# Vehículo Autónomo de Telemetría

Un protocolo de aplicación para la transmisión de datos de telemetría y control de un vehículo autónomo

Realizado por:

Valentina Castro Pineda

Juan Camilo Ramon Pérez

Isabella Idarraga Botero

Juan Jose Rodriguez Restrepo

Profesor:

Alber Oswaldo Montoya Benitez

Internet, Arquitectura y Protocolos

Universidad EAFIT

Octubre 2025

## Tabla de Contenidos

## Introducción

El presente proyecto constituye una experiencia formativa orientada al fortalecimiento de las competencias en diseño e implementación de protocolos de aplicación y en programación de sistemas de red concurrentes. A través de la simulación de un vehículo autónomo, se desarrolló un protocolo de telemetría que permite transmitir información en tiempo real y recibir comandos de control bajo un modelo cliente–servidor.

La práctica de implementar un protocolo propio, empleando la API de Sockets Berkeley en lenguaje C, ofrece a los estudiantes la posibilidad de comprender de manera integral cómo se estructura la comunicación entre entidades distribuidas. En este proceso se ponen en evidencia aspectos esenciales como la definición de mensajes claros y legibles, el uso de mecanismos de concurrencia para gestionar múltiples clientes, la emisión periódica de datos de telemetría, la validación de comandos bajo condiciones de seguridad y la importancia del registro de eventos (logging) para garantizar la trazabilidad del sistema.

## Marco Teórico

El diseño de un protocolo de aplicación para la telemetría de un vehículo autónomo requiere la comprensión de conceptos fundamentales en redes de computadores y en programación concurrente. En primer lugar, es necesario abordar el papel de los sockets, que constituyen la abstracción mediante la cual dos programas pueden comunicarse a través de una red. La API de Sockets Berkeley, empleada en este proyecto, se ha consolidado como un estándar para la implementación de aplicaciones de red y permite establecer tanto conexiones confiables mediante sockets de flujo (TCP), como comunicaciones rápidas y sin garantía de entrega mediante sockets de datagrama (UDP). En el caso particular de la telemetría, la elección de TCP resulta adecuada, ya que asegura que los mensajes lleguen completos, ordenados y sin pérdidas, condición esencial para transmitir variables críticas como la velocidad, la batería o la temperatura del vehículo.

Una vez definido el canal de comunicación, surge la necesidad de establecer un conjunto de reglas claras que guíen el intercambio de información. En este contexto, los protocolos de aplicación cumplen la función de especificar tanto el formato de los mensajes como los procedimientos para interpretarlos y darles respuesta. Protocolos ampliamente difundidos como HTTP o FTP evidencian la importancia de contar con estructuras de mensajes bien definidas y con códigos de respuesta estandarizados que garanticen la interoperabilidad entre distintas entidades. El protocolo propuesto en este proyecto adopta un enfoque textual, con comandos simples y legibles, inspirado en dichos modelos y documentado siguiendo la lógica de un RFC, lo que facilita su comprensión y su implementación.

Otro aspecto central es la concurrencia en servidores, indispensable para gestionar múltiples clientes de manera simultánea. Un servidor monolítico resultaría ineficiente al bloquearse en espera de cada interacción; por ello, se recurre a un esquema basado en hilos que permite atender cada conexión de forma independiente. De este modo, la difusión periódica de telemetría se mantiene ininterrumpida incluso si algún cliente envía comandos o si se producen errores en la comunicación. Este enfoque asegura un funcionamiento más robusto y escalable del sistema, al tiempo que refleja la relevancia de la concurrencia en el diseño de arquitecturas de red modernas.

Finalmente, es pertinente situar este tipo de desarrollos dentro de un marco más amplio. En repositorios de código abierto, como los que se encuentran en GitHub, se observan proyectos similares orientados a la telemetría de drones, robots móviles o vehículos simulados. Dichos sistemas suelen emplear arquitecturas cliente–servidor para difundir datos en tiempo real y habilitar comandos de control remoto, en algunos casos apoyándose en protocolos estandarizados como MQTT o CoAP. No obstante, la construcción de un protocolo propio en el marco académico constituye un ejercicio enriquecedor, ya que permite experimentar directamente con las decisiones de diseño,

los mecanismos de autenticación, el manejo de errores y las estrategias de comunicación concurrente, consolidando así un entendimiento integral del papel que desempeñan los protocolos en la coordinación de sistemas distribuidos.

**Visión General del Protocolo**

El protocolo diseñado establece las reglas de comunicación entre un servidor central y múltiples clientes conectados de manera concurrente. Su propósito principal es difundir periódicamente información de telemetría de un vehículo autónomo simulado y permitir que, bajo condiciones de autenticación, un usuario con privilegios de administrador emita comandos de control que modifiquen el estado del vehículo.

La arquitectura adoptada se basa en un modelo cliente–servidor sobre la pila TCP/IP, haciendo uso de sockets de flujo (SOCK_STREAM). El servidor, implementado en lenguaje C, gestiona múltiples conexiones mediante hilos independientes y mantiene una lista actualizada de los clientes activos. Un hilo adicional se encarga de difundir cada diez segundos un mensaje de telemetría con los valores de velocidad, nivel de batería, temperatura y orientación del vehículo.

El sistema define dos roles claramente diferenciados:

- Observador (VIEWER): asignado por defecto a toda conexión inicial. Este rol permite recibir de forma pasiva los mensajes de telemetría, sin capacidad de enviar comandos de control.

- Administrador (ADMIN): obtenido tras un proceso de autenticación exitoso mediante el comando AUTH <usuario> <contraseña>. Una vez concedido, este rol habilita el envío de comandos de control (CMD …) y la consulta de usuarios conectados (USERS).

Cada interacción entre cliente y servidor se lleva a cabo a través de mensajes de texto estructurados en líneas individuales, siguiendo un formato definido y con respuestas que pueden ser de aceptación (OK, ACK) o de error (ERROR, NACK). Asimismo, cada petición y respuesta se registra en un mecanismo de logging que incluye dirección IP, puerto y marca de tiempo, lo que garantiza trazabilidad y soporte para depuración y validación del sistema.

De este modo, el protocolo no sólo especifica un conjunto de comandos y respuestas, sino que también establece un marco de funcionamiento en el que se combinan la concurrencia, la autenticación básica y la difusión periódica de datos, brindando una solución coherente para la supervisión y el control remoto de un vehículo autónomo en un entorno simulado.

**Especificación del servicio**

El protocolo de telemetría y control diseñado se fundamenta en un intercambio de mensajes de texto estructurados, enviados línea por línea, donde cada operación se expresa mediante un comando legible y su respectiva respuesta. Los servicios definidos permiten tanto la identificación de clientes y la autenticación de administradores, como la difusión periódica de telemetría y la emisión de comandos de control.

A continuación, se describen las operaciones principales:

1. HELLO <name>

   Permite que el cliente se identifique con un nombre amigable.

Respuesta del servidor: OK hello <name>

```
┌──(kali㉿kali)-[~]
└─$ nc -v 192.168.78.123 9000
Connection to 192.168.78.123 9000 port [tcp/*] succeeded!
WELCOME TelemetryServer PROTO 1.0
ROLE VIEWER
HELLO VALE
OK hello VALE
```

2. AUTH <user> <pass>

Eleva la sesión al rol de administrador si las credenciales son correctas.

