C1 REDES

Nombre: Patricio Espinoza Acuña

P1.

1. **Discuta si usaría TCP o UDP**

   Elegiría UDP ya que este no mantiene un estado y por tanto la carga como el costo es menor en el servidor, lo que a su vez permitirá que una mayor cantidad de sensores IOT a comparación de si se usa TCP, sumado a esto, las conexiones UDP serían notablemente más simples que una gestión para TCP dado el volumen de sensores a un solo servidor.

   Por otra parte, una desventaja de UDP es la pérdida de datos, TCP resulta mucho más seguro y estable. Sin embargo, considerando el contexto del problema en donde las 10 hectáreas siguen siendo un terreno que se encontrará en una localidad especifica, difícilmente dos sensores apartados por los 100 m^2 tendrán mediciones muy distintas, combinando esto con el hecho de que basta con recibir respuestas de cada sonda dos veces al día implica que la pérdida de datos no representa un riesgo debido a otros sensores cercanos y a la frecuencia de las mediciones (basta con 2 pero pueden hacerse más).

   Un argumento en contra es que TCP resguardaría la información crítica para determinar el momento de la cosecha, sin embargo, se entendería que UDP es capaz de recibir la cantidad mínima requerida de mediciones y a un menor costo.

   En el caso de que fueran exclusivamente 2 mediciones al día, la perdida de información sería crítica (perder 2 mediciones de una misma sonda por ejemplo), en dicho caso es preferible la estabilidad que brinda TCP frente al costo extra.

2. **¿El servidor debería conectarse a las sondas o las sondas al servidor?**

   Las sondas deberían conectarse al servidor ya que así el servidor es un punto centralizado a donde se envían los datos, si cada sonda requiere de un almacenamiento y esta perceptible a perdida de datos la comunicación se verá afectada. Por otra parte, el servidor puede comparar la información de múltiples sondas, y en caso de que una pierda información, basta con observar los registros de otra cercana para tener una idea aproximada sobre el estado de aquella que se perdió.

   Otra razón es que es más simple y escalable que todas las sondas se encarguen puramente del envío de datos e inicien el proceso de transmisión de datos hacia el servidor, que a su vez permite una mayor expansión en caso de incorporar más sensores (bastaría a que se conecten al servidor y puerto para enviar su paquete).

3. **¿El servidor debería estructurarse con procesos pesados, threads o select?**

   Para determinar el tipo de estructura deben considerarse los factores del problema:

   - Mediciones aisladas dos veces al día para cada sensor.
   - Gran cantidad de sensores.
   - Recepción simple del paquete con la información del sensor.

   Los procesos pesados tienen como desventaja un alto costo en los recursos del servidor, y entendiéndose que se trabaja en una planta de agricultura no se esperarían múltiples servidores o con los mejores recursos necesariamente. Por otro lado, una estructura mediante select tiene como ventajas ser notablemente más eficiente que las otras dos, sin embargo, está altamente recomendado no usarlo ya que tiene un debuggeo difícil y poca escalibilidad ante tantos clientes.

   Esto deja la opción de threads como una intermedia que resulta más fiable que select al hacer uso de hilos y menos costosa que los procesos pesados.

4. **Defina el protocolo para que el servidor reciba los datos**

   ≫ indica mensaje del servidor & ≪ indica mensaje de la sonda

Código	Significado del mensaje
≫ 01	Bienvenida del servidor
≫ 02	Identificación de la sonda confirmada desde el servidor
≫ 03	Recepción confirmada desde el servidor
≫ 04	Recepción invalida desde el servidor
≫ 05	Información parseada para el ID de la sonda
≫ 06	Información guardada en el servidor
≫ 07	Finalizar conexión del servidor
≪ e01	Identificación de la sonda
≪ e02	Envío de información de los sensores de la sonda
≪ e03	Finalizar conexión de la sonda

   Ejemplo:
   ≫01 Hola, soy servidor v.01
   ≪e001 soy la sonda57345
   ≫02 Bienvenido sonda 57345
   ≪e002 {ID:57345, hora: "Tue Apr 08 2025 17:38:01 GMT-0400", estados: {"temperatura": 23.5, "humedad": 15, "radiación_UV": 3.5}}. Información al servidor v.01
   ≫03 Información recibida sonda57345
   ≫05 Información parseada sonda57345
   ≫06 ACK, Información guardada sonda57345
   ≪e003 Fin de la comunicación con servidor v.01
   ≫07 Fin de la comunicación sonda57345

P2.

