CODICE CLIENT

```
"""
Implementazione del Client per il protocollo Go-Back-N ARQ
Implementazione con socket UDP
"""

import socket
import threading
import time
import random
import json
import os
import glob
from datetime import datetime
from typing import Dict, List, Optional


class GBNClient:
    """
    Client per il protocollo Go-Back-N ARQ con gestione della finestra scorrevole,
    timeout e ritrasmissione automatica dei pacchetti.
    """

    def __init__(self, server_host: str = "localhost", server_port: int = 12345,
                 window_size: int = 3, timeout: float = 2.0, loss_probability: float = 0.0):
        """
        Inizializza il client Go-Back-N.

        Args:
            server_host (str): Indirizzo IP del server
            server_port (int): Porta del server
            window_size (int): Dimensione della finestra scorrevole
            timeout (float): Timeout in secondi per la ritrasmissione
            loss_probability (float): Probabilità di perdita pacchetti (0.0-1.0)
        """
        # Configurazione socket UDP
        self.server_host = server_host
        self.server_port = server_port
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.socket.settimeout(0.1)  # Timeout breve per non bloccare

        # Configurazione finestra scorrevole
        self.window_size = window_size
        self.base = 0  # Primo pacchetto non ancora confermato
        self.next_seq_num = 0  # Prossimo numero di sequenza da inviare
        self.window = {}  # Dizionario per tracciare i pacchetti nella finestra
```

```
        # Gestione timeout e ritrasmissione
        self.timeout_duration = timeout
        self.timers = {}  # Timer per ogni pacchetto
        self.timer_lock = threading.Lock()

        # Simulazione perdita pacchetti
        self.loss_probability = loss_probability

        # Statistiche e logging
        self.stats = {
            'packets_sent': 0,
            'packets_received': 0,
            'retransmissions': 0,
            'acks_received': 0,
            'packets_lost': 0
        }
        self.log_entries = []
        self.running = False

        # Gestione file di log
        self.log_file = self._get_log_filename()
        self._setup_log_file()

    def _get_log_filename(self):
        """Genera il nome del file di log con numerazione incrementale basata su sessione."""
        # Crea la cartella logs se non esiste
        if not os.path.exists('logs'):
            os.makedirs('logs')

        # Trova il prossimo numero di sessione disponibile
        session_number = self._get_next_session_number()

        if session_number == 0:
            return os.path.join('logs', 'demo_run.log')
        else:
            return os.path.join('logs', f'demo_run{session_number}.log')

    def _get_next_session_number(self):
        """Trova il prossimo numero di sessione disponibile controllando tutti i possibili file."""
        # Pattern per trovare tutti i file demo_run*.log
        pattern = os.path.join('logs', 'demo_run*.log')
        existing_files = glob.glob(pattern)

        if not existing_files:
            return 0

        # Estrai tutti i numeri dai nomi dei file esistenti
        numbers = []
```

```
    for file_path in existing_files:
        filename = os.path.basename(file_path)
        if filename == 'demo_run.log':
            numbers.append(0)
        elif filename.startswith('demo_run') and filename.endswith('.log'):
            try:
                # Estrai il numero tra 'demo_run' e '.log'
                number_str = filename[8:-4]  # Rimuovi 'demo_run' e '.log'
                if number_str.isdigit():
                    numbers.append(int(number_str))
            except:
                continue

    if not numbers:
        return 0

    # Restituisci il prossimo numero disponibile
    return max(numbers) + 1

def _setup_log_file(self):
    """Inizializza il file di log con header."""
    try:
        with open(self.log_file, 'w', encoding='utf-8') as f:
            f.write(f"=== LOG SESSIONE GO-BACK-N CLIENT ===\n")
            f.write(f"Avvio: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")
            f.write(f"Server: {self.server_host}:{self.server_port}\n")
            f.write(f"Window Size: {self.window_size}, Timeout: {self.timeout_duration}s\n")
            f.write(f"Loss Probability: {self.loss_probability}\n")
            f.write("="*60 + "\n\n")
    except Exception as e:
        print(f"Errore creazione file log: {e}")
        self.log_file = None

def log_event(self, event_type: str, seq_num: int = None, message: str = ""):
    """
    Registra un evento nel log con timestamp dettagliato.