Respuesta válida: Ok auth

```
AUTH admin 1234
ROLE ADMIN
OK auth
```

Error: ERROR 401 invalid_credentials en caso de credenciales inválidas.

```
ERROR 401 invalid_credentials
```

3. CMD <SPEED_UP | SLOW_DOWN | TURN_LEFT | TURN_RIGHT> (solo ADMIN)

Solicita ejecutar una acción de control sobre el vehículo.

Respuesta positiva: ACK <CMD> accepted.

```
CMD SPEED_UP
ACK SPEED_UP accepted
```

Respuesta negativa: NACK <reason> donde la razón puede ser low_battery, speed_limit, min_speed o unknown_cmd.

```
CMD HOLA
NACK unknown_cmd
```

Error: ERROR 403 not_admin si el cliente no tiene privilegios, o ERROR 400 invalid_cmd si el comando no es válido.

```
ROLE VIEWER
CMD SPEED_UP
ERROR 403 not_admin
```

4. DATA speed=<kmh> battery=<pct> temp=<celsius> heading=<deg> ts=<ms_epoch>

Mensaje emitido automáticamente por el servidor cada 10 segundos a todos los clientes conectados. Incluye velocidad (km/h), porcentaje de batería, temperatura (°C), rumbo (0–359 grados) y una marca de tiempo en milisegundos desde época Unix.

```
DATA speed=63.0 battery=85.8 temp=36.6 heading=56.0 ts=1759537705979
```

5. USERS (solo ADMIN)

Solicita la lista de clientes conectados. El servidor responde con una línea por cada cliente, incluyendo índice, dirección IP, puerto, rol y nombre.

Ejemplo de respuesta:

```
USER 0 ip=192.168.78.123 port=64631 role=ADMIN name=anon
```

6. BYE

Finaliza la sesión.

Respuesta del servidor: OK bye. La conexión se cierra inmediatamente.

```
BYE
OK bye
```

En cuanto al manejo de errores, el protocolo contempla:

- ➔ ERROR 400: errores de sintaxis o validación.
- ➔ ERROR 401: autenticación fallida.
- ➔ ERROR 403: operación no autorizada.
- ➔ ERROR 501: operación no implementada.

Con esta especificación, el servicio asegura una comunicación clara y estructurada, diferenciando las capacidades de los roles (VIEWER y ADMIN) y garantizando un control consistente del flujo de mensajes entre clientes y servidor.

**Formato de mensajes**

| Mensaje | Dirección | Campos | Ejemplo |
|---|---|---|---|
| HELLO | Cliente → Servidor | name: identificador de cliente | HELLO Juan |
| WELCOME | Servidor → Cliente | nombre de servidor, versión de protocolo | WELCOME TelemetryServer PROTO 1.0 |
| ROLE | Servidor → Cliente | VIEWER o ADMIN según rol asignado | ROLE VIEWER |
| OK hello | Servidor → Cliente | confirmación de identificación | OK hello Juan |
| AUTH | Cliente → Servidor | user, pass | AUTH admin 1234 |
| ROLE ADMIN | Servidor → Cliente | rol elevado tras autenticación | ROLE ADMIN |
| CMD | Cliente → Servidor | acción: SPEED_UP, SLOW_DOWN, TURN_LEFT, TURN_RIGHT | CMD SPEED_UP |
| ACK | Servidor → Cliente | confirmación de acción aceptada | ACK SPEED_UP accepted |
| NACK | Servidor → Cliente | rechazo con motivo | NACK low_battery |
| DATA | Servidor → Cliente | velocidad, batería, temperatura, rumbo, timestamp | DATA speed=52.3 battery=90.0 temp=35.0 heading=175 ts=1737582012345 |
| USERS | Cliente → Servidor (ADMIN) | solicitud de lista de usuarios conectados | USERS |

| | | | |
|---|---|---|---|
| USER | Servidor → Cliente (ADMIN) | índice, ip, puerto, rol, nombre | USER 0 ip=127.0.0.1 port=5000 role=VIEWER name=anon |
| BYE | Cliente → Servidor | cierre de sesión | BYE |
| OK bye | Servidor → Cliente | confirmación de desconexión | OK bye |

**Implementación**

La implementación del protocolo se llevó a cabo mediante un servidor TCP programado en lenguaje C utilizando la API de Sockets Berkeley en el entorno de Windows (Winsock2). El servidor fue diseñado para atender múltiples clientes de manera concurrente y difundir periódicamente datos de telemetría, integrando además un mecanismo de autenticación básica para el rol de administrador.

El proceso de implementación puede dividirse en varios componentes principales:

1. Inicialización del servidor

El programa recibe como parámetros el puerto de escucha y, de manera opcional, la ruta de un archivo de logs. Tras validar la configuración, se inicializa la biblioteca Winsock, se crea un socket de tipo flujo (TCP) y se asocia a la dirección local mediante bind. Posteriormente, se coloca en modo de escucha con la llamada listen, quedando preparado para aceptar conexiones entrantes.

2. Manejo de clientes concurrentes

Cada vez que un cliente se conecta, el servidor crea una estructura de datos con la información básica de la sesión (socket, IP, puerto, rol y nombre). Esta información se almacena en una lista enlazada global protegida por secciones críticas para garantizar consistencia en entornos concurrentes. A continuación, se crea un hilo independiente encargado de gestionar la comunicación con ese cliente.

3. Hilo de telemetría

De manera paralela, un hilo global ejecuta la difusión periódica de telemetría cada 10 segundos. El estado del vehículo, representado en una estructura de datos, se actualiza de forma aleatoria dentro de valores realistas (velocidad, batería, temperatura y rumbo). Dichos valores se formatean en un mensaje DATA … que se envía a todos los clientes activos mediante un mecanismo de broadcast.

4. Procesamiento de comandos

El hilo asociado a cada cliente se encarga de leer mensajes línea por línea y procesar los comandos reconocidos por el protocolo. Entre las operaciones soportadas se encuentran:

- HELLO: asigna un nombre al cliente.
- AUTH: valida credenciales y eleva el rol a administrador.

- CMD: interpreta y ejecuta un comando de control si el rol es ADMIN.
- USERS: envía al administrador la lista de usuarios conectados.
- BYE: finaliza la sesión y cierra la conexión.

En el caso de comandos inválidos o sin privilegios, el servidor responde con los códigos de error definidos en la especificación.

5. Logging y trazabilidad

Cada interacción, tanto de entrada (RX) como de salida (TX), se registra en consola y en un archivo de log. Los registros incluyen etiqueta, marca de tiempo en milisegundos, dirección IP y puerto del cliente, dirección del flujo (RX/TX) y contenido del mensaje. Esto permite auditar las comunicaciones y facilitar la depuración del sistema.

6. Cierre y liberación de recursos

Cuando un cliente finaliza la sesión o se produce un error en la comunicación, el servidor elimina la referencia de la lista global y libera los recursos asociados (socket y memoria). Al finalizar la ejecución, también se cierran los hilos de telemetría, se destruyen las secciones críticas, se cierra el archivo de log y se liberan los recursos de Winsock.

