Universidad Politécnica de Victoria

Ingeniería en Tecnologías de la Información e Innovación Digital

---

Sistema Decodificador de Protocolo Industrial (PRT-7)

Alumno:
Junior Arturo Vazquez Leonel

Asignatura:
Estructura de Datos

Doctor:
SAID POLANCO MARTAGÓN

6 de noviembre del 2025

I. Introducción

En el ámbito de la Industria 4.0 y el Internet de las Cosas (IoT), la transmisión de datos entre sensores y estaciones centrales es fundamental. A menudo, estos datos se envían en formatos compactos o protocolos personalizados que requieren un procesamiento en el lado del receptor para ser interpretados.

El proyecto "Sistema Decodificador de Protocolo Industrial (PRT-7)"simula este escenario. Se utiliza un microcontrolador ESP32 como *emisor*, que envía una secuencia de tramas de datos a través del puerto serial. Una aplicación C++ en el lado del *receptor* (una PC) escucha este puerto, interpreta las tramas y reconstruye un mensaje oculto.

El protocolo PRT-7 diseñado para este proyecto consta de dos tipos de tramas:

- **Trama de Carga (LOAD):** Formato `L,<caracter>`. Indica que un carácter debe ser decodificado y agregado al mensaje.
- **Trama de Mapeo (MAP):** Formato `M,<numero>`. Indica que el estado del decodificador (el rotor) debe cambiar, rotando su posición.

El objetivo principal de este trabajo es aplicar los conceptos teóricos de estructuras de datos, específicamente listas doblemente enlazadas y listas circulares, para resolver un problema práctico de decodificación de datos. Se utiliza polimorfismo para gestionar los diferentes tipos de tramas, demostrando un diseño de software orientado a objetos limpio y extensible.

II. Manual Técnico

Esta sección detalla la arquitectura del sistema, el diseño de las estructuras de datos y el flujo de procesamiento de la información.

II-A. Diseño y Arquitectura

El sistema se divide en dos componentes principales: el Emisor (ESP32) y el Receptor (PC).

II-A1. Emisor (ESP32): El programa `DecodificadorPRT7.ino` se carga en un ESP32. Su única función es simular un sensor que envía datos. Define un arreglo de tramas (ej: "L,E", "L,L", "M,2", etc.) y las envía una por una a través de `Serial.println()` con un retardo de 1 segundo entre cada una. Al finalizar la secuencia, envía un mensaje de reinicio y vuelve a empezar.

II-A2. Receptor (PC): Es la aplicación principal (`main.cpp`). Se conecta al puerto serial (ej: `/dev/ttyUSB0`) y entra en un bucle de lectura. Utiliza dos estructuras de datos clave para su funcionamiento: `RotorDeMapeo` y `ListaDeCarga`.

II-A3. Diseño de Clases (Polimorfismo): Para manejar las tramas `L` y `M` de manera elegante, se utiliza un diseño basado en polimorfismo:

- **TramaBase:** Es una clase base abstracta que define la interfaz `virtual void procesar(...)`.
- **TramaLoad:** Hereda de `TramaBase`. Implementa `procesar()` para decodificar un carácter usando el rotor y agregarlo a la lista de carga.
- **TramaMap:** Hereda de `TramaBase`. Implementa `procesar()` para rotar la posición del rotor de mapeo.

En `main.cpp`, al recibir una línea, se crea un objeto del tipo correspondiente (`new TramaLoad(...)` o `new TramaMap(...)`) y se invoca su método `procesar()`, permitiendo que cada objeto maneje su propia lógica.

II-B. Componentes (Estructuras de Datos)

El núcleo del proyecto son dos listas enlazadas implementadas manualmente.

II-B1. RotorDeMapeo (Lista Circular Doblemente Enlazada): Esta estructura actúa como un disco de cifrado César dinámico.

