UNIVERSIDAD NACIONAL DE PILAR

Tecnicatura en Desarrollo de Software

Redes de Datos

Trabajo Practico Integrador

Comisión: 6
Miembros del grupo:

- Vera Francisco
- Kelly Jorge Nicolas
- Vivas Gonzalo

UNIVERSIDAD NACIONAL DE PILAR

Informe

● ¿Qué hace nuestro código? (Breve informe de funcionamiento)

Nuestro código implementa una comunicación confiable sobre el protocolo UDP (no confiable) mediante dos scripts en Python: server.py y client.py que actúan como receptor y emisor.

1. El emisor(cliente.py) toma un mensaje del usuario y le asigna un número de secuencia alternando entre 0 o 1, calcula un CRC-16 sobre los datos y envía el paquete completo al servidor. Para cerrar la sesión del archivo se debe ingresar un espacio vacío.

2. Después el servidor.py recibe el paquete y para simular ruido en la red aplicando la función maybe_corrupt según la probabilidad que se le mande altera un bit del paquete. Después intenta decodificar este paquete y recalcular CRC-16 sobre los datos recibidos y si está todo bien responde con ACK y si no coincide (CRC ERROR) responde con ACK. Tomamos la decisión de que el archivo servidor siempre esté activo.

3. Por último el cliente espera una respuesta con un timeout, si recibe el ACK correcto se considera como envío exitoso y alterna la secuencia a 1 para el próximo mensaje, en caso de que reciba NACK, un ACK incorrecto o sufra un timeout asume un fallo y envía el mismo paquete (ahora con secuencia 0) en un bucle de reintento hasta un máximo de MAX_RETRIES de 5 intentos antes de abortar el envío

● ¿Qué función cumple el CRC en la comunicación?

El CRC es el mecanismo que le permite al receptor saber si los datos que recibió están corruptos o no. En nuestro caso se realiza:
-client.py: Se calcula el crc16 del mensaje, se armar el paquete con los datos, secuencia (o seq), mensaje, y el crc.

-server.py: recibe el paquete recibido de client.py , toma el crc y lo calcula para conocer si está correcto. En nuestro caso, dentro de este archivo existe una función que cumple la función de simular una ”corrupción” de mensaje

- **¿Qué diferencias existen entre usar TCP y UDP para este tipo de tareas?**

La diferencia principal radica en que TCP es un protocolo confiable por naturaleza, trabaja con segmentos de datos y no hay pérdida de datos mientras que UDP es “no confiable” ya que si pierde datagramas no importa. Es decir para este tipo de tareas si la velocidad es crítica conviene usar UDP ya que podemos tolerar pérdidas

Característica	TCP	UDP
Confiabilidad	Alta (maneja errores, orden, duplicados)	Baja (no garantiza entrega ni orden)
Retransmisión	Automática	Manual (como en tu TP)
Velocidad	Más lento (por control de flujo	Más rápido (sin verificación interna)
Uso típico	Web, correo, archivos	Streaming, juegos, sensores

- **¿Qué ocurre cuando aumenta la probabilidad de error?**

Aumenta la posibilidad de que los mensajes lleguen incorrectos, lo que aumenta la cantidad de retransmisiones, el cliente va a recibir más NACK o timeouts. Y en nuestro caso, al llegar al límite se devuelve el mensaje de error y se aborta el mensaje.

Además esto generaría mayor tráfico y aumentaría el tiempo de entrega. Que en un entorno real de comunicaciones, sería el equivalente a una red con mucho ruido o interferencia.

● **¿Por qué se usa un número de secuencia?**

El número de secuencia (que alterna entre 0 y 1) sirve para evitar duplicados cuando se pierde un ACK.

Por ejemplo:

1. El cliente envía un mensaje con Seq=0.
2. El servidor lo recibe bien y responde con ACK 0.
3. Pero ese ACK 0 se pierde en la red.
4. El cliente espera, no recibe respuesta y vuelve a enviar el mensaje con Seq=0.
5. El servidor, que ya había recibido ese mensaje y ahora esperaba Seq=1, detecta que es un duplicado.
6. Entonces no lo procesa de nuevo, solo reenvía ACK 0 para confirmar.

Sin el número de secuencia, el servidor no podría distinguir entre un mensaje nuevo y uno repetido, y podría procesarlo dos veces por error.