    Args:
        event_type (str): Tipo di evento (SEND, RECV, TIMEOUT, etc.)
        seq_num (int): Numero di sequenza del pacchetto
        message (str): Messaggio aggiuntivo
    """
    timestamp = datetime.now().strftime("%H:%M:%S.%f")[:-3]
    log_entry = f"[{timestamp}] CLIENT - {event_type}"

    if seq_num is not None:
        log_entry += f" SEQ:{seq_num}"
```

```
        if message:
            log_entry += f" - {message}"

        self.log_entries.append(log_entry)
        print(log_entry)

        # Salva nel file di log
        if self.log_file:
            try:
                with open(self.log_file, 'a', encoding='utf-8') as f:
                    f.write(log_entry + '\n')
            except Exception as e:
                print(f"Errore scrittura log: {e}")

    def create_packet(self, seq_num: int, data: str) -> bytes:
        """
        Crea un pacchetto con numero di sequenza e dati.

        Args:
            seq_num (int): Numero di sequenza
            data (str): Dati da inviare

        Returns:
            bytes: Pacchetto serializzato in JSON
        """
        packet = {
            'seq_num': seq_num,
            'data': data,
            'timestamp': time.time()
        }
        return json.dumps(packet).encode('utf-8')

    def simulate_packet_loss(self) -> bool:
        """
        Simula la perdita casuale di un pacchetto.

        Returns:
            bool: True se il pacchetto deve essere "perso", False altrimenti
        """
        if self.loss_probability > 0:
            lost = random.random() < self.loss_probability
            if lost:
                self.stats['packets_lost'] += 1
            return lost
        return False

    def send_packet(self, seq_num: int, data: str, is_retransmission: bool = False):
        """
```

```
        Invia un pacchetto al server con possibile simulazione di perdita.

        Args:
            seq_num (int): Numero di sequenza
            data (str): Dati da inviare
            is_retransmission (bool): True se è una ritrasmissione
        """
        packet = self.create_packet(seq_num, data)

        # Simula perdita pacchetto
        if self.simulate_packet_loss():
            self.log_event("LOSS", seq_num, "Pacchetto simulato come perso")
            return

        try:
            self.socket.sendto(packet, (self.server_host, self.server_port))
            self.stats['packets_sent'] += 1

            if is_retransmission:
                self.stats['retransmissions'] += 1
                self.log_event("RETX", seq_num, f"Ritrasmissione pacchetto: {data}")
            else:
                self.log_event("SEND", seq_num, f"Invio pacchetto: {data}")

        except Exception as e:
            self.log_event("ERROR", seq_num, f"Errore invio: {e}")

    def start_timer(self, seq_num: int):
        """
        Avvia un timer per un pacchetto specifico.

        Args:
            seq_num (int): Numero di sequenza del pacchetto
        """
        with self.timer_lock:
            if seq_num in self.timers:
                self.timers[seq_num].cancel()

            # Crea nuovo timer per il timeout
            timer = threading.Timer(self.timeout_duration, self.timeout_handler, [seq_num])
            self.timers[seq_num] = timer
            timer.start()
            self.log_event("TIMER_START", seq_num, f"Timer avviato
({self.timeout_duration}s)")

    def stop_timer(self, seq_num: int):
        """
        Ferma il timer per un pacchetto specifico.
```

```
    Args:
        seq_num (int): Numero di sequenza del pacchetto
    """
    with self.timer_lock:
        if seq_num in self.timers:
            self.timers[seq_num].cancel()
            del self.timers[seq_num]
            self.log_event("TIMER_STOP", seq_num, "Timer fermato")

def timeout_handler(self, seq_num: int):
    """
    Gestisce il timeout di un pacchetto, attivando la ritrasmissione Go-Back-N.