- **Estructura:** Implementada como una lista circular doblemente enlazada usando `NodoRotor`. Cada nodo almacena un `char` (de 'A' a 'Z' más un espacio).
- **Nodos:** `NodoRotor` contiene `char dato`, `NodoRotor* siguiente`, y `NodoRotor* previo`.
- **Estado:** Mantiene un puntero `cabeza` que indica la posición çero.ªctual del rotor. Inicialmente, `cabeza` apunta a 'A'.
- **Operación (Cifrado):** El método `getMapeo(char in)` decodifica un carácter. Lo hace calculando el *offset* del carácter de entrada respecto a 'A' (ej: 'C' tiene un offset de 2) y avanza ese mismo número de posiciones desde el nodo `cabeza` actual.
- **Operación (Rotación):** El método `rotar(int n)` simplemente mueve el puntero `cabeza n` posiciones hacia adelante (si `n` es positivo) o hacia atrás (si `n` es negativo) en la lista circular.

II-B2. ListaDeCarga (Lista Doblemente Enlazada): Esta estructura almacena el mensaje final decodificado.

- **Estructura:** Implementada como una lista doblemente enlazada estándar, con punteros `cabeza` y `cola`.
- **Nodos:** `NodoCarga` almacena el carácter original (`datoCodificado`) y el carácter resultante de la decodificación (`datoDecodificado`).
- **Operación (Inserción):** El método `insertarAlFinal(char codif, char decodif)` crea un nuevo `NodoCarga` y lo añade al final de la lista, actualizando el puntero `cola`.
- **Operación (Impresión):** El método `imprimirMensaje()` recorre la lista desde `cabeza` hasta `cola` e imprime solo el `datoDecodificado` de cada nodo.

II-C. Desarrollo y Flujo de Procesamiento

El flujo de datos completo, desde la transmisión hasta la decodificación final, sigue estos pasos:

1. Transmisión: El ESP32 envía una trama, por ejemplo: `L,E`.

2. Recepción: `main.cpp` lee la línea `"L,E"` del puerto serial.

3. Parseo: La función `procesarLinea` identifica el tipo 'L' y el dato 'E'.

4. Creación Polimórfica: Se instancia un objeto `TramaLoad`: `TramaBase* trama = new TramaLoad('E');`

5. Procesamiento (Caso TramaLoad): Se invoca `trama->procesar(carga, rotor);`.

1. `TramaLoad::procesar` llama a `rotor->getMapeo('E')`.
2. `RotorDeMapeo::getMapeo('E')` calcula el offset de 'E' (4) y busca el nodo 4 posiciones después de la `cabeza` actual. Si la `cabeza` apunta a 'A', el resultado es 'E'.
3. `TramaLoad::procesar` llama a `carga->insertarAlFinal('E', 'E')`.
4. `ListaDeCarga::insertarAlFinal` crea un `NodoCarga` y lo añade al final de la lista de carga.

6. Transmisión (Trama de Mapeo): El ESP32 envía la trama `M,2`.

7. Procesamiento (Caso TramaMap):

1. Se crea `TramaBase* trama = new TramaMap(2);`.
2. Se invoca `trama->procesar(carga, rotor);`.
3. `TramaMap::procesar` llama a `rotor->rotar(2)`.
4. `RotorDeMapeo::rotar(2)` mueve el puntero `cabeza` dos posiciones hacia adelante. La `cabeza` ahora apunta a 'C'. El mapeo ha cambiado: ahora 'A' se decodifica como 'C'.

8. Transmisión (Siguiente TramaLoad): El ESP32 envía `L,A`.

9. Procesamiento (Efecto del Mapeo):

1. Se crea `TramaBase* trama = new TramaLoad('A');`.
2. `TramaLoad::procesar` llama a `rotor->getMapeo('A')`.
3. `RotorDeMapeo::getMapeo('A')` calcula el offset de 'A' (0). Como la `cabeza` ahora apunta a 'C', el resultado de la decodificación es 'C'.
4. `TramaLoad::procesar` llama a `carga->insertarAlFinal('A', 'C')`.

