

# **Optimización y Gestión de Recursos en Microservicios C++**

Antonio Espín Herranz

# Contenidos

- Optimización del rendimiento en microservicios de alto rendimiento:
  - Uso de técnicas de **gestión de memoria** eficiente (malloc/free, RAII).
  - Minimización de la **latencia** en sistemas distribuidos con C++.
- **Apéndice:**
  - **Herramientas para Linux (no lo vemos)**
  - **Profiling y benchmarking** de microservicios:
    - Uso de herramientas como **valgrind**, **gprof** o **perf**.
    - Estrategias para mejorar la eficiencia del CPU y la memoria.

# Optimización de rendimiento en microservicios de alto rendimiento

# Gestión de memoria eficiente

## malloc / free / RAII

- **RAII** → Resource Acquisition Is Initialization
  - Adquisición de recursos en la inicialización es el patrón más seguro y moderno para manejar memoria, archivos o sockets.
  - Evita fugas de memoria y asegura la liberación completa de memoria.
  - Podemos utilizar:
    - `std::unique_ptr`, `std::vector`, `std::string` en lugar de las funciones de bajo nivel de C, como `malloc` / `calloc` / `realloc` y `free`.
    - O los operadores de C++ `new`, `new[]`, `delete`, `delete[]`

# malloc / free

- Uso manual de los recursos de la memoria
- Control extremos o integración en C
- La memoria hay que liberarla manualmente y es responsabilidad del programador, si no, se quedaran lagunas de memoria que a la larga producirán errores.
- En caso de tener que utilizarlo, pueden ser interesantes herramientas como **valgrind** que detecta e informa las **fugas** de memoria en un programa.

# Smart Pointers

- **unique\_ptr**: propiedad exclusiva del objeto.
- **make\_unique**: crea un unique\_ptr de forma segura.
- **shared\_ptr**: propiedad compartida entre varios punteros.
- **make\_shared**: crea un shared\_ptr de forma eficiente.

# Smart Pointers

- Se añadieron en **C++11** y representan un wrapper (envoltorio) de un puntero y gestionan su ciclo de vida.
- Tanto las funciones **malloc**, **calloc** y **realloc** de **C** y los operadores **new** y **delete** de **C++** son propensas para cometer errores de programación por no liberar la memoria adecuadamente. Para paliar estos problemas se añadieron los Smart pointers.
- Se encuentran en el fichero de cabecera: **memory**
  - `#include <memory>`
  - Básicamente tenemos 2 tipos: **unique\_ptr** y **shared\_ptr**

# Smart Pointers

- Se definen con tipos parametrizados en los cuales se puede utilizar tipos básicos de C, clases o tipos creados por el programador. Ya sean clases o estructuras.
- Los Smart pointers, a parte, de encargarse de la reserva de memoria se pueden comportar como los punteros raw. Añadiendo:
  - Se pueden establecer a **nullptr**.
  - Se puede utilizar: **\*ptr** para desreferenciar ptr.
  - Operador flecha para acceder a los métodos: **ptr-> método()**
- Adicionalmente añaden:
  - **ptr.get()**:
    - Retorna el puntero raw que gestiona por debajo.
  - **ptr.reset(raw\_ptr)**
    - Para la gestión del puntero, libera la memoria ocupada y establece ptr a ptr\_raw.



# Unique pointer (std::unique\_ptr)

- Sirve para construir un único puntero que libera automáticamente la memoria ocupada por este cuando deja de utilizarse.
- Suponiendo un tipo cualquiera llamado: **Type**

```
#include <memory>
```

```
// Utilizando el constructor por defecto de Type (se puede inferir el tipo con auto).
```

```
    auto p = std::unique_ptr<Type>(new Type);
```

```
// Utilizando el constructor con parámetros de Type:
```

```
    auto p = std::unique_ptr<Type>(new Type(<params>));
```

```
// También se puede utilizar la función: make_unique
```

```
    auto p = std::make_unique<Type>(<params>)
```

# Porque es único

- No tiene constructor copia.
- No puede ser copiado.
- Se puede mover.
- Garantiza que la memoria sea siempre propiedad de un único puntero.

# Ejemplo

```
#include <iostream >
```

```
#include <memory >
```

```
struct A {  
    int a = 10;  
};
```

```
int main() {  
    auto a_ptr = std::unique_ptr <A>(new A);  
    std::cout << a_ptr ->a << std::endl;  
    auto b_ptr = std::move(a_ptr);  
    std::cout << b_ptr ->a << std::endl;  
    return 0;  
}
```

# Consecuencias

- Pasar por parámetro un `unique_ptr` a una función trae consecuencias:
  - No se puede copiar, al pasarlo por parámetro se tiene que realizar una copia del puntero (y está eliminado el constructor copia).
  - Se tiene que mantener como único.
- Si necesitamos pasarlo como parámetro se puede hacer uso de la función **`std::move`** (mueve la memoria del puntero).