    Args:
        seq_num (int): Numero di sequenza del pacchetto in timeout
    """
    if not self.running:
        return

    self.log_event("TIMEOUT", seq_num, "Timeout scaduto - Avvio Go-Back-N")

    # Go-Back-N: ritrasmetti tutti i pacchetti dalla base della finestra
    for i in range(self.base, self.next_seq_num):
        if i in self.window:
            self.send_packet(i, self.window[i], is_retransmission=True)
            self.start_timer(i)

def receive_acks(self):
    """
    Thread per ricevere gli ACK dal server e aggiornare la finestra.
    """
    while self.running:
        try:
            data, addr = self.socket.recvfrom(1024)
            ack = json.loads(data.decode('utf-8'))
            ack_num = ack['ack_num']

            self.stats['acks_received'] += 1
            self.log_event("ACK_RECV", ack_num, f"ACK ricevuto da {addr}")

            # Aggiorna la base della finestra (ACK cumulativo)
            if ack_num >= self.base:
                # Ferma i timer per tutti i pacchetti confermati
                for seq in range(self.base, ack_num + 1):
                    self.stop_timer(seq)
                    if seq in self.window:
                        del self.window[seq]
```

```
                self.base = ack_num + 1
                self.log_event("WINDOW_UPDATE", None, f"Base finestra aggiornata a:
{self.base}")

            except socket.timeout:
                continue
            except Exception as e:
                if self.running:
                    self.log_event("ERROR", None, f"Errore ricezione ACK: {e}")

    def send_data(self, messages: List[str]):
        """
        Invia una lista di messaggi utilizzando il protocollo Go-Back-N.

        Args:
            messages (List[str]): Lista dei messaggi da inviare
        """
        self.running = True
        self.log_event("SESSION_START", None, f"Inizio trasmissione {len(messages)}
messaggi")

        # Avvia thread per ricevere ACK
        ack_thread = threading.Thread(target=self.receive_acks)
        ack_thread.daemon = True
        ack_thread.start()

        message_index = 0

        while message_index < len(messages) or len(self.window) > 0:
            # Invia nuovi pacchetti se c'è spazio nella finestra
            while (len(self.window) < self.window_size and
                   message_index < len(messages)):

                data = messages[message_index]
                self.window[self.next_seq_num] = data
                self.send_packet(self.next_seq_num, data)
                self.start_timer(self.next_seq_num)

                self.log_event("WINDOW_ADD", self.next_seq_num,
                               f"Aggiunto alla finestra (size: {len(self.window)})")

                self.next_seq_num += 1
                message_index += 1

            time.sleep(0.1)  # Breve pausa per evitare loop intensivo

            # Verifica se tutti i messaggi sono stati confermati
```

```
            if message_index >= len(messages) and len(self.window) == 0:
                break

        # Attendi un po' per eventuali ACK in ritardo
        time.sleep(1)
        self.running = False

        # Ferma tutti i timer rimanenti
        with self.timer_lock:
            for timer in self.timers.values():
                timer.cancel()
            self.timers.clear()

        self.log_event("SESSION_END", None, "Trasmissione completata")

    def print_statistics(self):
        """
        Stampa le statistiche della trasmissione in formato tabellare.