10. Resultado Final: Este proceso se repite. Cuando la aplicación receptora detecta el mensaje REINICIANDO SECUENCIA", llama a `carga->imprimirMensaje()`, que recorre la `ListaDeCarga` y muestra el mensaje oculto ensamblado.

II-D. Implementación de Clases

A continuación se presentan las cabeceras (.h) de las clases principales que definen la arquitectura del sistema.

```
#ifndef TRAMABASE_H
#define TRAMABASE_H

// Forward declarations
class ListaDeCarga;
class RotorDeMapeo;

class TramaBase {
public:
    virtual ~TramaBase() {}
    virtual void procesar(ListaDeCarga* carga,
     RotorDeMapeo* rotor) = 0;
};

#endif // TRAMABASE_H
```

Listing 1: TramaBase.h

Explicación: Esta es la clase base abstracta. Define la interfaz común `procesar(...)` que todas las tramas (Load y Map) deben implementar. El uso de un destructor virtual `~TramaBase()` es crucial para asegurar que se llame al destructor correcto (el de la clase derivada) cuando se elimina un objeto a través de un puntero a `TramaBase`, evitando fugas de memoria.

```
#ifndef ROTORDEMAPEO_H
#define ROTORDEMAPEO_H

struct NodoRotor {
    char dato;
    NodoRotor* siguiente;
    NodoRotor* previo;

    NodoRotor(char c) : dato(c), siguiente(
    nullptr), previo(nullptr) {}
};

class RotorDeMapeo {
private:
    NodoRotor* cabeza;
    int tamanio;

    NodoRotor* encontrarNodo(char c) const;
    int calcularDistancia(NodoRotor* desde,
    NodoRotor* hasta) const;

public:
    RotorDeMapeo();
    ~RotorDeMapeo();

    void rotar(int n);
    char getMapeo(char in) const;
    char getPosicionActual() const;
};

#endif // ROTORDEMAPEO_H
```

Listing 2: RotorDeMapeo.h

Explicación: Esta es la interfaz para la lista circular doblemente enlazada. Actúa como el disco de cifrado. Su constructor crea el alfabeto ('A'-'Z' y ' ') y su destructor libera la memoria. `rotar(n)` mueve el puntero `cabeza`, que es el que marca la referencia de cifrado. `getMapeo(char in)` calcula la decodificación.

```
#ifndef LISTADECARGA_H
#define LISTADECARGA_H

struct NodoCarga {
    char datoCodificado;
```

```
    char datoDecodificado;
    NodoCarga* siguiente;
    NodoCarga* previo;

    NodoCarga(char codif, char decodif) :
    datoCodificado(codif), datoDecodificado(
    decodif),

    siguiente(nullptr), previo(nullptr) {}
};

class ListaDeCarga {
private:
    NodoCarga* cabeza;
    NodoCarga* cola;
    int tamanio;

public:
    ListaDeCarga();
    ~ListaDeCarga();

    void insertarAlFinal(char codificado, char
     decodificado);
    void imprimirMensaje() const;
    void imprimirMensajeParcial() const;
    int obtenerTamanio() const;
    bool estaVacia() const;
};

#endif // LISTADECARGA_H
```

Listing 3: ListaDeCarga.h

Explicación: Define la interfaz para una lista doblemente enlazada estándar que almacenará el mensaje. Mantiene punteros a `cabeza` y `cola` para inserciones eficientes al final. Define los métodos para `insertarAlFinal(...)` y para `imprimirMensaje()` y `imprimirMensajeParcial()`.

```
#ifndef TRAMALOAD_H
#define TRAMALOAD_H

#include "TramaBase.h"

class TramaLoad : public TramaBase {
private:
    char dato;

public:
    TramaLoad(char c);
    ~TramaLoad();

    void procesar(ListaDeCarga* carga,
    RotorDeMapeo* rotor) override;
};

#endif // TRAMALOAD_H
```