● **¿Qué pasaría si el emisor no implementara retransmisión?**

Si el cliente no implementa la lógica de send_with_retries (es decir, si no reintentara en caso de NACK o timeout), la comunicación dejaría de ser confiable y se volvería insegura**.**

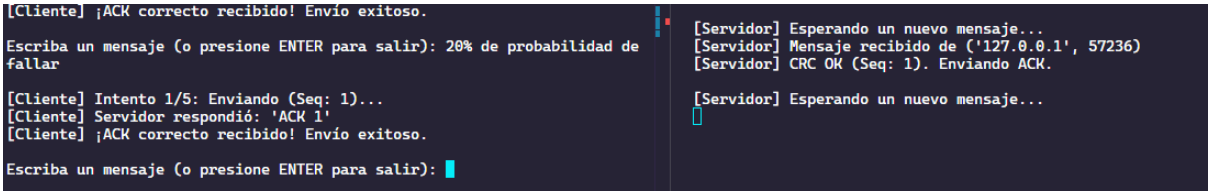
El sistema fallaría ante el mínimo error:

- Si un paquete se corrompe (NACK)**:** El cliente recibiría el "NACK", pero al no tener lógica de retransmisión, simplemente va a ignorar el error y el mensaje nunca se entregaría.

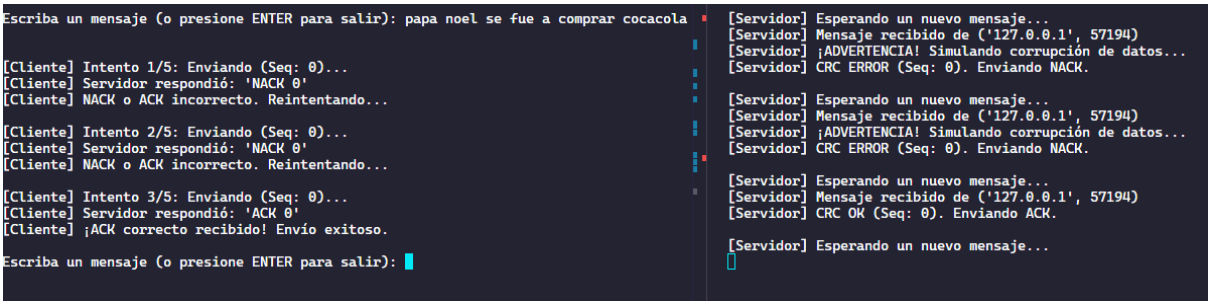
- Si un paquete se pierde (Timeout)**:** El cliente esperaría el TIMEOUT_S, la excepción saltaría, y el programa simplemente continuaría (o fallaría), pero el mensaje nunca se entregaría.
- Si el ACK/NACK se pierde (Timeout)**:** El cliente sufriría un timeout y, de nuevo, no haría nada al respecto, dejando la entrega del mensaje sin confirmar.

En conclusión, la función send_with_retries es el motor de la confiabilidad en este protocolo. Sin esta función, todo el sistema de CRC y ACK/NACK sería inútil, ya que solo serviría para detectar errores, pero no para hacer nada al respecto.

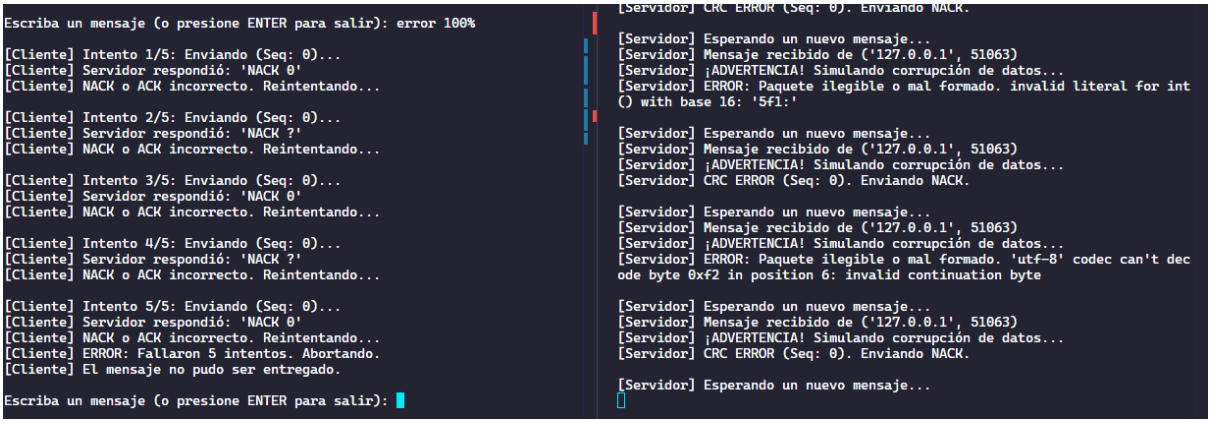
Captura de pantalla de pruebas



funcionamiento normal con 20% de fallas simuladas.



0.50 de fallo



en este caso falla el 100% de las veces para corroborar que exitosamente termine la ejecución de client.py después de 5 intentos.

Fuentes, librerías y guías:

https://www.geeksforgeeks.org/python/python-bitwise-operators/

https://pypi.org/project/crcmod/

https://ellibrodepython.com/operadores-bitwise

https://www.purestorage.com/la/knowledge/cyclic-redundancy-check.html