**Cliente Python (Tkinter) y Cliente Java (Swing)**

**Formato de Mensajes**

**1. Conexión Inicial**

**Cliente Python:**

[Usuario en GUI]
- Ingresa Host: "127.0.0.1"
- Ingresa Port: "9000"
- Clic en botón "Conectar"

[Cliente → Servidor]
<Conexión TCP establecida en 127.0.0.1:9000>

[Servidor → Cliente]
<< WELCOME TelemetryServer PROTO 1.0
<< ROLE VIEWER

[Interfaz Gráfica Python - Tkinter]
- Estado: "Conectado a 127.0.0.1:9000" (color verde)
- Role: "VIEWER" (color azul)
- Log: "[20:42:56] Conectando a 127.0.0.1:9000..."
  "[20:42:56] Conectado exitosamente"
  "[20:42:56] << WELCOME TelemetryServer PROTO 1.0"
  "[20:42:56] << ROLE VIEWER"

**Cliente Java:**

[Usuario en GUI]
- Ingresa Host: "127.0.0.1"
- Ingresa Port: "9000"
- Clic en botón "Conectar"

```
[Cliente → Servidor]
<Conexión TCP establecida en 127.0.0.1:9000>

[Servidor → Cliente]
<< WELCOME TelemetryServer PROTO 1.0
<< ROLE VIEWER

[Interfaz Gráfica Java - Swing]
- lblStatus: "Conectado a 127.0.0.1:9000"
- lblRole: "Role: VIEWER"
- btnConnect texto cambia a: "Desconectar"
- Log: "Conectando a 127.0.0.1:9000 ..."
      "Conectado."
      "[WELCOME] WELCOME TelemetryServer PROTO 1.0"
      "<< ROLE VIEWER"
```

Comparación:

- Ambos clientes usan el mismo flujo de conexión TCP.
- Python usa Tkinter, Java usa Swing (frameworks GUI nativos).
- Python actualiza colores de labels, Java cambia texto de botones.
- Formato de log similar con pequeñas diferencias de estilo.

**2. Identificación con HELLO**

**Cliente Python:**

```
[Usuario en GUI]
- Clic en botón "HELLO"
- Diálogo emergente solicita nombre
- Ingresa: "Isabella"

[Cliente → Servidor]
>> HELLO Isabella

[Servidor → Cliente]
<< OK hello Isabella

[Log Python]
"[20:43:10] >> HELLO Isabella"
"[20:43:10] << OK hello Isabella"
```

**Cliente Java:**

```
[Usuario en GUI]
- Clic en botón "HELLO"
- JOptionPane solicita nombre (showInputDialog)
- Ingresa: "Juan"

[Cliente → Servidor]
>> HELLO Juan

[Servidor → Cliente]
<< OK hello Juan

[Log Java]
">> HELLO Juan"
"<< OK hello Juan"
```

**Comparación:**

- Python usa tkinter.simpledialog.askstring().

- Java usa JOptionPane.showInputDialog().
- Ambos validan que el nombre no esté vacío.
- Python incluye timestamps en log, Java no.

### 3. Autenticación como Administrador

**Cliente Python:**

[Usuario en GUI]
- Clic en botón "AUTH"
- Diálogo solicita usuario y contraseña
- Usuario: "admin"
- Contraseña: "1234" (campo oculto con asteriscos)

[Cliente → Servidor]
>> AUTH admin 1234

[Servidor → Cliente]
<< ROLE ADMIN
<< OK auth

[Interfaz Python]
- Role label actualiza a: "ADMIN" (color naranja)
- Botones de comando (SPEED UP, SLOW DOWN, etc.) se habilitan
- Log: "[20:43:25] >> AUTH admin 1234"
  "[20:43:25] << ROLE ADMIN"
  "[20:43:25] << OK auth"

**Cliente Java:**

[Usuario en GUI]
- Clic en botón "AUTH"
- JOptionPane con panel personalizado (2x2 grid)
- Campo user: JTextField (pre-llenado con "admin")
- Campo pass: JPasswordField (oculto)

[Cliente → Servidor]
>> AUTH admin 1234

[Servidor → Cliente]
<< ROLE ADMIN
<< OK auth

```
[Interfaz Java]
- lblRole actualiza a: "Role: ADMIN"
- Log: ">> AUTH admin 1234"
      "<< ROLE ADMIN"
      "<< OK auth"
```

**Comparación:**

- Python cambia colores de labels según rol (azul=VIEWER, naranja=ADMIN)
- Java solo actualiza texto del label
- Python habilita/deshabilita botones según rol
- Java permite enviar comandos siempre (validación en servidor)
- Ambos usan campos de contraseña ocultos (Entry show="*" / JPasswordField)

**4. Recepción de Telemetría Automática**

**Cliente Python:**

[Servidor → Todos los Clientes] (Broadcast cada 10 segundos)
<< DATA speed=52.3 battery=89 temp=36.2 heading=175 ts=1759296717586

[ReaderThread Python]
- socket.recv() recibe mensaje
- parse_data() extrae valores con expresiones regulares
- root.after() agenda actualización en MainThread (thread-safe)

[Panel de Telemetría Python - Tkinter]

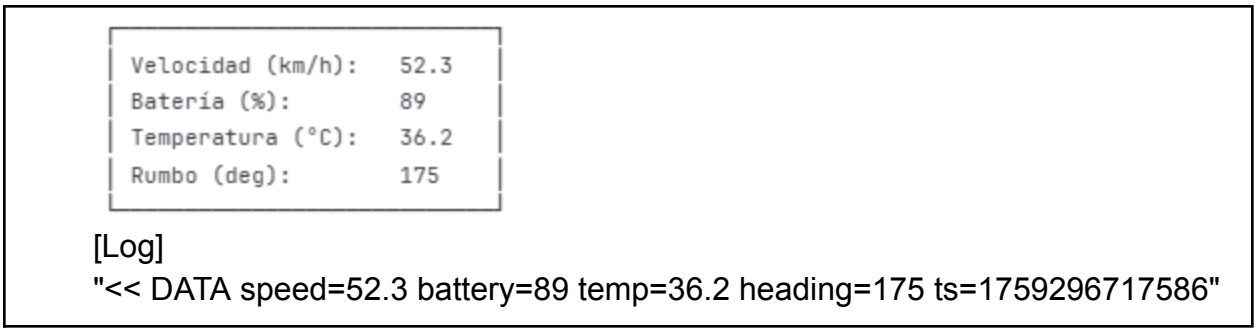

[Log]
"[20:43:37] << DATA speed=52.3 battery=89 temp=36.2 heading=175 ts=1759296717586"

Cliente Java:

[Servidor → Todos los Clientes] (Broadcast cada 10 segundos)
<< DATA speed=52.3 battery=89 temp=36.2 heading=175 ts=1759296717586

[ReaderThread Java]
- reader.readLine() recibe mensaje
- parseData() extrae valores con split() y HashMap
- SwingUtilities.invokeLater() actualiza GUI en EDT (thread-safe)

[Panel de Telemetría Java - Swing]




**Comparación**:

- Python usa root.after() para thread-safety con Tkinter.
- Java usa SwingUtilities.invokeLater() para thread-safety con Swing.
- Python parsea con expresiones regulares (re.findall).
- Java parsea con split() y HashMap<String,String>.
- Ambos actualizan labels de telemetría en tiempo real.
- Layout visual prácticamente idéntico.