Listing 4: TramaLoad.h

Explicación: Clase derivada que hereda de `TramaBase` y maneja las tramas ”L”. Almacena el carácter recibido (`dato`). Sobrescribe el método `procesar(...)` para implementar la lógica de decodificación (usando el Rotor) y almacenamiento (usando la ListaDeCarga).

```
#ifndef TRAMAMAP_H
#define TRAMAMAP_H

#include "TramaBase.h"

class TramaMap : public TramaBase {
private:
    int rotacion;

public:
    TramaMap(int n);
    ~TramaMap();

    void procesar(ListaDeCarga* carga,
    RotorDeMapeo* rotor) override;
};

#endif // TRAMAMAP_H
```

Listing 5: TramaMap.h

Explicación: Clase derivada que hereda de `TramaBase` y maneja las tramas ”M”. Almacena el valor de la rotación (`rotacion`). Sobrescribe el método `procesar(...)` para simplemente invocar a `rotor->rotar(rotacion)`, cambiando el estado del cifrado.

III. Resultados

La ejecución del sistema demuestra la correcta implementación de las estructuras de datos y el procesamiento polimórfico de las tramas. La aplicación receptora (`main.cpp`) se conecta exitosamente al ESP32 que actúa como emisor.

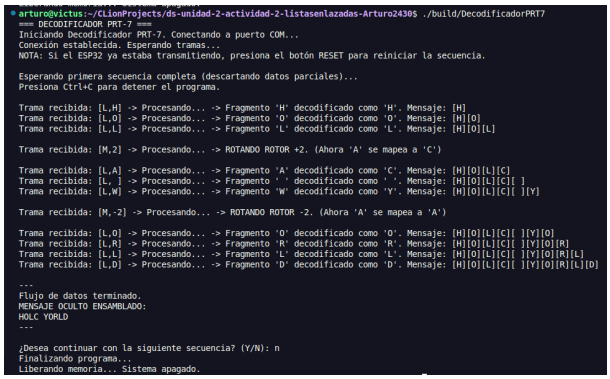

Figura 1: Ejecución del decodificador PRT-7 recibiendo la secuencia de tramas del ESP32. Se observa cómo las tramas ’L’ (LOAD) añaden caracteres y las tramas ’M’ (MAP) alteran el rotor, cambiando el mapeo (ej: ’A’ se mapea a ’C’) para formar el mensaje ”HOLA WORLD”.

Como se observa en la Figura 1, el programa principal decodifica el mensaje ”HOLA WORLD”. Al detectar la señal de reinicio del ESP32 (REINICIANDO SECUENCIA”), el sistema muestra el mensaje ensamblado por la `ListaDeCarga` y libera la memoria del `RotorDeMapeo` y la `ListaDeCarga`, preparándose para una nueva secuencia.

Para verificar la robustez del sistema, se cargó una secuencia de tramas diferente en el emisor (ESP32), como se muestra en la Figura 2.

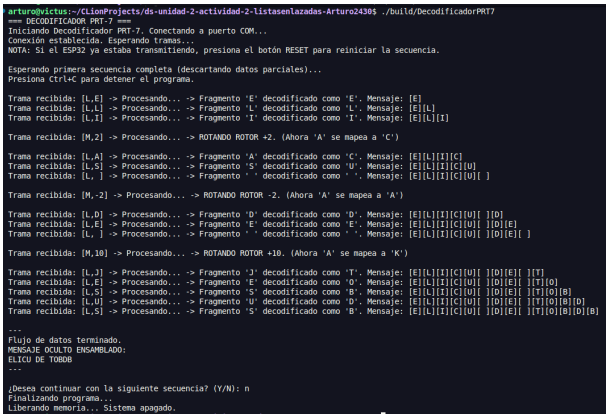

Figura 2: Ejecución del sistema con un conjunto de tramas alternativo. El sistema decodifica correctamente el mensaje ELIAS DE JESUS, demostrando que la lógica del rotor y la lista de carga funcionan independientemente del contenido transmitido.

El sistema fue capaz de decodificar el mensaje ELIAS DE JESUS sin necesidad de modificar el código del receptor, validando el diseño modular y la correcta gestión de estado del `RotorDeMapeo` y la `ListaDeCarga`.