mensajes entre usuarios, así como el estado del servidor para evitar caídas por un uso indebido de la aplicación por parte del usuario. Destaca en esta implementación la utilización de 2 librerías, las cuales son socket, para la correcta configuración de los sockets que se utilizan para la comunicación entre los usuarios y el servidor, así como la librería threading, que permite la creación de hilos y herramientas de sincronización para aquellos escenarios en los que se modifican variables, las cuales pueden ser alteradas por más de un usuario al mismo tiempo.

Código 1: Variables globales y mutex

```
# Cargar el JSON desde el archivo 'artefactos.json'
with open('artefactos.json', 'r') as json_file:
    data_artifacts = json.load(json_file)

# Diccionario para guardar los intercambios
clients_exchange = {}

# Crear un mutex
mutex = threading.Lock()
```

Código 2: Funcion principla del servidor

```
def handle_client(client_socket, clients, client_names,
    client_artifacts):

    # Recibe el nombre del cliente
    while True:
        name = client_socket.recv(1024).decode("utf-8")
        if is_name_taken(name, client_names):
            client_socket.send("[SERVER] Nombre ya en uso."
    .encode("utf-8"))
        else:
            welcome_message = f"[SERVER] Cliente {name}
    conectado."
            print(welcome_message)
            with mutex:
                client_names[client_socket] = name
            break

    # Envia el mensaje de bienvenida al cliente
    welcome_message_client = f"Bienvenid@ al chat de
    Granjerxs!"
```

```
    client_socket.send(welcome_message_client.encode("utf-8"))

    # Notifica a los otros clientes sobre la nueva conexi n
    broadcast_message = f"[SERVER] Cliente {name} conectado."
    broadcast(clients, client_socket, broadcast_message)


    # Bucle de artefactos
    while True:
        a = artefactos(client_socket, data_artifacts)
        desition = client_socket.recv(1024).decode("utf-8")
        if (desition.lower() == "si"):
            client_artifacts = a
            with mutex:
                client_names[client_socket] = {'name': name, 'artifacts': client_artifacts}
            break

    # Envia el mensaje de OK
    message_client_ok = f"[SERVER]  OK !"
    client_socket.send(message_client_ok.encode("utf-8"))

    # Bucle principal para manejar los mensajes del cliente
    while True:
        try:
            message = client_socket.recv(1024).decode("utf-8")
            if not message:
                break

            # Comandos
            if message.startswith(":q"):
                client_socket.send("[SERVER]    Adis    y suerte completando tu colecci n!".encode("utf-8"))
                remove_client(clients, client_names, client_socket)
                client_socket.close()
                break

            elif message.startswith(":p"):
                _, identifier, private_message = message.split(" ", 2)
                whisp(name, client_names, private_message, identifier)

            elif message.startswith(":u"):
                connected_users = map(lambda x: x['name'], client_names.values())
                connected_users = list(connected_users)
                client_socket.send(f"[SERVER] Usuarios conectados: {', '.join(connected_users)}".encode("utf-8"))


            # Emojis
            elif message.startswith(":smile"):
                emoticon_message = ":)"
                broadcast_message = f"{name}: {emoticon_message}"
                broadcast(clients, client_socket, broadcast_message)

            elif message.startswith(":angry"):
                emoticon_message = ">:("
                broadcast_message = f"{name}: {emoticon_message}"
                broadcast(clients, client_socket, broadcast_message)

            elif message.startswith(":combito"):
                emoticon_message = "Q(    -    Q )"
                broadcast_message = f"{name}: {emoticon_message}"
                broadcast(clients, client_socket, broadcast_message)

            elif message.startswith(":larva"):
                emoticon_message = "  Larva  !  (:o)OOOooo"
                broadcast_message = f"{name}: {emoticon_message}"
                broadcast(clients, client_socket, broadcast_message)

            # Logica de artefactos
            elif message.startswith(":artefactos"):
                client_socket.send(f"[SERVER] Tus artefactos son: \n{', '.join(client_artifacts)}".encode("utf-8"))

            elif message.startswith(":artefacto"):
                _, artifact = message.split(" ", 1)
                client_socket.send(f"[SERVER] El artefacto es: {ask_artifact(data_artifacts, artifact)}".encode("utf-8"))

            elif message.startswith(":offer"):
                _, identifier, my_artifact, his_artifact = message.split(" ", 3)
                his_artifact = ask_artifact(data_artifacts, his_artifact)
                my_artifact = ask_artifact(data_artifacts, my_artifact)

                # Si tengo el artefacto que quiero intercambiar y si el otro usuario tiene el item que quiero intercambiar
                if (my_artifact in client_artifacts and them_artifact(identifier, his_artifact, client_names)):
                    # Pregunta al usuario correspondiente
                    exchange(name, client_names, f"su {my_artifact} por tu {his_artifact}", identifier)

                    #Agregamos en un diccionario los artefactos que se van a cambiar
                    with mutex:
                        clients_exchange[identifier] = str(name), my_artifact, his_artifact
                    #print(f"Tradeo: {clients_exchange}")
                else:
                    client_socket.send(f"[SERVER] Error de Intercambio".encode("utf-8"))

            elif message.startswith(":accept"):
                #aceptamos la solicitud de intercambio
                accept(name,client_artifacts, clients_exchange,client_names)
                clients_exchange.pop(name)
                client_socket.send(f"[SERVER] Intercambio aceptado, ahora tus artefactos son: \n{', '.join(client_artifacts)}\n".encode("utf-8"))

            elif message.startswith(":reject"):
                # Mensaje de Intercambio rechazado
                reject_message = f"[SERVER] Rechazaste el intercambio"
                client_socket.send(reject_message.encode("utf-8"))

                identifier = clients_exchange[name][0]

                reject(client_names, identifier)
                with mutex:
                    clients_exchange.pop(name)


            else:
                broadcast_message = f"{name}: {message}"
                broadcast(clients, client_socket, broadcast_message)

        except Exception as e:
            break

    # Elimina al cliente desconectado
    remove_client(clients, client_names, client_socket)

    # Notifica a los demas usuarios
    disconnect_message = f"[SERVER] {name} desconectado."
    broadcast(clients, client_socket, disconnect_message)

    # Cierre de socket
    client_socket.close()
```

