Problema 2: Implementación de un protocolo de red personalizado sobre TCP

Contexto

Diseñar un protocolo de aplicación personalizado para un sistema de archivos distribuido que se ejecutará sobre TCP, utilizando técnicas como multiplexación y control de flujo.

Requisitos

- Definir el formato del mensaje, incluyendo cabeceras y extensiones de cabecera.
- Desarrollar un esquema de control de flujo que gestione eficazmente la transferencia de archivos grandes a través de redes con alta latencia.
- Escribir un pseudocódigo para las funciones de conexión, como el handshake de tres vías de TCP, y cómo se manejará la retransmisión.
- Evaluar el uso de NAT y su impacto en las conexiones de red en este protocolo, particularmente cuando se utilizan direcciones IP dinámicas.

Objetivos

1. **Diseño del protocolo**: Definir el formato del mensaje, incluidas las cabeceras y las extensiones de cabecera para manejar funcionalidades específicas como control de flujo y recuperación de errores.
2. **Implementación de control de flujo**: Desarrollar un esquema de control de flujo para manejar eficazmente la transferencia de datos sobre TCP.
3. **Simulación con Python**: Simular el protocolo utilizando Python para evaluar su rendimiento y robustez en escenarios de red simulados.

Paso 1: Diseño del protocolo

Vamos a definir un protocolo simple que incluya operaciones básicas como PUT, GET y DELETE para interactuar con archivos en el sistema distribuido.

Ejemplo de Especificación del Protocolo:

- PUT: Enviar un archivo al sistema.
- GET: Recuperar un archivo del sistema.
- DELETE: Eliminar un archivo del sistema.

Cada mensaje tendrá una cabecera que incluye el tipo de operación, el tamaño del mensaje y un número de secuencia para el control de flujo y la recuperación de errores.

En esta parte definiremos:

- Las operaciones básicas que nos solicitan: PUT, GET, DELETE.
- El formato del mensaje, que constará de una cabecera y cuerpo de datos.

- La definición de los mensajes, utilizando la biblioteca `struct` para empacar y desempaquetar los datos del mensaje en la cabecera, y luego codificar el cuerpo en forma de texto.
- El control de errores y manejo de flujo de datos para que los mensajes se entreguen en el orden correcto y para verificar la pérdida de datos.
- La evaluación de la eficacia, verificando la eficiencia al momento de ver los resultados con la simulación en los escenarios de la red de alta y baja latencia o pérdida de errores.

Paso 2: Implementación de control de flujo

Usaremos un mecanismo de control de flujo basado en ventana deslizante para asegurar la entrega fiable y eficiente de los archivos, especialmente en redes con alta latencia.

Código Python para simulación del protocolo:

```
import socket
import struct

def send_message(sock, msg_type, seq_num, data):
    header = struct.pack('!I I', msg_type, seq_num)
    message = header + data.encode()
    sock.sendall(message)

def receive_message(sock):
    header = sock.recv(8)
    msg_type, seq_num = struct.unpack('!I I', header)
    data = sock.recv(1024) # ajustar según el tamaño esperado del mensaje
    return msg_type, seq_num, data.decode()

def main():
    host = 'localhost'
    port = 12345
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((host, port))
    server.listen(1)
    print("Server listening on port", port)
    client_sock, addr = server.accept()
    print("Connected by", addr)
    msg_type, seq_num, data = receive_message(client_sock)
    print("Received:", msg_type, seq_num, data)
    send_message(client_sock, 1, seq_num + 1, "Ack")
    client_sock.close()
    server.close()

if __name__ == '__main__':
    main()
```

Código mejorado

```
import socket
import struct

# Definición de constantes para tipos de mensaje
MSG_PUT = 1
```

```
MSG_GET = 2
MSG_DELETE = 3

# Definición de constantes para estados del control de flujo
STATE_WAITING_ACK = 0
STATE_WAITING_DATA = 1

def send_message(sock, msg_type, seq_num, data):
    # Empaqueta el mensaje con la cabecera y lo envía
    header = struct.pack('!I I', msg_type, seq_num)
    message = header + data.encode()
    sock.sendall(message)

def receive_message(sock):
    # Recibe un mensaje y lo desempaqueta
    header = sock.recv(8)
    msg_type, seq_num = struct.unpack('!I I', header)
    data = sock.recv(1024)  # ajustar según el tamaño esperado del mensaje
    return msg_type, seq_num, data.decode()

def main():
    host = 'localhost'
    port = 12345

    # Configuración del servidor TCP
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((host, port))
    server.listen(1)
    print("Server listening on port", port)

    # Acepta la conexión entrante
    client_sock, addr = server.accept()
    print("Connected by", addr)

    # Inicialización del control de flujo
    state = STATE_WAITING_DATA
    expected_seq_num = 0

    # Bucle principal para la comunicación
    while True:
        if state == STATE_WAITING_DATA:
            # Espera recibir un mensaje GET
            msg_type, seq_num, data = receive_message(client_sock)
            if msg_type == MSG_GET and seq_num == expected_seq_num:
                print("Received GET:", seq_num, data)
                # Simulación de procesamiento de la solicitud GET
                response_data = f"File content for {data}"
                send_message(client_sock, MSG_GET, seq_num + 1, response_data)
                expected_seq_num += 1
            else:
                print("Error: Unexpected message or sequence number")

        elif state == STATE_WAITING_ACK:
            # Espera recibir un mensaje ACK
            msg_type, seq_num, data = receive_message(client_sock)
```

```
            if msg_type == MSG_ACK and seq_num == expected_seq_num: # type: ignore
                print("Received ACK:", seq_num)
                state = STATE_WAITING_DATA
            else:
                print("Error: Unexpected message or sequence number")

    # Cierra la conexión
    client_sock.close()
    server.close()

if __name__ == '__main__':
    main()
```

Paso 3: Evaluación del protocolo

Después de implementar el protocolo, usaríamos herramientas como Wireshark para monitorear la eficacia del control de flujo y el manejo de errores durante la transferencia de archivos. Esto podría involucrar la simulación de condiciones de red adversas, como alta latencia y pérdida de paquetes, para ver cómo el protocolo se comporta y se recupera de estos problemas.

Integrantes

- Huanca Hampuero Lila Zaray
- Gavino Isidro Michael Richard
- Manosalva Peralta Yojan Alexander