Reporte Técnico: Implementación del Decodificador de Protocolo Industrial PRT-7

[Melissa Jazmin Torres Martinez]

Estructuras de Datos

[Universidad Politecnica de Victoria]

[ITIID]

7 de noviembre de 2025

Índice

1. Introducción

El presente documento describe el diseño, implementación y análisis del decodificador para el protocolo industrial PRT-7. Este sistema procesa tramas de comunicación serial utilizando estructuras de datos fundamentales implementadas manualmente sin el uso de la Biblioteca Estándar de Plantillas (STL) de C++.

1.1. Objetivos

- Implementar una lista doblemente enlazada para almacenar fragmentos decodificados
- Desarrollar una lista circular como rotor de mapeo (cifrado César dinámico)
- Aplicar polimorfismo para el procesamiento uniforme de diferentes tipos de tramas
- Gestionar comunicación serial y archivos de simulación

1.2. Alcance

El sistema decodifica tramas de dos tipos:

- **LOAD (L,X)**: Contiene un fragmento de dato a decodificar
- **MAP (M,N)**: Modifica la rotación del rotor de mapeo

2. Marco Teórico

2.1. Listas Enlazadas

2.1.1. Lista Doblemente Enlazada

Una estructura de datos lineal donde cada nodo contiene:

- Un dato
- Un puntero al nodo anterior
- Un puntero al nodo siguiente

Complejidad temporal:

- Inserción al final: $O(1)$ (con puntero tail)
- Recorrido completo: $O(n)$
- Eliminación: $O(1)$ si se tiene referencia al nodo

2.1.2. Lista Circular

Lista enlazada donde el último nodo apunta al primero, creando un ciclo. Permite rotaciones eficientes sin fin de lista.

2.2. Polimorfismo en C++

Capacidad de objetos de diferentes clases de responder a la misma interfaz. Implementado mediante:

- Clases base abstractas con funciones virtuales puras
- Destructores virtuales para correcta liberación de memoria
- Enlace dinámico en tiempo de ejecución

2.3. Protocolo PRT-7

Sistema de codificación que utiliza un rotor de mapeo basado en el cifrado César. La rotación dinámica del rotor permite diferentes mapeos para cada fragmento decodificado.

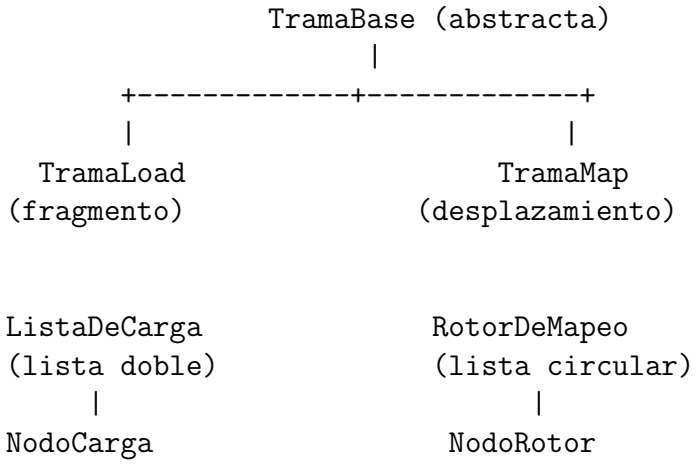
3. Diseño del Sistema

3.1. Arquitectura General

El sistema sigue una arquitectura modular con los siguientes componentes principales:

1. **Jerarquía de Tramas**: Sistema polimórfico para procesamiento uniforme
2. **Estructuras de Datos**: Implementaciones manuales de listas
3. **Sistema de E/S**: Abstracción de lectura serial/archivo
4. **Motor de Procesamiento**: Bucle principal de decodificación

3.2. Diagrama de Clases



3.3. Jerarquía de Clases

3.3.1. TramaBase

Clase abstracta que define la interfaz para todas las tramas:

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

Justificación del diseño:

- Permite extensibilidad para nuevos tipos de tramas
- Facilita el procesamiento uniforme mediante polimorfismo
- El destructor virtual previene fugas de memoria

3.3.2. TramaLoad

Hereda de TramaBase y procesa fragmentos de datos:

- Decodifica el fragmento usando el rotor actual
- Inserta el resultado en la lista de carga
- Muestra el estado actual del mensaje

3.3.3. TramaMap

Hereda de TramaBase y modifica el rotor:

- Rota el rotor N posiciones
- Maneja rotaciones positivas (derecha) y negativas (izquierda)
- No modifica la lista de carga