1. El modelo OSI de capas es una arquitectura que tiene como objetivo el diseño de redes con capacidad de comunicación e interoperabilidad entre ellas (trabajo en conjunto). Esto soluciona la previa práctica de utilizar un protocolo propio el cual resultaba en un problema debido a las dificultades derivadas de la unicidad de cada implementación. Asimismo, soluciona los problemas de red al tener la capacidad de aislar un error en una de sus capas.

   Fuentes: [Link, Link]

   Por otra parte, los sockets introdujeron la funcionalidad de facilitar la comunicación de los paquetes al ser recibidos o enviados de una forma uniforme. Esto soluciona tanto el problema de desarrollar una configuración de red de bajo nivel como el de que esta sea compatible en los sistemas.

   "El API de sockets hizo que el modelo cliente/servidor fuera facil de implementar, porque el programador podía usar una pequeña cantidad de llamadas de sistema y agregarlas a su código existente ("desconectado") para aprovechar otros recursos computacionales. Aunque era posible usar otros modelos, el API de sockets hizo que el modelo cliente/servidor dominara al mundo de las redes." Fuente: [Link]

2. Las principales dificultades de un protocolo binario son:

   Dificultad de depuración: No se puede inspeccionar fácilmente el contenido de los mensajes sin decodificadores específicos.

   Ambigüedad en la interpretación: Si no hay una especificación clara o no se manejan correctamente los errores, pequeñas diferencias en la implementación pueden causar problemas graves debido a su falta de legibilidad.

   Compatibilidad: Pueden tener problemas entre sistemas con arquitecturas distintas (endianness, tamaño de enteros, etc.).

   El protoloco binario puede ser bueno en algunos casos como para el uso de protocolos de bases de datos e IoT. Esto debido a que:
   - **Velocidad:** Tiene un procesamiento más rápido al evitar un parsing complejo.
   - **Eficiencia:** Tiene menos sobrecarga que protocolos textuales.

3. No tiene razón ya que el costo de añadir TSL es bajo comparado a las ventajas. HTTPS garantiza seguridad en la web mediante cifrado, integridad y autenticación (guardado de contraseñas, cookies y datos personales). Además, HTTPS no modifica el protocolo HTTP en sí puesto que lo encapsula en una capa adicional llamada TLS/SSL sin cambiar la lógica de las peticiones/respuestas HTTP.

4. El programa actual termina el envío de datos y duerme 3 segundos antes de cerrar el socket, esperando que en ese tiempo lleguen todas las respuestas del servidor. El problema con esta implementación es que genera un retraso que produce un error en la medición precisa del tiempo total de comunicación, con la que se calcula el ancho de banda.

   Dentro de las posibles soluciones a este problema están:

   Usar protocolo de terminación: El servidor puede enviar un mensaje especifico a modo de señal para el fin de la transmisión como “END”, permitiendo que el cliente espere hasta que haya recibido el indicador.

   Envío del tamaño por adelantado: Como en la T1 se puede calcular y enviar la longitud total de datos que se transmitirán, y luego el cliente puede recibir los datos hasta completar exactamente esa cantidad.

   Recepción bloqueante con cierre remoto: Implementar un bucle de lectura bloqueante que termine cuando el servidor cierre la conexión, y por tanto cuando recv() devuelva una cadena vacía.

P3. Cliente

```
import jsockets, sys, time, random
from datetime import datetime
# A mi me funciona python3 new_client.py 127.0.0.1 1818
# Al probar con un host incorrecto se ven los intentos fallidos
# Parametros
M = 1   # Minutos entre mediciones
S = 10  # Segundos entre reintentos
R = 10  # Numero de reintentos
# Infomracion del paquete
COMPUTER_ID = "00000000-0000-0000-0000-000000000001"    # Key to DB
CPU_IDS = ["cpu0", "cpu1", "cpu2"]                  # Key to DB


def send_measurement(s, cpu_id):
  cpu_load = random.randint(0, 100)
  time_stamp = datetime.now().strftime("%Y%m%d%H%M%S") # Key to DB
  message = f"{COMPUTER_ID},{cpu_id},{cpu_load},{time_stamp}"
  print("CPU load:", cpu_load)
  print("Timestamp:", time_stamp)
  for attempt in range(R):
    try:
      s.send(message.encode())
      s.settimeout(S)
      data = s.recv(1024).decode()
      if data.startswith("ACK"):
        print(f"[PACKAGE RECEIVED] {data.strip()}")
        return
    except Exception as e:
      print(f"[RETRY {attempt+1}] No answer 'ACK' received from the server, retrying...")
  print(f"[FAILED] Measurement discarded after {R} retries: {message}")
```

```
if len(sys.argv) != 3:
   print("Use: " + sys.argv[0] + " host port")
   sys.exit(1)

s = jsockets.socket_udp_connect(sys.argv[1], sys.argv[2])
if s is None:
   print("could not open socket")
   sys.exit(1)

while True:
   for cpu_id in CPU_IDS:
      send_measurement(s, cpu_id)
   time.sleep(M * 60) # Esperar M minutos entre mediciones
# s.close() -> no se ejecutará ya que el bucle no termina
```