# Ejemplos

```
void funcion(std::unique_ptr<Clase> arg){  
    //Contenido de la función  
}
```

```
int main(){  
    std::unique_ptr<Clase> instancia {new clase};  
    // Da error:  
    funcion(instancia);  
  
    // Podemos utilizar:  
    funcion(std::move(instancia));  
}
```

# Shared pointer (`std::shared_ptr`)

- Dispone de un constructor como `unique_ptr`
- Se puede copiar.
- Almacena un contador interno y un puntero raw.
  - Incrementa el contador en 1, cuando se copia.
  - Decrementa el contador en 1 cuando se destruye.
- Libera la memoria cuando el contador es 0.
- Se pueden inicializar con un `unique_ptr`.

# Ejemplo

```
#include <memory >
```

```
// Utilizando el constructor por defecto: Type();
```

```
auto p = std:: shared_ptr <Type >(new Type);
```

```
auto p = std:: make_shared <Type >();
```

```
// Utilizando el constructor con parámetros: Type(<params >);
```

```
auto p = std:: shared_ptr <Type >(new Type(<params >));
```

```
auto p = std:: make_shared <Type >(<params >);
```

```
#include <iostream >
#include <memory >

struct A {
    A(int a) { std::cout << "Se crea el objeto...\n"; }
    ~A() { std::cout << "Se elimina el objeto...\n"; }
};

int main() {
    // Equivalente a: std::shared_ptr <A>(new A(10));
    auto a_ptr = std::make_shared <A >(10);
    std::cout << a_ptr.use_count () << std::endl;
    {
        auto b_ptr = a_ptr;
        std::cout << a_ptr.use_count () << std::endl;
    }
    std::cout << "En el scope de main\n";
    std::cout << a_ptr.use_count () << std::endl;
    return 0;
}
```



# Cuando utilizarlos

- Utilizar punteros inteligentes para administrar la memoria dinámica.
- Por defecto, utilizar **unique\_ptr**.
- Si hay varios objetos que deben compartir la propiedad, utilizar **shared\_ptr**.
- El uso de punteros inteligentes evita tener destructores en las clases.
- Se aconseja utilizar las funciones `make_unique` y `make_shared`, si no, inicializar con un puntero raw.

# Ejemplo

- En el caso de llamar a una función un puntero **shared\_ptr** se puede copiar el puntero (por debajo se llama al constructor copia y contador interno del puntero se incrementa en uno)

```
void funcion2(std::shared_ptr<Alertador> arg){  
    std::cout << "Se ejecuta la función 2: count: " << arg.use_count() << "\n";  
  
}
```

```
int main(){  
    std::shared_ptr<Alertador> ps {new Alertador};  
    funcion2(ps);  
}
```

# Situaciones

- Cuando utilizamos un **shared\_ptr** en una serie de ámbitos anidados y que apuntan a la misma instancia (el contador interno del puntero cada vez que termina un ámbito se libera un puntero (y se descuenta el contador)) Cuando el contador llega a 0 se libera el recurso y se llama al destructor.

# Errores habituales


```
#include <iostream >
#include <memory >
int main() {
    int a = 0;
    // Hay que inicializar con memoria del heap.
    auto a_ptr = std::unique_ptr<int>(&a);
    // Se producirá un error al intentar liberar.
    return 0;
}
```

```
#include <iostream >
#include <vector >
#include <memory >
using std::cout; using std:: unique_ptr;
struct AbstractShape { // Structs to save space.
    virtual void Print () const = 0;
};
struct Square : public AbstractShape {
    void Print () const override { cout << "Square\n"; }
};
struct Triangle : public AbstractShape {
    void Print () const override { cout << "Triangle\n"; }
};
int main() {
    std::vector <unique_ptr <AbstractShape >> shapes;
    shapes.emplace_back(new Square);
    auto triangle = unique_ptr <Triangle >(new Triangle);
    shapes.emplace_back(std::move(triangle));
    for (const auto& shape : shapes) { shape ->Print (); }
    return 0;
}
```

# Ejemplo para arrays

**Para compilar: -std=c++14**

```
unique_ptr array = make_unique<int []>(N);
```



**N** indica el número de posiciones del Array.

```
for (int i = 0 ; i < N ; i++)  
    array[i] = i+1;
```

```
for (int i = 0 ; i < N ; i++)  
    cout << array[i] << ", ";  
cout << endl;
```