4. Implementación

4.1. Lista Doblemente Enlazada: ListaDeCarga

4.1.1. Estructura del Nodo

```
struct NodoCarga {
    char dato;
    NodoCarga* prev;
    NodoCarga* next;

    NodoCarga(char d) : dato(d),
                        prev(nullptr),
                        next(nullptr) {}
};
```

4.1.2. Operaciones Principales

Inserción al final:

```
void insertarAlFinal(char dato) {
    NodoCarga* n = new NodoCarga(dato);
    if (!tail) {
        head = tail = n;
    } else {
        tail->next = n;
        n->prev = tail;
        tail = n;
    }
}
```

Complejidad: $O(1)$ gracias al puntero `tail`
Destructor:

```
~ListaDeCarga() {
    NodoCarga* cur = head;
    while (cur) {
        NodoCarga* nx = cur->next;
        delete cur;
        cur = nx;
    }
}
```

Complejidad: $O(n)$ - debe recorrer toda la lista

4.2. Lista Circular: RotorDeMapeo

4.2.1. Construcción del Rotor

El rotor se inicializa con las 26 letras del alfabeto en orden:

```
RotorDeMapeo() : head(nullptr), size(0) {
    NodoRotor* first = nullptr;
    NodoRotor* prev = nullptr;

    for (char ch = 'A'; ch <= 'Z'; ++ch) {
        NodoRotor* n = new NodoRotor(ch);
        if (!first) first = n;
        if (prev) {
            prev->next = n;
            n->prev = prev;
        }
        prev = n;
        ++size;
    }

    // Cerrar circularidad
    if (first && prev) {
        first->prev = prev;
        prev->next = first;
```

```
        head = first;
    }
}
```

4.2.2. Rotación del Rotor

Implementación eficiente usando aritmética modular:

```
void rotar(int N) {
    if (!head || size <= 1) return;

    // Calcular desplazamiento efectivo
    int effective = N % size;
    if (effective < 0) effective += size;

    // Mover head
    for (int i = 0; i < effective; ++i) {
        head = head->next;
    }
}
```

Complejidad: $O(k)$ donde $k = N \bmod 26$

4.2.3. Mapeo de Caracteres

```
char getMapeo(char in) {
    if (in == ' ') return ' ';

    // Normalizar a mayuscula
    if (in >= 'a' && in <= 'z')
        in = char(in - 'a' + 'A');

    if (in < 'A' || in > 'Z')
        return in;

    // Encontrar posicion
    int index = in - 'A';
    NodoRotor* cur = head;
    for (int i = 0; i < index; ++i) {
        cur = cur->next;
    }
    return cur->c;
}
```

Complejidad: $O(k)$ donde k es la posición de la letra

4.3. Sistema de E/S: SerialReader

Abstrae la lectura de datos desde:

- Puerto serial real (Linux con termios)
- Archivo de texto para simulación

4.3.1. Características

- Detección automática del tipo de entrada
- Configuración de puerto serial (9600 baud, 8N1)
- Lectura línea por línea
- Eliminación automática de caracteres de control

4.4. Parser de Tramas

4.4.1. Función parseLinea

Convierte una línea de texto en un objeto trama:

```
TramaBase* parseLinea(const char* lineaC) {
    // Tokenizar por coma
    char* token = strtok(buffer, ",");

    // Trama LOAD
    if (token[0] == 'L' || token[0] == 'l') {
        char* arg = strtok(nullptr, ",");
        // Detectar "Space"
        if (strcasecmp(arg, "Space") == 0) {
            return new TramaLoad(' ');
        }
        return new TramaLoad(arg[0]);
    }

    // Trama MAP
    else if (token[0] == 'M' || token[0] == 'm') {
        char* arg = strtok(nullptr, ",");
        int n = atoi(arg);
        return new TramaMap(n);
    }

    return nullptr;
}
```

5. Análisis de Complejidad

5.1. Complejidad Temporal

Operación	Complejidad	Justificación
Insertar en lista carga	$O(1)$	Acceso directo a tail
Rotar rotor	$O(k)$	$k = N \bmod 26 \leq 26$
Mapear carácter	$O(k)$	$k \leq 26$
Procesar trama LOAD	$O(n)$	n = long. mensaje actual
Procesar trama MAP	$O(1)$	Rotación constante
Imprimir mensaje	$O(n)$	Recorrer lista completa

5.2. Complejidad Espacial

- **Lista de carga**: $O(m)$ donde m es el número de fragmentos
- **Rotor**: $O(26) = O(1)$ (tamaño constante)
- **Total**: $O(m)$ espacio lineal respecto al mensaje

5.3. Análisis de Caso Completo

Para un mensaje con:

- n tramas totales
- m tramas LOAD
- r tramas MAP

Complejidad total: $O(n \cdot m)$ en el peor caso, considerando que cada trama LOAD imprime el mensaje completo.

