

Programación Orientada a Objetos
Universidad Nacional de Cuyo - Facultad de
Ingeniería

TP2 – Actividad 2

TRABAJO PRACTICO Nº 2 – Desarrollo
Orientado a Objetos

F. Barrios Retta

Septiembre 2025

Incluye código probado en hardware.

Contents

1	Consideraciones Generales	3
1.1	Alcance	3
1.2	Hipótesis y supuestos de diseño	3
1.3	Principios OO aplicados	3
2	Esquema general de la solución	3
2.1	Diseño de la solución implementada	3
2.2	Diseño de solución secundaria: Intérprete tabular con Frame único y Codec JSONL	4
2.2.1	Descripción general	4
2.2.2	Objetivos	5
2.2.3	Componentes	5
2.2.4	Contrato de mensaje (resumen)	5
2.2.5	Flujo de operación	5
2.2.6	Ventajas y limitaciones	5
2.2.7	Párrafo breve para el informe	6
3	Interfaces de usuario	6
3.1	Descripción general de la Interfaz de Usuario	6
3.2	Modalidades de uso	6
3.2.1	Modo tubería (pipeline)	6
3.2.2	Modo archivo	6
3.2.3	Modo puerto serie (Linux)	6
3.3	Convenciones de entrada/salida	7
4	Recursos adicionales	7
5	Manual de instrucciones de la aplicación	7
5.1	Requisitos del entorno	7
5.2	Compilación	8
5.2.1	Con Makefile	8
5.2.2	Línea de comandos (alternativa)	8
5.3	Ejecución Básica (STDIN/STDOUT)	9

5.4	Contrato del mensaje (formato)	9
5.4.1	Estructura del bloque data (columnar)	9
5.5	Flujo de operación	9
5.5.1	Envío	9
5.5.2	Recepción	10
5.6	Integración con archivos y tuberías	10
5.7	Uso con puerto serie (GNU/Linux)	10
5.7.1	Detección y configuración	10
5.7.2	Intercambio de mensajes	10
5.8	Pruebas y validación	10
5.8.1	Prueba de humo (JSON Lines)	10
5.8.2	Verificación de DataBag	11
5.8.3	Normalización de finales de línea	11
6	Conclusiones	11
7	Referencias consultadas	12
8	Anexo	12

1 Consideraciones Generales

Implementar un canal de mensajería line-delimited JSON entre procesos/dispositivos, con jerarquía de clases para mensajes, serialización/deserialización y un contenedor tabular columnar para el payload. La app se compila con C++17 y organiza headers/definitions con compilación separada.

1.1 Alcance

- Envío y recepción de mensajes por streams estándar (`std::istream/std::ostream`) mediante la clase `Node`.
- Mensajes concretos: `Message` y `AuthMessage` derivados de `IMessage`.
- Serialización: `toJSON()` en cada mensaje; deserialización: `fromJSON(string_view)` tipo factoría.
- `DataBag` columnar: `rows + columns vector<T>` con $T \in \{\text{int}, \text{double}, \text{bool}, \text{string}\}$.
- El framing es “una línea = un mensaje”. Esto replica la pauta de “un objeto por línea” (en el documento de referencia se enumeran CSV/JSON/XML por línea)

1.2 Hipótesis y supuestos de diseño

- Transporte: lectura con `std::getline`, tolerando `\r\n`. Cada mensaje llega completo por línea.
- Formato: JSON UTF-8, un objeto por línea.
- Errores: si la línea es ilegible, la factoría puede devolver `nullptr` o emitir excepción. Estas hipótesis reflejan la guía del TP sobre framing por línea y robustez de entrada/salida, ajustadas a nuestro caso exclusivamente JSON

1.3 Principios OO aplicados

- **Encapsulamiento y cohesión:** `Node` solo conoce streams y funciones de serialización.
- **Polimorfismo:** `IMessage` como interfaz; `Message/AuthMessage` sobrescriben `toJSON()`.
- **Compilación separada:** headers livianos, definiciones en `.cpp`. Esto es coherente con los principios señalados en el modelo (encapsulamiento, modularidad, bajo acoplamiento)