P3. Servidor

```
import jsockets, sys

# python3 new_server.py 1818


LOG_FILE = "cpu_log.txt"

RECEIVED_KEYS = set()


s = jsockets.socket_udp_bind(1818)

if s is None:

  print("Couldn't open socket.")

  sys.exit(1)

print("Server is active")


while True:

  try:

    data, addr = s.recvfrom(1024)

    if not data:

      continue

    decoded = data.decode().strip()

    parts = decoded.split(",")

    if len(parts) != 4:

      print("[ERROR] Packet has invalid construction:", decoded)

      continue


    computer_id, cpu_id, cpu_load, timestamp = parts

    key = f"{computer_id}_{cpu_id}_{timestamp}" # Keys to DB


    if key not in RECEIVED_KEYS:

      RECEIVED_KEYS.add(key)

      with open(LOG_FILE, "a") as f:

        f.write(f"{computer_id},{cpu_id},{cpu_load},{timestamp}\n")
```

```
        print(f"[RECEIVED] {decoded}")

        ack_msg = f"ACK,{timestamp},{computer_id},{cpu_id}" # ACK message from server to client
        print(f"[ACK] {ack_msg}")
        s.sendto(ack_msg.encode(), addr)
    except Exception as e:
        print("[ERROR]", e)
        continue
```

P4. Proxy

Dentro de los problemas principales encontrados están:

- **Una falta de conexión persistente.**
- **Falta de desarrollo en los sockets para determinar de donde viene el mensaje (cliente o servidor), actualmente se usa solo uno para cliente y servidor.**
- **Posible caída en un ciclo infinito de reenvío de respuestas al servidor por el punto anterior.**
- **No hay mecanismo para manejar múltiples clientes.**

```
import jsockets, threading


# Configuración del proxy
PROXY_PORT = 9999  # Puerto en que escucha el proxy
TARGET_HOST = "anakena.dcc.uchile.cl" # Direcci´on del servidor de destino
TARGET_PORT = 1818 # Puerto del servidor de destino


# Socket para recibir desde clientes
proxy_socket = jsockets.socket_udp_bind(PROXY_PORT)


# Socket para enviar/recibir del servidor
server_socket = jsockets.socket_udp_bind(0)
server_socket.settimeout(2)  # Prevenir un bloqueo por tiempo de espera


def handle_packet(data, client_address):
  try:
    # Enviar al servidor real
    server_socket.sendto(data, (TARGET_HOST, TARGET_PORT))


    # Recibir respuesta del servidor
    response, server_addr = server_socket.recvfrom(1024*1024)


    # Reenviar al cliente original
    proxy_socket.sendto(response, client_address)
```

```
    except Exception as e:
        print(f"Error con {client_address}: {e}")


def handle_clients():
    while True:
        try:
            data, client_address = proxy_socket.recvfrom(1024*1024)
            print(f"Paquete de {client_address}: {data.decode(errors='ignore')}")
            threading.Thread(target=handle_packet, args=(data, client_address), daemon=True).start()
        except Exception as e:
            print(f"Error al recibir datos: {e}")


# Hilo principal de escucha
threading.Thread(target=handle_clients, daemon=True).start()
print(f"Proxy UDP escuchando en puerto {PROXY_PORT}, redirigiendo a
{TARGET_HOST}:{TARGET_PORT}")
# Mantener el proxy en ejecuci´on
try:
    while True:
        pass
except KeyboardInterrupt:
    print("Proxy UDP detenido.")
    proxy_socket.close()
    server_socket.close()
```