B. Cliente

La implementación por parte del cliente, empieza conectándose a un servidor en el localhost (127.0.0.1) y

puerto 5555 utilizando sockets TCP/IP. El programa inicia un hilo para recibir mensajes del servidor de manera asíncrona mientras permite al usuario enviar mensajes y responder a las solicitudes del servidor. La función receive_messages maneja la recepción de mensajes del servidor, mostrándolos en la consola, y realiza acciones específicas dependiendo del contenido del mensaje, como solicitar al usuario su nombre, artefactos que posee, su respuesta a una pregunta, o permitirle enviar mensajes adicionales al servidor. La aplicación utiliza la biblioteca threading para ejecutar múltiples tareas simultáneamente, y el código está estructurado en una función principal llamada start_client que establece la conexión, inicia el hilo de recepción y maneja la interacción inicial con el usuario, como ingresar el nombre de usuario. Implementa un cliente de chat interactivo que se comunica con un servidor y permite al usuario participar en conversaciones y responder a solicitudes específicas del servidor.

Código 3: Toda funcionalidad del codigo del cliente

```
import socket
import threading

def receive_messages(client_socket):
    while True:
        try:
            message = client_socket.recv(1024).decode("utf-8")
            print(message)

            # Nombre ya ocupado
            if message == "[SERVER] Nombre ya en uso.":
                name = input("Ingresa tu nombre: ")
                client_socket.send(name.encode("utf-8"))

            if message == "[SERVER] Cu ntame,  qu artefactos tienes?":
                artifacts = input("Ingresa tus artefactos: ")
                client_socket.send(artifacts.encode("utf-8"))

            if message == "[SERVER]  Est  Bien?":
                desition = input("Ingresa tus respuesta: ")
                client_socket.send(desition.encode("utf-8"))

            if message == "[SERVER]  OK !":
                while True:
                    message_client = input()
                    client_socket.send(message_client.encode("utf-8"))
                    if (message_client.startswith(":q")):
                        break
                break

        except Exception as e:
            print(f"[ERROR] {e}")
            break

def start_client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(("127.0.0.1", 5555))  # Con ctate al servidor en el localhost y puerto 5555

    # Inicia un hilo para recibir mensajes del servidor
    receive_thread = threading.Thread(target=receive_messages, args=(client_socket,))
    receive_thread.start()

    # Ingresar nombre de usuario
    name = input("Ingresa tu nombre: ")
    client_socket.send(name.encode("utf-8"))

    receive_messages(client_socket)

if __name__ == "__main__":
    start_client()
```

IV. Discusión y Conclusiones

La implementación del servidor se destaca por la gestión integral de la comunicación entre usuarios, el intercambio de artefactos, el envío de emoticonos, así como la administración de conexiones y desconexiones. El código del servidor se enfoca en aspectos cruciales como la integridad de datos, especialmente ante la concurrencia generada por el intercambio de artefactos y mensajes entre usuarios.

En el contexto del cliente, la implementación comienza estableciendo la conexión con el servidor, seguido de la inicialización de un hilo encargado de recibir mensajes de forma asíncrona. Esta estructura permite que el usuario envíe mensajes y responda a las solicitudes del servidor en un entorno interactivo. La función receive_messages se encarga de manejar la recepción de mensajes del servidor, presentándolos en la consola, y tomando acciones específicas según el contenido del mensaje, como la solicitud del nombre del usuario, información sobre artefactos, respuestas a preguntas, o el envío de mensajes adicionales al servidor. La implementación demuestra un uso eficaz de la biblioteca threading para ejecutar múltiples tareas simultáneamente, creando una experiencia de usuario interactiva y receptiva.

Gracias a este trabajo, pudimos entender y aplicar los distintos requerimientos que se necesitan para implementar un paradigma Cliente-Servidor aplicado en la creación de un chat online, así como el intercambio de artefactos entre los distintos usuarios.

References

[1] A. Tanenbaum., *Computer Networkss, 5o Ed.* Prentice-Hall, 2010.
[2] D. Comer, *Internetworking with TCP/IP, Vol 1, 6th Edition*. Pearson, 2013.
[3] J. F. Kurose, *Computer Networking, 8th Edition*. Pearson, 2020.