6. Ejemplo de Ejecución

6.1. Entrada de Ejemplo

```
L,H
L,O
L,L
M,2
L,A
L,Space
L,W
M,-2
L,O
L,R
L,L
L,D
```

6.2. Traza de Ejecución

Estado inicial:

- Lista carga: vacía
- Rotor: ABCDEFGHIJKLMNOPQRSTUVWXYZ (head en A)

Paso 1: L,H

- Mapeo: H $\rightarrow$ A + 7 = H
- Lista: [H]

Paso 2: L,O

- Mapeo: O → A + 14 = O
- Lista: [H][O]

Paso 3: L,L

- Mapeo: L → A + 11 = L
- Lista: [H][O][L]

Paso 4: M,2

- Rotar +2 posiciones
- Rotor: CDEFGHIJKLMNOPQRSTUVWXYZAB (head en C)

Paso 5: L,A

- Mapeo con rotor rotado: A → C + (-2) = A
- Lista: [H][O][L][A]

Resultado final: HOLA WORLD

7. Compilación y Uso

7.1. Requisitos

- Compilador C++ (g++ 7.0 o superior)
- CMake 3.10 o superior
- Sistema operativo Linux (para puerto serial)

7.2. Compilación

```
mkdir build
cd build
cmake ..
make
```

7.3. Ejecución

Modo simulación:

```
./prtdcd --sim entrada.txt
```

Modo serial:

```
./prtdcd --serial /dev/ttyUSB0
```

8. Pruebas y Validación

8.1. Casos de Prueba

8.1.1. Caso 1: Mensaje Simple

Entrada: L,A L,B L,C
Salida esperada: ABC
Resultado: Correcto

8.1.2. Caso 2: Con Rotaciones Positivas

Entrada: M,5 L,A L,B
Salida esperada: FG
Resultado: Correcto

8.1.3. Caso 3: Con Rotaciones Negativas

Entrada: M,-3 L,D L,E
Salida esperada: AB
Resultado: Correcto

8.1.4. Caso 4: Con Espacios

Entrada: L,H L,I L,Space L,T L,H
Salida esperada: HI TH
Resultado: Correcto

8.2. Manejo de Errores

- Tramas inválidas son ignoradas con mensaje informativo
- Archivos inexistentes generan error descriptivo
- Caracteres no alfabéticos se pasan sin modificación

9. Conclusiones

9.1. Logros

1. Implementación exitosa de estructuras de datos fundamentales sin STL
2. Aplicación correcta de polimorfismo y programación orientada a objetos
3. Sistema robusto de decodificación con manejo de errores
4. Documentación completa del código con estándar Doxygen

9.2. Aprendizajes

- Importancia de la gestión manual de memoria en C++
- Ventajas del polimorfismo para código extensible
- Utilidad de las listas circulares para operaciones de rotación
- Complejidad de la programación a bajo nivel de puertos seriales

9.3. Posibles Mejoras

1. Implementar validación de checksums en las tramas
2. Agregar soporte para múltiples rotores (máquina Enigma)
3. Optimizar impresión del mensaje (solo mostrar al final)
4. Agregar modo de depuración detallado
5. Implementar reconexión automática en caso de falla serial

10. Referencias

1. Cormen, T. H., et al. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.
3. Malik, D. S. (2017). *Data Structures Using C++* (2nd ed.). Course Technology.
4. POSIX.1-2008. *The Open Group Base Specifications*. IEEE.
5. Documentación Doxygen: `https://www.doxygen.nl/`

A. Código Fuente Completo

El código fuente completo del proyecto está disponible en el archivo `main.cpp` adjunto a este reporte.

A.1. Estructura de Archivos

```
proyecto/
 main.cpp
 CMakeLists.txt
 README.md
 entrada.txt (ejemplo)
 build/
     prtdcd (ejecutable)
```

B. Ejemplos Adicionales de Entrada

B.1. Ejemplo Complejo

```
M,3
L,H
L,E
M,-1
L,L
L,L
L,O
L,Space
M,5
L,W
L,O
M,-8
L,R
L,L
L,D
```