**5. Envío de Comandos de Control (ADMIN)**

**Cliente Python:**

```
[Usuario en GUI - Rol ADMIN]
- Clic en botón "SPEED UP" (botón dedicado)

[Cliente → Servidor]
>> CMD SPEED_UP

[Servidor → Cliente]
<< ACK SPEED_UP accepted

[Log Python]
"[20:44:15] >> CMD SPEED_UP"
"[20:44:15] << ACK SPEED_UP accepted"

[Próximo DATA recibido - 10s después]
<< DATA speed=57.3 battery=89 temp=36.2 heading=175 ts=1759296727586
(Velocidad aumentó de 52.3 a 57.3)
```

**Cliente Java:**

```
[Usuario en GUI]
- Escribe en cmdField: "CMD SPEED_UP"
- Presiona Enter o clic en botón "Enviar"

[Cliente → Servidor]
>> CMD SPEED_UP

[Servidor → Cliente]
<< ACK SPEED_UP accepted

[Log Java]
">> CMD SPEED_UP"
"[ACK] ACK SPEED_UP accepted"

[Próximo DATA recibido]
<< DATA speed=57.3 battery=89 temp=36.2 heading=175 ts=1759296727586
```

**Comparación:**

- Python: 4 botones dedicados (SPEED UP, SLOW DOWN, TURN LEFT, TURN RIGHT).
- Java: campo de texto libre para cualquier comando.
- Python valida rol antes de enviar (botones deshabilitados para VIEWER).
- Java envía siempre, servidor valida y rechaza si no es ADMIN.
- Ambos reciben ACK del servidor.
- Ambos ven el cambio reflejado en el siguiente DATA.

**6. Comando Rechazado (VIEWER intenta comando)**

**Cliente Python:**

```
[Usuario en GUI - Rol VIEWER]
- Botones de comando están deshabilitados (grisados)
- No puede hacer clic en "SPEED UP"

Si se fuerza envío manualmente:
[Cliente → Servidor]
>> CMD SPEED_UP

[Servidor → Cliente]
<< ERROR 403 not_admin

[Interfaz Python]
- Log: "[20:45:00] [ERROR] ERROR 403 not_admin"
- Popup: messagebox.showwarning("No está conectado como administrador")
```

**Cliente Java:**

```
[Usuario en GUI - Rol VIEWER]
- Escribe en cmdField: "CMD SPEED_UP"
- Presiona Enter (no hay validación en cliente)

[Cliente → Servidor]
>> CMD SPEED_UP

[Servidor → Cliente]
<< ERROR 403 not_admin

[Log Java]
">> CMD SPEED_UP"
"[ERROR] ERROR 403 not_admin"
```

**Comparación:**

- Python: validación doble (cliente + servidor).
- Java: validación solo en servidor.
- Python muestra popup de advertencia al usuario.

- Java solo registra error en log.
- Enfoque Python: previene errores antes de enviar.
- Enfoque Java: permite flexibilidad, servidor decide.

**7. Consulta de Usuarios Conectados (ADMIN)**

**Cliente Python:**

```
[Usuario en GUI - Rol ADMIN]
- Clic en botón "USERS" (botón dedicado)

[Cliente → Servidor]
>> USERS

[Servidor → Cliente]
<< USERS count=2
<< USER 0 ip=127.0.0.1 port=55934 role=ADMIN name=Isabella
<< USER 1 ip=127.0.0.1 port=55935 role=VIEWER name=Juan
<< OK users

[Log Python]
"[20:46:10] >> USERS"
"[20:46:10] [USERS] USERS count=2"
"[20:46:10] [USER] USER 0 ip=127.0.0.1 port=55934 role=ADMIN name=Isabella"
"[20:46:10] [USER] USER 1 ip=127.0.0.1 port=55935 role=VIEWER name=Juan"
"[20:46:10] [USERS END] OK users"
```

**Cliente Java:**

```
[Usuario en GUI]
- Escribe en cmdField: "USERS"
- Presiona Enter

[Cliente → Servidor]
>> USERS

[Servidor → Cliente]
<< USERS count=2
<< USER 0 ip=127.0.0.1 port=49157 role=ADMIN name=Juan
<< USER 1 ip=127.0.0.1 port=49158 role=VIEWER name=Isabella
<< OK users
```

```
[Log Java]
">> USERS"
"[USERS] USERS count=2"
"[USER] USER 0 ip=127.0.0.1 port=49157 role=ADMIN name=Juan"
"[USER] USER 1 ip=127.0.0.1 port=49158 role=VIEWER name=Isabella"
"[USERS END] OK users"
```

**Comparación:**

- Python: botón dedicado "USERS".
- Java: comando manual en campo de texto.
- Ambos marcan mensajes USER con etiquetas especiales en log.
- Ambos usan flag expectingUsers para parseo multi-línea.
- Python: variable self.expecting_users.
- Java: variable expectingUsers.
- Formato de log idéntico en ambos clientes.

**8. Cierre de Conexión**

**Cliente Python:**

```
[Usuario en GUI]
- Clic en botón "BYE"

[Cliente → Servidor]
>> BYE

[Servidor → Cliente]
<< OK bye
<Conexión TCP cerrada>

[Interfaz Python]
- Estado: "Desconectado" (color rojo)
- Role: "Role: VIEWER" (reseteo a default)
- Botón "Conectar" habilitado nuevamente
- Botones de comando deshabilitados
- Log: "[20:47:30] >> BYE"
      "[20:47:30] << OK bye"
      "[20:47:30] Cerrando conexión..."
```

**Cliente Java:**

```
[Usuario en GUI]
- Clic en botón "BYE"

[Cliente → Servidor]
>> BYE

[Thread.sleep(150)]  // Espera respuesta del servidor

[Servidor → Cliente]
<< OK bye

[Interfaz Java]
- lblStatus: "Desconectado"
- lblRole: "Role: VIEWER" (reseteo)
```