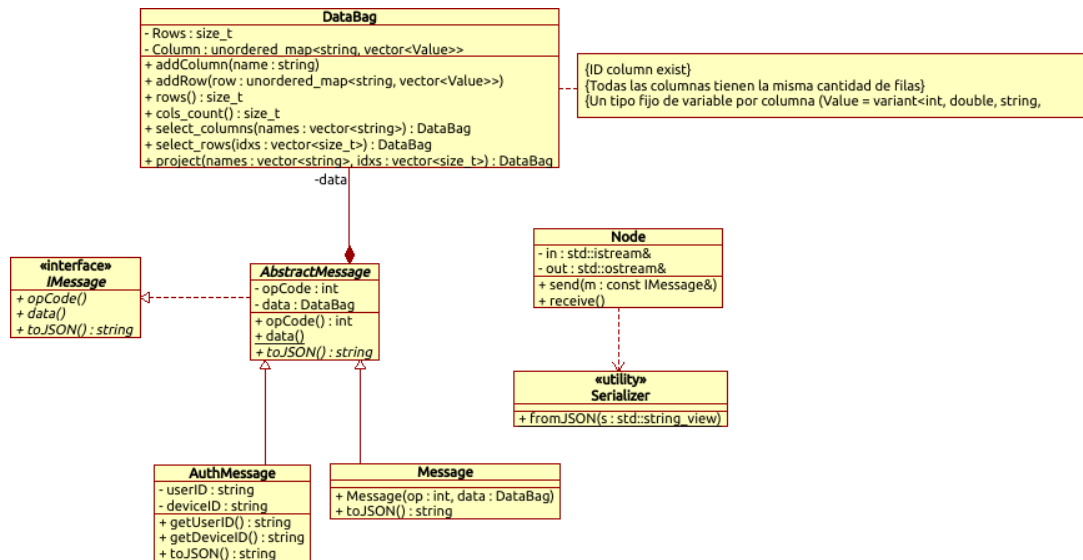
2 Esquema general de la solución

2.1 Diseño de la solución implementada

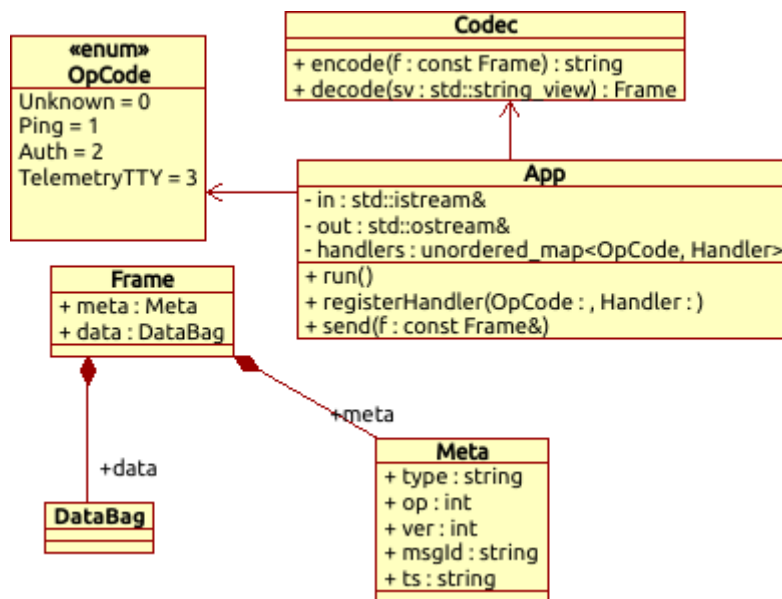
Se adoptó una arquitectura simple orientada a objetos, respetando el diagrama de clases provisto en la consigna. El diagrama actualizado se ilustra a continuación:

- `Node`: I/O de líneas. `sendMessage(IMessage&)` escribe `toJSON()` y agrega `'\n'`. `receiveMessage()` hace `getline` y llama a `fromJSON()`.
- `IMessage / AbstractMessage`: exponen `opCode()` y `data()`.
- `Message`: mensaje genérico con `op` y `DataBag`.
- `AuthMessage`: agrega `userID` y `deviceID`.

- **DataBag**: columnar: rows y columns como `variant<vector<int|double|string|bool>>`.
- **fromJSON()**: factoría mínima que detecta tipo y crea el subtipo correspondiente.