        """
        print("\n" + "="*60)
        print("STATISTICHE TRASMISSIONE GO-BACK-N ARQ")
        print("="*60)
        print(f"{'Pacchetti inviati:':<25} {self.stats['packets_sent']:>10}")
        print(f"{'Pacchetti ricevuti:':<25} {self.stats['packets_received']:>10}")
        print(f"{'ACK ricevuti:':<25} {self.stats['acks_received']:>10}")
        print(f"{'Ritrasmissioni:':<25} {self.stats['retransmissions']:>10}")
        print(f"{'Pacchetti persi:':<25} {self.stats['packets_lost']:>10}")

        if self.stats['packets_sent'] > 0:
            loss_rate = (self.stats['packets_lost'] /
                         (self.stats['packets_sent'] + self.stats['packets_lost'])) * 100
            retx_rate = (self.stats['retransmissions'] / self.stats['packets_sent']) * 100
            print(f"{'Tasso di perdita:':<25} {loss_rate:>9.1f}%")
            print(f"{'Tasso ritrasmissione:':<25} {retx_rate:>9.1f}%")

        print("="*60)

        # Salva le statistiche nel file di log
        if self.log_file:
            try:
                with open(self.log_file, 'a', encoding='utf-8') as f:
                    f.write(f"\n{datetime.now().strftime('%H:%M:%S')} - STATISTICHE FINALI:\n")
                    f.write(f"Pacchetti inviati: {self.stats['packets_sent']}\n")
                    f.write(f"ACK ricevuti: {self.stats['acks_received']}\n")
                    f.write(f"Ritrasmissioni: {self.stats['retransmissions']}\n")
                    f.write(f"Pacchetti persi: {self.stats['packets_lost']}\n")
                    if self.stats['packets_sent'] > 0:
                        loss_rate = (self.stats['packets_lost'] /
```

```
                        (self.stats['packets_sent'] + self.stats['packets_lost'])) * 100
                    retx_rate = (self.stats['retransmissions'] / self.stats['packets_sent']) * 100
                    f.write(f"Tasso di perdita: {loss_rate:.1f}%\n")
                    f.write(f"Tasso ritrasmissione: {retx_rate:.1f}%\n")
                f.write("="*60 + "\n")
            except Exception as e:
                print(f"Errore scrittura statistiche nel log: {e}")

    def close(self):
        """
        Chiude il socket del client.
        """
        self.running = False
        self.socket.close()


def test_client_without_loss():
    """
    Test del client senza simulazione di perdita pacchetti.
    """
    print("\n🧪 TEST 1: TRASMISSIONE SENZA PERDITA DI PACCHETTI")
    print("-" * 50)

    client = GBNClient(
        server_host="localhost",
        server_port=12345,
        window_size=3,
        timeout=2.0,
        loss_probability=0.0  # Nessuna perdita
    )

    messages = [
        "Messaggio 1: Hello World",
        "Messaggio 2: Go-Back-N ARQ",
        "Messaggio 3: Socket UDP",
        "Messaggio 4: Protocollo affidabile",
        "Messaggio 5: Fine trasmissione"
    ]

    try:
        client.send_data(messages)
        client.print_statistics()
    finally:
        client.close()


def test_client_with_loss():
    """
```

```
    Test del client con simulazione di perdita pacchetti.
    """
    print("\n🧪 TEST 2: TRASMISSIONE CON PERDITA DI PACCHETTI")
    print("-" * 50)

    client = GBNClient(
        server_host="localhost",
        server_port=12345,
        window_size=3,
        timeout=1.5,
        loss_probability=0.3  # 30% di perdita
    )

    messages = [
        "Messaggio 1: Test con perdita",
        "Messaggio 2: Simulazione realistica",
        "Messaggio 3: Go-Back-N robusto",
        "Messaggio 4: Gestione errori",
        "Messaggio 5: Ritrasmissione automatica",
        "Messaggio 6: Finestra scorrevole",
        "Messaggio 7: Protocollo UDP"
    ]

    try:
        client.send_data(messages)
        client.print_statistics()
    finally:
        client.close()


if __name__ == "__main__":
    print("🚀 CLIENT GO-BACK-N ARQ")
    print("Assicurati che il server sia in esecuzione prima di avviare i test.\n")

    input("Premi INVIO per avviare il TEST 1 (senza perdita)...")
    test_client_without_loss()

    input("\nPremi INVIO per avviare il TEST 2 (con perdita)...")
    test_client_with_loss()

    print("\n✅ Test completati!")
```