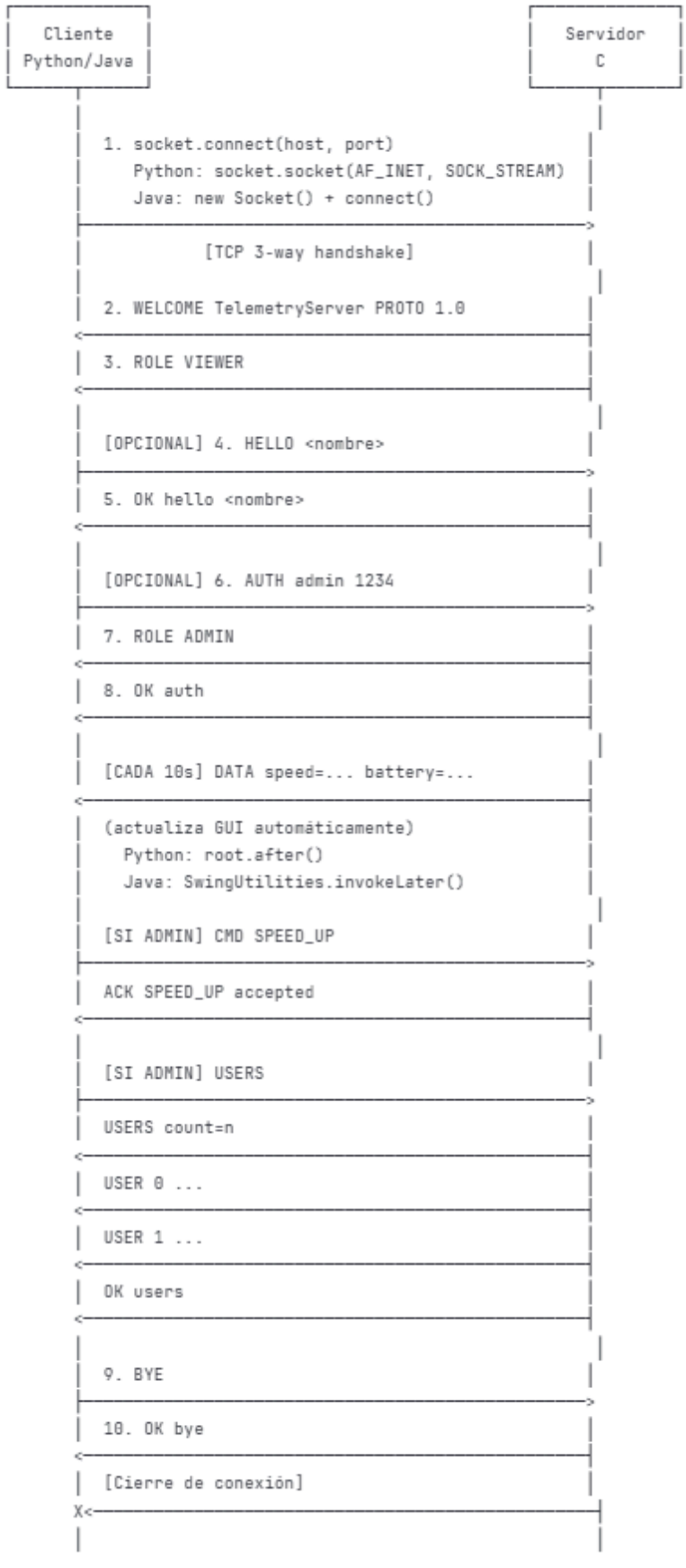
- btnConnect texto cambia a: "Conectar"
- socket.close()
- Log: ">> BYE"
  "<< OK bye"
  "Cerrando conexión..."

**Comparación:**

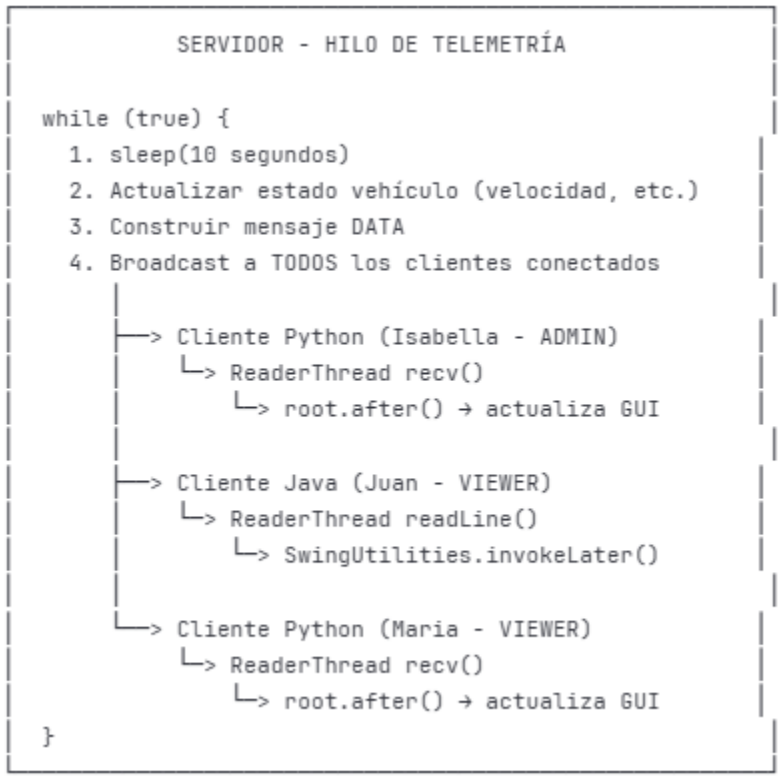
- Python: cierra socket inmediatamente después de enviar BYE
- Java: espera 150ms antes de cerrar (da tiempo al servidor)
- Ambos resetean rol a VIEWER
- Ambos actualizan estado visual de conexión
- Ambos permiten reconexión sin reiniciar aplicación

## Reglas de Procedimiento

## Diagrama de Flujo de Conexión (Secuencia)



**Difusión Periódica de Telemetría**




**Envío y Validación de Comandos**

**Cliente Python:**

Usuario hace clic en botón "SPEED UP"
↓
Verificar self.role == "ADMIN"
↓
Si no es ADMIN:
- Mostrar messagebox.showwarning()
- NO enviar comando
↓
Si es ADMIN:
- send_raw("CMD SPEED_UP\r\n")
- Esperar respuesta: ACK / NACK / ERROR

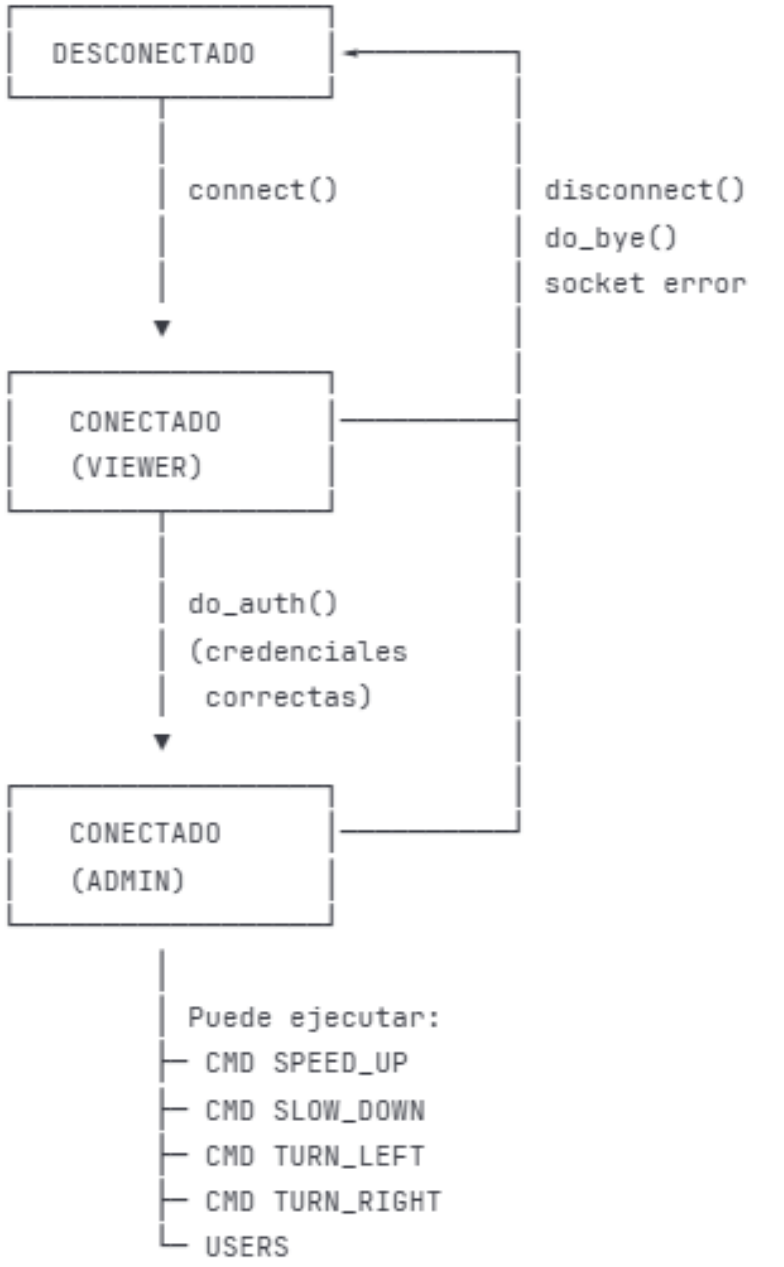
**Cliente Java:**

```
Usuario escribe "CMD SPEED_UP" y presiona Enter
  ↓
sendRaw("CMD SPEED_UP\r\n")
  (NO validación de rol en cliente)
  ↓
Servidor valida rol
  ↓
Si no es ADMIN:
  - Servidor envía: ERROR 403 not_admin
  - Cliente recibe y registra en log
  ↓
Si es ADMIN:
  - Servidor valida condiciones operacionales (batería, estado del vehículo)
  - Si OK: envía ACK SPEED_UP accepted
  - Si NO: envía NACK low_battery (u otro motivo)
```