2.2 Diseño de solución secundaria: Intérprete tabular con Frame único y Codec JSONL



2.2.1 Descripción general

Esta solución elimina la jerarquía de mensajes y el despacho polimórfico. En su lugar, utiliza:

- Un **único tipo de dato** Frame que contiene Meta y DataBag.
- Un **Codec** que codifica/decodifica una línea JSON Frame.
- Un **intérprete tabular**: una tabla `handlers[OpCode]` que mapea códigos de operación a funciones manejadoras.
- Una clase **App** que orquesta el bucle de lectura, decodificación, ejecución del handler y (opcional) emisión de respuesta.

Se mantiene el protocolo: **JSON por línea** (JSONL), UTF-8, con **data** en formato **columnar**.

2.2.2 Objetivos

- Reducir al mínimo el número de clases y dependencias.
- Conservar el framing por línea y `DataBag` columnar.
- Lograr ruteo determinista por `OpCode` sin polimorfismo ni `dynamic_cast`.

2.2.3 Componentes

- **App**: bucle principal, registro de handlers, E/S por streams.
- **Codec**: `encode(const Frame&) -> std::string` y `decode(std::string_view) -> Frame`.
- **Frame**: estructura con `Meta meta;` `DataBag data;`.
- **Meta**: `type, op` (compatible con `enum class OpCode:int`), `versión`, `msgId`, `ts`.
- **DataBag**: bloque columnar (`rows + columns<string → vector<T>>`, con `T {int,double,string,bool}`).
- **Tabla de handlers**: `std::unordered_map<OpCode, Handler>` donde `Handler` es `std::function<void(const Frame&, App&>`.

2.2.4 Contrato de mensaje (resumen)

- **Framing**: 1 línea = 1 mensaje (`\n`; se tolera `\r\n`).
- **Campos mínimos**:
 - `meta.type`: "auth", "message", etc.
 - `meta.op`: entero mapeable a `OpCode`.
- **data opcional (columnar)**:

```
"data": {  
  "rows": 3,  
  "columns": {  
    "id": [1,2,3],  
    "ok": [true,false,true]  
  }  
}
```

2.2.5 Flujo de operación

1. `App` lee una línea del `istream`.
2. `Codec.decode` produce un `Frame`.
3. `App` consulta `handlers[frame.meta.op]` y ejecuta la función registrada.
4. El `handler`, si corresponde, construye un `Frame` de respuesta y llama a `App.send(frameResp)`, que usa `Codec.encode` y escribe una línea al `ostream`.

2.2.6 Ventajas y limitaciones

- **Ventajas**: muy pocas clases, cero herencia, cero conversión en tiempo de ejecución; testeo y extensión por tabla (`registerHandler`).

- **Limitaciones:** no hay validaciones transversales integradas (versionado/autorización); si se requieren, se agregan en **App** antes del despacho.

2.2.7 Párrafo breve para el informe

La variante C-min reemplaza la jerarquía polimórfica por un **intérprete tabular**: un **Frame** único encapsula **Meta** y **DataBag**, y una tabla de handlers asocia cada **OpCode** con su lógica. **App** realiza el bucle de lectura y despacho, mientras que **Codec** se encarga de la (de)serialización JSON por línea. El resultado preserva el contrato de intercambio y simplifica la estructura a tres clases principales más una tabla de funciones, manteniendo el diseño orientado a objetos donde aporta valor (orquestración y encapsulamiento), pero evitando herencia y conversiones dinámicas.

3 Interfaces de usuario

3.1 Descripción general de la Interfaz de Usuario

La aplicación no dispone de interfaz gráfica. La interacción se realiza mediante:

- **Consola (STDIN/STDOUT/STDERR)** para procesamiento por lotes o encadenamiento con otros procesos.
- **Archivos** en formato JSON Lines (`.jsonl`) para entradas y salidas persistentes.
- **Puerto serie** para intercambio con dispositivos embebidos (por ejemplo, microcontroladores).

El sistema opera en modo **no interactivo**: procesa una secuencia de mensajes, uno por línea, y termina al alcanzar EOF o un error fatal.

3.2 Modalidades de uso

3.2.1 Modo tubería (pipeline)

Permite integrar la aplicación en cadenas de procesamiento.

```
1 productor | ./tp2_actividad2 | consumidor
2
```

- Entrada y salida son flujos de texto UTF-8.
- Cada línea corresponde a un mensaje JSON completo.

3.2.2 Modo archivo

Adecuado para pruebas reproducibles o integración con datasets.

```
1 ./tp2_actividad2 < input.jsonl > output.jsonl
2
```

- `input.jsonl`: un objeto JSON por línea.
- `output.jsonl`: una línea por mensaje producido.