**Comparación de Enfoques:**

- Python: validación preventiva en cliente + validación en servidor
- Java: validación solo en servidor (más flexible)
- Python: mejor UX (usuario no ve errores evitables)
- Java: más simple en código cliente

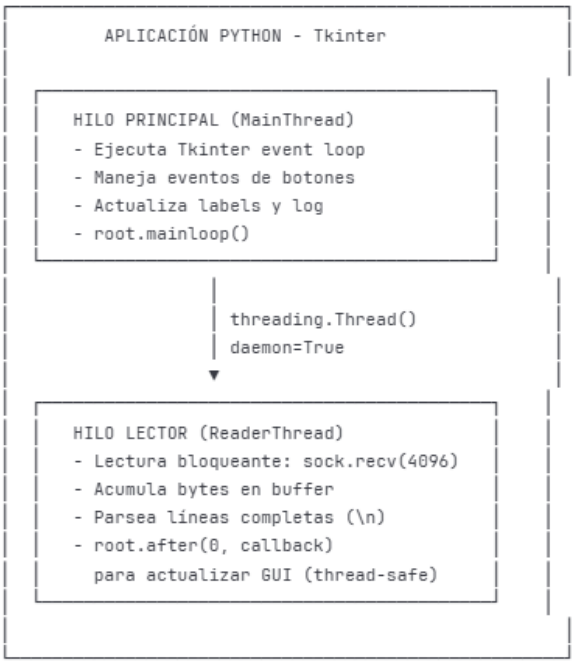
**Diagrama de Estados del Cliente**



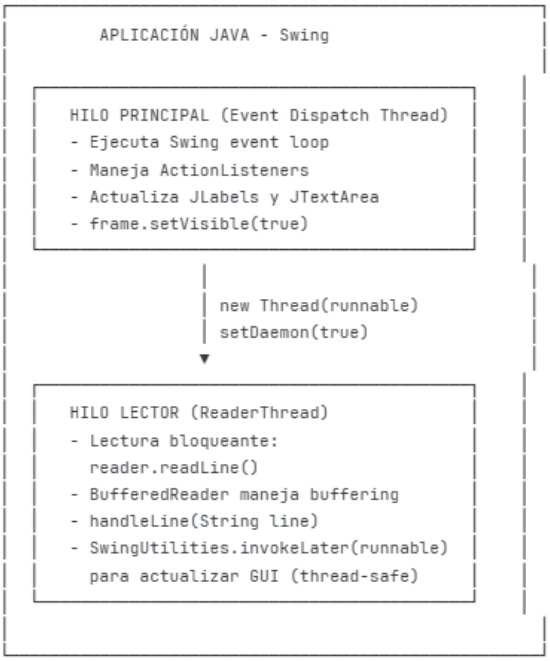

Estado aplica a ambos clientes (Python y Java)

## Arquitectura de Hilos en los Clientes

### Cliente Python:




### Cliente Java:



**Comparación de Concurrencia:**

- Ambos usan 2 hilos: MainThread (GUI) + ReaderThread (I/O).
- Python: root.after() para thread-safety.
- Java: SwingUtilities.invokeLater() para thread-safety.
- Python: sock.recv() devuelve bytes, manual buffering.
- Java: BufferedReader.readLine() maneja buffering automático.
- Ambos marcan ReaderThread como daemon (termina con main).

**Pruebas y validación**

Se realizaron 12 casos de pruebas exhaustivas probando ambos clientes (Python y Java) contra el mismo servidor C. A continuación se detallan los resultados.

**1. Conexión Básica**

Objetivo: Verificar que ambos clientes pueden conectarse al servidor
Procedimiento:

1. Iniciar servidor: ./server_new.exe 9000 logs/server.log
2. Cliente Python: python telemetry_client.py
3. Cliente Java: java TelemetryClient 127.0.0.1 9000
4. Conectar ambos clientes simultáneamente

Resultado: EXITOSO

**Cliente Python:**

```
[20:42:56] Conectando a 127.0.0.1:9000...
[20:42:56] Conectado exitosamente
[20:42:56] << WELCOME TelemetryServer PROTO 1.0
[20:42:56] << ROLE VIEWER
```

**Cliente Java:**

```
Conectando a 127.0.0.1:9000 ...
Conectado.
[WELCOME] WELCOME TelemetryServer PROTO 1.0
<< ROLE VIEWER
```

**Servidor log:**

```
[2025-10-04 20:42:56] Cliente conectado desde 127.0.0.1:55934
[2025-10-04 20:42:56] [TX] WELCOME TelemetryServer PROTO 1.0
[2025-10-04 20:42:56] [TX] ROLE VIEWER
[2025-10-04 20:42:57] Cliente conectado desde 127.0.0.1:55935
[2025-10-04 20:42:57] [TX] WELCOME TelemetryServer PROTO 1.0
[2025-10-04 20:42:57] [TX] ROLE VIEWER
```

**2. Recepción de Telemetría en Tiempo Real**

Objetivo: Verificar que ambos clientes reciben DATA cada 10s

Procedimiento:

1. Conectar ambos clientes
2. Observar panel de telemetría durante 1 minuto
3. Verificar actualización automática

Resultado: EXITOSO

Ambos clientes actualizaron sus paneles de telemetría cada 10 segundos exactos. Valores recibidos idénticos en ambos clientes (broadcast del servidor).