3.2.3 Modo puerto serie (Linux)

Intercambio con el dispositivo a través de `stdin/stdout` redirigidos al TTY.

```
1 stty -F /dev/ttyACM0 115200 cs8 -parenb -cstopb -echo -icanon -ixon -ixoff -crnl
2 ./tp2_actividad2 < /dev/ttyACM0
3 # o envío de una línea JSON
4 printf '%s\n' '{"type":"auth","op":2,"userID":"u","deviceID":"esp32"}' > /dev/ttyACM0
5
```

- Requisito: el firmware debe emitir **un \n por mensaje**. Se tolera \r\n y se normaliza internamente.

3.3 Convenciones de entrada/salida

- **Framing:** 1 línea = 1 mensaje. El emisor no debe incluir \n dentro del objeto; la aplicación agrega \n al enviar y usa `getline` al recibir.
- **Codificación:** UTF-8.
- **Formato de mensaje:** objeto JSON con campos mínimos `type` y `op`, y opcional `data` (bloque tabular columnar).
- **Buffering:** la aplicación no fuerza `flush` continuo; en pipelines, el flushing lo gestiona el SO. Para diagnóstico en tiempo real, el consumidor puede forzar `std::flush` o usar `stdbuf -oL`.

4 Recursos adicionales

No se hace uso de ningún componente de software no estándar del lenguaje ni de la plataforma para codificar el programa. A continuación, se listan las librerías estándar de C++ y de Linux que se utilizan en los diferentes archivos `.h` y `.cpp`.

```
1 #include <istream>
2 #include <ostream>
3 #include <sstream>
4 #include <memory>
5 #include <stdexcept>
6 #include <cstdint>
7 #include <utility>
8
9 // Tipos de variables
10 #include <string>
11 #include <vector>
12 #include <variant>
13 #include <unordered_map>
14
```

5 Manual de instrucciones de la aplicación

5.1 Requisitos del entorno

Compilador C++ con soporte para C++17 (por ejemplo, `g++ 9`).

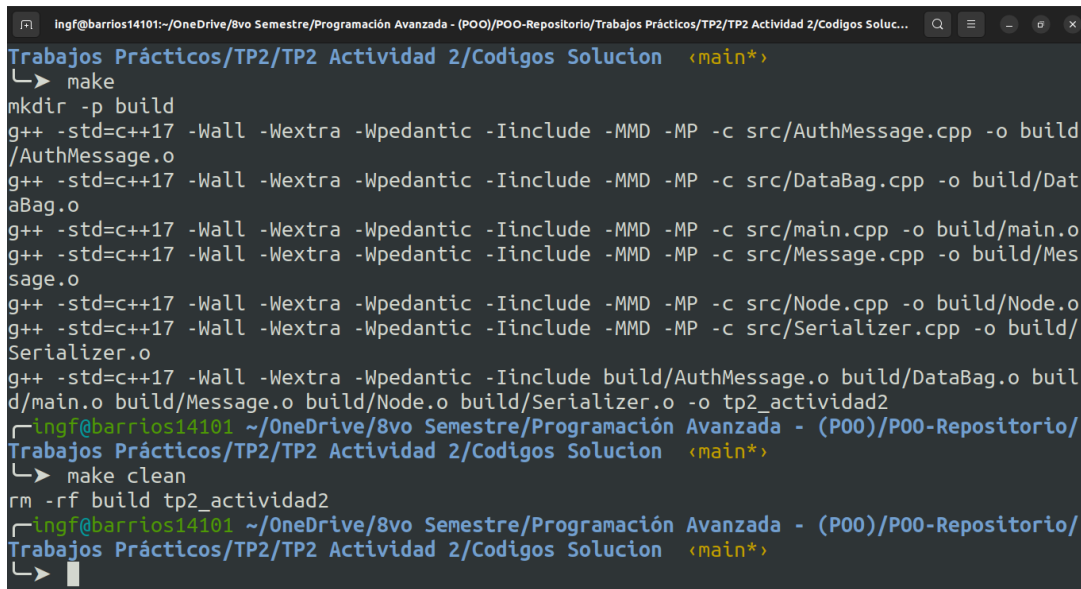
Herramienta de construcción `make` si se utiliza el `Makefile` provisto.

Sistema operativo tipo POSIX para pruebas con puerto serie (ej.: GNU/Linux).

5.2 Compilación

5.2.1 Con Makefile

```
1 make          # compila el proyecto
2 make clean    # elimina artefactos de compilación
3
```

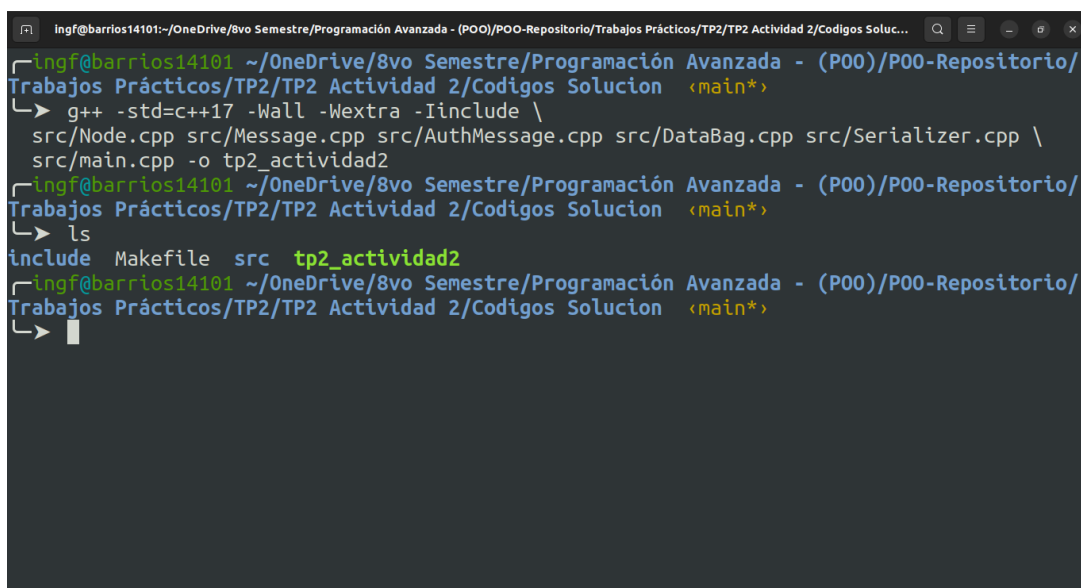


```
Trabajos Prácticos/TP2/TP2 Actividad 2/Codigos Solucion <main*>
└─> make
mkdir -p build
g++ -std=c++17 -Wall -Wextra -Wpedantic -Iinclude -MMD -MP -c src/AuthMessage.cpp -o build/AuthMessage.o
g++ -std=c++17 -Wall -Wextra -Wpedantic -Iinclude -MMD -MP -c src/DataBag.cpp -o build/DataBag.o
g++ -std=c++17 -Wall -Wextra -Wpedantic -Iinclude -MMD -MP -c src/main.cpp -o build/main.o
g++ -std=c++17 -Wall -Wextra -Wpedantic -Iinclude -MMD -MP -c src/Message.cpp -o build/Message.o
g++ -std=c++17 -Wall -Wextra -Wpedantic -Iinclude -MMD -MP -c src/Node.cpp -o build/Node.o
g++ -std=c++17 -Wall -Wextra -Wpedantic -Iinclude -MMD -MP -c src/Serializer.cpp -o build/Serializer.o
g++ -std=c++17 -Wall -Wextra -Wpedantic -Iinclude build/AuthMessage.o build/DataBag.o build/main.o build/Message.o build/Node.o build/Serializer.o -o tp2_actividad2
ingf@barrios14101 ~/OneDrive/8vo Semestre/Programación Avanzada - (P00)/P00-Repositorio/Trabajos Prácticos/TP2/TP2 Actividad 2/Codigos Solucion <main*>
└─> make clean
rm -rf build tp2_actividad2
ingf@barrios14101 ~/OneDrive/8vo Semestre/Programación Avanzada - (P00)/P00-Repositorio/Trabajos Prácticos/TP2/TP2 Actividad 2/Codigos Solucion <main*>
└─>
```