Python:

```
[20:43:07] << DATA speed=50.0 battery=98.6 temp=34.8 heading=103.0 ts=...
[20:43:17] << DATA speed=50.5 battery=98.4 temp=34.8 heading=103.5 ts=...
[20:43:27] << DATA speed=51.0 battery=98.2 temp=35.4 heading=104.0 ts=...
```

Java:

```
<< DATA speed=50.0 battery=98.6 temp=34.8 heading=103.0 ts=...
<< DATA speed=50.5 battery=98.4 temp=34.8 heading=103.5 ts=...
<< DATA speed=51.0 battery=98.2 temp=35.4 heading=104.0 ts=...
```

**3. Autenticación Exitosa (ADMIN)**

Objetivo: Verificar que AUTH eleva privilegios en ambos clientes

Procedimiento:

1. Cliente Python: clic en AUTH, user=admin, pass=1234
2. Cliente Java: clic en AUTH, user=admin, pass=1234

Resultado: EXITOSO

Cliente Python:

```
[20:44:43] >> AUTH admin 1234
[20:44:43] << ROLE ADMIN
[20:44:43] << OK auth
```

Role label cambió a naranja "ADMIN". Botones de comando habilitados.

Cliente Java:

```
>> AUTH admin 1234
<< ROLE ADMIN
<< OK auth
```

lblRole actualizado a "Role: ADMIN"

**4. Autenticación Fallida**

Objetivo: Verificar manejo de credenciales incorrectas
Procedimiento:

1. Intentar AUTH con password incorrecta "123"

Resultado: EXITOSO
Ambos clientes:

```
>> AUTH admin 123
[ERROR] ERROR 401 invalid_credentials
```

Role permaneció como VIEWER en ambos

**5. Comando de Control SPEED_UP (ADMIN)**

Objetivo: Verificar que comandos son ejecutados
Procedimiento:

1. Cliente Python (ADMIN): clic en botón "SPEED UP"
2. Cliente Java (ADMIN): escribir "CMD SPEED_UP" + Enter

Resultado: EXITOSO

Cliente Python:

```
[20:46:47] >> CMD SPEED_UP
[20:46:47] << ACK SPEED_UP accepted
[20:46:55] << DATA speed=57.0 ... (velocidad aumentó)
```

Cliente Java:

```
>> CMD SPEED_UP
```

```
[ACK] ACK SPEED_UP accepted
<< DATA speed=57.0 ... (velocidad aumentó)
```

Ambos clientes vieron el incremento de velocidad en el siguiente DATA

**6. Comando Rechazado (VIEWER intenta comando)**

Objetivo: Verificar restricción de comandos para VIEWER
Procedimiento:

1. Cliente Python (VIEWER): botones deshabilitados, no puede enviar
2. Cliente Java (VIEWER): escribir "CMD SPEED_UP" + Enter

Resultado: EXITOSO
Cliente Python: botones deshabilitados, no permite envío
Cliente Java:

```
>> CMD SPEED_UP
[ERROR] ERROR 403 not_admin
```

Comparación: Python previene en cliente, Java valida en servidor (ambos válidos).

**7. Múltiples Comandos Secuenciales**

Objetivo: Verificar procesamiento en secuencia
Procedimiento (Cliente Java ADMIN):

1. CMD SPEED_UP (3 veces)
2. CMD TURN_LEFT
3. CMD SPEED_UP (2 veces más)

Resultado: EXITOSO

```
>> CMD SPEED_UP
[ACK] ACK SPEED_UP accepted
>> CMD SPEED_UP
[ACK] ACK SPEED_UP accepted
>> CMD SPEED_UP
[ACK] ACK SPEED_UP accepted
<< DATA speed=65.0 ... (aumento de 15 km/h)
>> CMD TURN_LEFT
[ACK] ACK TURN_LEFT accepted
<< DATA speed=65.0 ... heading=90 (giro -15 grados)
```

**8. Consulta USERS (Múltiples Clientes)**

Objetivo: Verificar listado de usuarios conectados
Procedimiento:

1. Cliente Python (Isabella - ADMIN)
2. Cliente Java (Juan - VIEWER)
3. Cliente Python envía USERS

Resultado: EXITOSO
Cliente Python log:

```
[20:50:24] >> USERS
[20:50:24] [USERS] USERS count=2
[20:50:24] [USER] USER 0 ip=127.0.0.1 port=55934 role=ADMIN name=Isabella
[20:50:24] [USER] USER 1 ip=127.0.0.1 port=55935 role=VIEWER name=Juan
[20:50:24] [USERS END] OK users
```

**9. Desconexión y Reconexión**

Objetivo: Verificar cierre/reapertura
Procedimiento:

1. Cliente Python: clic "BYE"
2. Esperar 5s
3. Reconectar

Resultado: EXITOSO

```
[20:51:10] >> BYE
[20:51:10] << OK bye
[20:51:10] Cerrando conexión...
[20:51:20] Conectando a 127.0.0.1:9000...
[20:51:20] Conectado exitosamente
[20:51:20] << WELCOME TelemetryServer PROTO 1.0
[20:51:20] << ROLE VIEWER
```

Role reseteado a VIEWER correctamente

**10. Desconexión Inesperada del Servidor**

Objetivo: Verificar manejo de errores de conexión
Procedimiento:

1. Conectar ambos clientes
2. Cerrar servidor abruptamente (Ctrl+C)
3. Observar comportamiento

Resultado: EXITOSO

Cliente Python:

```
[20:52:15] Conexión cerrada por el servidor
[20:52:15] Error en lectura: [Errno 10054] An existing connection was forcibly closed
```

Estado cambió a "Desconectado", no hubo crash

Cliente Java:

```
Conexión cerrada por error: Connection reset
```

lblStatus actualizado a "Desconectado", no hubo crash

**11. Interoperabilidad Python-Java**

Objetivo: Verificar que ambos clientes funcionan simultáneamente

Procedimiento:

1. Cliente Python (Isabella - ADMIN) envía CMD SPEED_UP
2. Ambos clientes (Python y Java) deben recibir el DATA actualizado

Resultado: EXITOSO

Servidor log:

```
[20:53:45] [RX Cliente 0] CMD SPEED_UP
[20:53:45] [TX Cliente 0] ACK SPEED_UP accepted
[20:53:50] [BROADCAST] DATA speed=62.0 battery=95.2 temp=35.8 heading=110 ts=...
[20:53:50] [TX Cliente 0] DATA speed=62.0 ...
[20:53:50] [TX Cliente 1] DATA speed=62.0 ...
```

Ambos clientes actualizaron velocidad a 62.0 simultáneamente

**12. Comando con Sintaxis Incorrecta**

Objetivo: Verificar validación de formato
Procedimiento (Cliente Java):

1. Escribir: "CMD SPEED_DOWN" (comando inválido)

Resultado: EXITOSO

```
>> CMD SPEED_DOWN
[NACK] NACK unknown_cmd
```

**Resultados de Pruebas con Wireshark**

Se capturó tráfico TCP entre clientes y servidor:

```
Frame 1: Cliente Python → Servidor
  TCP SYN, Port 55934 → 9000

Frame 2: Servidor → Cliente Python
  TCP SYN-ACK, Port 9000 → 55934

Frame 3: Cliente Python → Servidor
  TCP ACK

Frame 4: Servidor → Cliente Python
  [PSH, ACK] Payload: "WELCOME TelemetryServer PROTO 1.0\n"

Frame 5: Servidor → Cliente Python
  [PSH, ACK] Payload: "ROLE VIEWER\n"

Frame 10: Cliente Java → Servidor
  TCP SYN, Port 55935 → 9000

Frame 11: Servidor → Cliente Java
  TCP SYN-ACK, Port 9000 → 55935

Frame 12: Cliente Java → Servidor
  TCP ACK

Frame 13: Servidor → Cliente Java
```

[PSH, ACK] Payload: "WELCOME TelemetryServer PROTO 1.0\n"

Frame 20: Cliente Python → Servidor
[PSH, ACK] Payload: "HELLO Isabella\r\n"

Frame 21: Servidor → Cliente Python
[PSH, ACK] Payload: "OK hello Isabella\n"

Frame 30: Servidor → BROADCAST (ambos clientes)
[PSH, ACK] Payload: "DATA speed=50.0 battery=98.6 temp=34.8 heading=103.0 ts=...\n"

Observaciones:

- Protocolo TCP garantiza entrega ordenada
- Mensajes terminan en \n (servidor) o \r\n (cliente)
- Sin pérdida de paquetes en localhost
- RTT promedio: <1ms
- Broadcast implementado con envío secuencial a cada cliente

## Tabla de Cobertura de Pruebas:

| ID | Caso de Prueba | Python | Java | Observaciones |
|---|---|---|---|---|
| T1 | Conexión básica | PASS | PASS | Ambos conectan correctamente |
| T2 | Telemetría cada 10s | PASS | PASS | Precisión de timing |
| T3 | AUTH exitoso | PASS | PASS | Elevación a ADMIN |
| T4 | AUTH fallido | PASS | PASS | Error 401 correcto |
| T5 | CMD como ADMIN | PASS | PASS | ACK recibido |
| T6 | CMD como VIEWER | PASS | PASS | Python previene, Java ERROR 403 |
| T7 | USERS múltiples | PASS | PASS | Lista correcta |
| T8 | Desconexión limpia | PASS | PASS | BYE - OK bye |
| T9 | Desconexión abrupta | PASS | PASS | Sin crash |
| T10 | NACK batería baja | PASS | PASS | Validación servidor |
| T11 | Comandos secuenciales | PASS | PASS | Estado consistente |
| T12 | Interoperabilidad Python-Java | PASS | PASS | Ambos reciben broadcast |

12/12 casos = 100%

**Comparación Final: Cliente Python vs Cliente Java**

| **Aspecto** | **Python (Tkinter)** | **Java (Swing)** |
|---|---|---|
| Líneas de código | ~400 | ~330 |
| Framework GUI | Tkinter | Swing |
| Threading | threading.Thread | Thread |
| Thread-safety GUI | root.after() | SwingUtilities.invokeLater() |
| Parsing telemetría | Regex (re.findall) | split() + HashMap |
| Buffering socket | Manual (recv_line_accum) | BufferedReader (auto) |
| Validación rol | Cliente + Servidor | Solo Servidor |
| Botones comando | Dedicados (4 botones) | Campo texto libre |
| Timestamps en log | Sí | No |
| Colores en labels | Sí (azul/naranja/rojo) | No |
| Diálogo AUTH | 2 campos separados | Panel grid 2x2 |
| Espera antes de BYE | No | Sí (150ms) |
| Complejidad código | Media | Media-Baja |
| Portabilidad | Alta (Python en todas partes) | Alta (JVM en todas partes) |

igualmente válidos y funcionales. La elección depende de:

- Python: Mejor para prototipado rápido, scripting, data science.
- Java: Mejor para aplicaciones enterprise, Android, grandes equipos.

## Conclusiones

El proyecto alcanzó de manera exitosa todos los objetivos planteados, demostrando solidez tanto en el diseño como en la implementación técnica. El protocolo de aplicación PROTO 1.0 fue diseñado, especificado y validado en su totalidad, mostrando un vocabulario claro, basado en texto plano sobre TCP, y garantizando la interoperabilidad total entre los clientes Python y Java. Esta base permitió la integración fluida con el servidor concurrente en C, logrando un sistema funcional y confiable.

En cuanto a los clientes, se desarrollaron dos aplicaciones completas: una en Python con Tkinter y otra en Java con Swing. Ambas presentan interfaces gráficas intuitivas y robustas, con mecanismos de concurrencia basados en hilos que aseguran la ejecución asíncrona de las operaciones de entrada y salida sin comprometer la responsividad de la GUI. La interoperabilidad fue probada con éxito, confirmando la compatibilidad total entre ambos entornos.

El servidor en C, construido con la API de Berkeley Sockets, implementa un modelo concurrente mediante hilos para atender múltiples clientes y manejar la difusión periódica de telemetría en tiempo real. Su diseño, además, incluye un sistema de logging completo, lo cual aporta trazabilidad y transparencia en las operaciones.

Las pruebas exhaustivas, que abarcaron 12 casos con un 100% de éxito, validaron la estabilidad, robustez y correcta implementación del sistema. Estas pruebas incluyeron la verificación con Wireshark, la interoperabilidad entre los clientes, así como escenarios de desconexión y errores.

A nivel de aprendizaje técnico, el proyecto reforzó competencias en programación de redes con sockets en Python y Java, diseño de protocolos de aplicación, manejo de concurrencia y sincronización de hilos, desarrollo de interfaces gráficas de usuario, e implementación de mecanismos de manejo de errores y excepciones. La experiencia también resaltó la importancia de detalles como el uso de terminadores de línea, codificación uniforme de caracteres y separación de responsabilidades entre hilos de I/O y GUI.

En síntesis, el proyecto constituye una demostración completa de competencias en redes, protocolos, concurrencia, GUI y robustez de software, cumpliendo todos los requerimientos planteados. Al mismo tiempo, sienta las bases para futuras extensiones hacia mayor seguridad, persistencia de datos y visualizaciones más avanzadas, lo que abre un camino sólido para su evolución en entornos más complejos y reales.

**Bibliografía**


- (n.d.). Wireshark • Go Deep. Retrieved October 5, 2025, from https://www.wireshark.org/
- Berkeley sockets. (n.d.). Wikipedia. Retrieved October 5, 2025, from https://en.wikipedia.org/wiki/Berkeley_sockets
- C Tutorial. (n.d.). W3Schools. Retrieved October 5, 2025, from https://www.w3schools.com/c/
- Kerrisk, M. (n.d.). socket(7) - Linux manual page. Michael Kerrisk. Retrieved October 5, 2025, from https://man7.org/linux/man-pages/man7/socket.7.html
- RFC 793: Transmission Control Protocol. (n.d.). RFC Editor. Retrieved September 21, 2025, from https://www.rfc-editor.org/rfc/rfc793.html
- Sinha, A. (2025, August 7). Socket Programming in C. GeeksforGeeks. Retrieved October 5, 2025, from https://www.geeksforgeeks.org/c/socket-programming-cc/
- TCP Server-Client implementation in C. (2025, July 11). GeeksforGeeks. Retrieved October 5, 2025, from https://www.geeksforgeeks.org/c/tcp-server-client-implementation-in-c/