Parámetros esperados del compilador: -std=c++17 -Wall -Wextra -Wpedantic

5.2.2 Línea de comandos (alternativa)

```
1 g++ -std=c++17 -Wall -Wextra -Iinclude \
2 src/Node.cpp src/Message.cpp src/AuthMessage.cpp src/DataBag.cpp src/Serializer.cpp \
3 src/main.cpp -o tp2_actividad2
4
```



```
ingf@barrios14101 ~/OneDrive/8vo Semestre/Programación Avanzada - (P00)/P00-Repositorio/Trabajos Prácticos/TP2/TP2 Actividad 2/Codigos Solucion <main*>
└─> g++ -std=c++17 -Wall -Wextra -Iinclude \
    src/Node.cpp src/Message.cpp src/AuthMessage.cpp src/DataBag.cpp src/Serializer.cpp \
    src/main.cpp -o tp2_actividad2
ingf@barrios14101 ~/OneDrive/8vo Semestre/Programación Avanzada - (P00)/P00-Repositorio/Trabajos Prácticos/TP2/TP2 Actividad 2/Codigos Solucion <main*>
└─> ls
include Makefile src tp2_actividad2
ingf@barrios14101 ~/OneDrive/8vo Semestre/Programación Avanzada - (P00)/P00-Repositorio/Trabajos Prácticos/TP2/TP2 Actividad 2/Codigos Solucion <main*>
└─>
```


5.3 Ejecución Básica (STDIN/STDOUT)

La aplicación procesa un mensaje por línea en formato JSON (JSON Lines).

- Entrada: `stdin` (una línea JSON por mensaje, terminada en `\n`).
- Salida: `stdout` (una línea JSON por mensaje producido).

Por ejemplo:

```
1 printf '%s\n' \  
2 '{"type":"message","op":42}' \  
3 '{"type":"auth","op":7,"userID":"u","deviceID":"d"}' \  
4 | ./tp2_actividad2  
5
```

La aplicación leerá cada línea, la deserializará y la procesará de acuerdo con su tipo.

5.4 Contrato del mensaje (formato)

- Framing: 1 línea = 1 mensaje. Fin de línea `\n`. Se acepta `\r\n` y se normaliza a `\n`.
- Codificación: UTF-8.
- Campos obligatorios:
 - `type`: "message" o "auth".
 - `op`: entero que identifica la operación (código de operación).
- Campo opcional:
 - `data`: bloque tabular columnar.

5.4.1 Estructura del bloque data (columnar)

```
1 "data": {  
2   "rows": 3,  
3   "columns": {  
4     "id": [1, 2, 3],  
5     "temp": [21.5, 22.0, 23.1],  
6     "ok": [true, false, true],  
7     "name": ["a", "b", "c"]  
8   }  
9 }  
10
```

Invariante: para toda columna `columns[k]`, `columns[k].size() == rows`.

5.5 Flujo de operación

5.5.1 Envío

1. El emisor construye el objeto de mensaje.
2. `toJSON()` genera una cadena **sin** salto de línea.
3. `Node::sendMessage(const IMessage&)` escribe la cadena y agrega `'\n'` al flujo de salida.

5.5.2 Recepción

1. `Node::receiveMessage()` lee una línea mediante `std::getline`.
2. Se elimina un `'\r'` final si existiese (`\r\n`).
3. `fromJson(std::string_view)` decide el subtipo (`Message` o `AuthMessage`) y construye el objeto resultante.
4. En caso de EOF o error de lectura, `receiveMessage()` retorna `nullptr`.

5.6 Integración con archivos y tuberías

- Lectura desde archivo:

```
./tp2_actividad2 < input.jsonl > output.jsonl
```

- Composición con otros procesos:

```
productor | ./tp2_actividad2 | consumidor
```

5.7 Uso con puerto serie (GNU/Linux)

5.7.1 Detección y configuración

Identificar el dispositivo (p. ej., `/dev/ttyACM0` o `/dev/ttyUSB0`) y configurar el modo raw:

```
1 stty -F /dev/ttyACM0 115200 cs8 -parenb -cstopb \  
2   -echo -icanon -ixon -ixoff -crnl  
3
```

- Velocidad: 115200 baudios (ajustable).
- Formato: 8N1 (8 bits de datos, sin paridad, 1 bit de stop).
- Desactivación de eco, modo canónico y control de flujo por software.

5.7.2 Intercambio de mensajes

- Enviar al dispositivo:

```
printf '%s\n' '{"type":"auth","op":2,"userID":"u","deviceID":"esp32"}' > /dev/ttyACM0
```

- Recibir desde el dispositivo:

```
./tp2_actividad2 < /dev/ttyACM0
```

Requisito: el firmware debe transmitir una línea JSON por mensaje. Se admiten finales `\r\n`.

5.8 Pruebas y validación

5.8.1 Prueba de humo (JSON Lines)

Entrada:

```
1 {"type":"message","op":42}  
2 {"type":"auth","op":7,"userID":"u","deviceID":"d"}  
3
```

Procedimiento:

```
1 printf '%s\n' \  
2 '{"type":"message","op":42}' \  
3 '{"type":"auth","op":7,"userID":"u","deviceID":"d"}' \  
4 | ./tp2_actividad2  
5
```

Criterio de aceptación: cada línea se procesa sin error; el programa mantiene el framing y reconoce el subtipo.



```
ingf@barrios14101:~/OneDrive/8vo Semestre/Programación Avanzada - (P00)/P00-Repositorio/Trabajos Prácticos/TP2/TP2 Actividad 2/Codigos Solucion <main*>  
└─> ./tp2_actividad2  
{\"type\":\"message\",\"op\":42}  
{\"type\":\"auth\",\"op\":7,\"userID\":\"u\",\"deviceID\":\"d\"}  
OUT BUF: {\"type\":\"message\",\"op\":99,\"data\":{\"rows\":3,\"columns\":{\"name\":[\"a\",\"b\",\"c\"],\"ok\":[true,false,true],\"temp\":[21.5,22,23.1],\"id\":[1,2,3]}}}  
  
ingf@barrios14101:~/OneDrive/8vo Semestre/Programación Avanzada - (P00)/P00-Repositorio/Trabajos Prácticos/TP2/TP2 Actividad 2/Codigos Solucion <main*>  
└─>
```

5.8.2 Verificación de DataBag

1. Construir un DataBag con 2–4 columnas y rows > 0.
2. Enviar un Message con data y comprobar que la serialización JSON cumple el invariante de tamaños.
3. Probar selectRows y selectColumns con índices válidos y detectar errores ante índices fuera de rango.

5.8.3 Normalización de finales de línea

Probar entradas con `\n` y `\r\n`. Criterio: el receptor debe aceptar ambos y operar internamente con `\n`.

6 Conclusiones

La solución implementada cumple con los objetivos establecidos: define un protocolo de intercambio basado en JSON line-delimited (un mensaje por línea) y una jerarquía de clases que separa responsabilidades entre interfaz (`IMessage`), factor común (`AbstractMessage`) y tipos concretos (`Message`, `AuthMessage`). La clase `Node` encapsula la E/S sobre `std::istream`/`std::ostream`, garantizando el enmarcado por línea en el envío (`sendMessage` agrega el salto de línea) y la lectura robusta en la recepción (`receiveMessage` normaliza `\r\n` y delega la construcción a la factoría `fromJSON`).

El campo `op` opera como código de operación y permite el ruteo determinista del flujo de

control en el receptor, mientras que el bloque `data` encapsula el payload en un modelo tabular columnar. Esta representación columnar, sustentada en vectores tipados (`int`, `double`, `bool`, `string`), asegura coherencia estructural mediante el invariante `|columna| = rows` y optimiza el acceso secuencial a datos. Las operaciones de proyección y filtrado (`selectColumns`, `selectRows`) se implementan sin romper dicho invariante.

Las pruebas de humo demuestran el funcionamiento extremo a extremo: envío y recepción por líneas, identificación polimórfica del tipo de mensaje y serialización consistente. La arquitectura resultante es extensible: incorporar nuevos tipos de mensajes o columnas requiere cambios localizados (un subtipo adicional y su caso en la factoría), sin afectar el resto del sistema. Como trabajo futuro, se recomienda:

- 1) completar el parseo inverso del bloque `data` hacia `DataBag` con validación de tipos y tamaños,
- 2) formalizar un esquema de columnas por operación (`op`) para verificación temprana, y
- 3) parametrizar políticas de tiempo de espera, reintentos y manejo de errores en `Node` según el medio de transporte (archivo, tubería o puerto serie).

7 Referencias consultadas

- Apuntes de cátedra de POO (sección “Guía de Trabajos Prácticos”).
- 115 Ejercicios resueltos de programación C++(Ra-Ma) by Joefebeus & Iryopogu

8 Anexo