# Clase Punto

```
class Punto {  
  
    friend ostream & operator<<(ostream &os, const Punto &p){  
        return os << "[x = " << p.x << ", y = " << p.y << "];"  
    }  
    int x, y;  
  
    public:  
        Punto(int _x=0,int _y=0):x(_x),y(_y){}  
        int getX(){ return this->x; }  
        int getY(){ return this->y; }  
  
};
```

# Ejemplo para arrays

```
void test_array_Puntos(){  
    cout << endl << "TEST ARRAY DE PUNTOS:" << endl;  
    unique_ptr array = make_unique<Punto []>(N);  
  
    for (int i = 0 ; i < N ; i++)  
        cout << array[i] << endl;  
    cout << endl;  
}
```



# Ejemplo: Con un objeto

```
void test_ptr_Punto(){  
    cout << endl << "TEST PUNTERO A OBJETO PUNTO:" << endl;  
    unique_ptr<Punto> ppunto = make_unique<Punto>(5,-8);  
    cout << "X: " << ppunto->getX() << endl;  
    cout << "Y: " << ppunto->getY() << endl;  
  
    cout << "Punto: " << *ppunto << endl;  
}
```

# reset

- Se puede llamar al método **reset** y se va a liberar el **shared\_ptr**.

```
int main(){  
    std::shared_ptr<Clase> instancia {new Clase};  
    instancia.reset(); → Salta el destructor!  
    return 0;  
}
```

# make\_shared

- Hay una función **más eficiente** que es **make\_shared**.
  - `std::shared_ptr<Clase> instancia = std::make_shared<Clase>();`
- Tanto con `shared_ptr` como con `unique_ptr` se puede utilizar `auto` en las declaraciones:
  - `auto ps3 = std::make_shared<Clase>();`

# Notas

- **No se aconseja utilizar `unique_ptr` para crear una matriz**
- Y `make_shared<int []>` No está soportado. Incluso ni en C++17
- Se aconseja utilizar **`vector<vector<int>>`** o la clase **`array`** de C++

# Minimización de la latencia en sistemas distribuidos

- **Serialización rápida**

- nlohmann/json es cómodo de usar y fácil pero no es de las opciones más rápidas para grandes volúmenes.
- Disponemos de formatos binarios como MessagePack, CBOR o FlatBuffers.

# Minimización de la latencia en sistemas distribuidos

- **Comunicación eficiente:**
  - Utilizar **HTTP/2** o **gRPC**
  - Minimizar el tamaño de las respuestas y evitar redirecciones.
- **Concurrencia y paralelismo**
  - Crow permite utilización de varios: con **.multithreaded()** pero las estructuras que estén compartidas tienen que estar protegidas.

# Minimización de la latencia en sistemas distribuidos

- **Caching inteligente:**

- Cachear los resultados de las consultas frecuentes en memoria (`std::unordered_map`).
- Evitar llamadas repetidas a base de datos o servicios externos.

- **Lazy loading** y procesamiento diferido:

- Procesar solo lo necesario en cada solicitud

# Ejemplo de caché en memoria

- Una posible colección para la implementación puede ser: **std::unordered\_map** protegida con un **mutex**.
- #include <unordered\_map>
- #include <string>
- #include <mutex>
- std::unordered\_map<std::string, std::string> cache;
- std::mutex cache\_mutex;



# Ejemplo de caché en memoria

```
std::string get_from_cache(const std::string& key) {  
    std::lock_guard<std::mutex> lock(cache_mutex);  
    auto it = cache.find(key);  
    return (it != cache.end()) ? it->second : "";  
}
```

```
void set_in_cache(const std::string& key, const std::string& value) {  
    std::lock_guard<std::mutex> lock(cache_mutex);  
    cache[key] = value;  
}
```

# Ejemplo de caché en memoria

```
CROW_ROUTE(app, "/datos/<string>")
([](const crow::request& req, const std::string& clave){
    std::string resultado = get_from_cache(clave);
    if (!resultado.empty()) {
        CROW_LOG_INFO << "Respuesta desde caché";
        return crow::response(200, resultado);
    }

    // Simular cálculo o consulta costosa
    std::string nuevo_resultado = "Resultado para " + clave;

    set_in_cache(clave, nuevo_resultado);
    return crow::response(200, nuevo_resultado);
});
```

# Consideraciones

- Implementar una clase con una plantilla que nos sirva la caché para distintos tipos de objetos.

```
template<typename T>
class MemCache {
    private:
        std::unordered_map<std::string, T> data_;
        std::mutex mutex_; // Para el acceso concurrente a la colección
}
```

# **APÉNDICE**

## **Profiling y benchmarking de microservicios**

# Profiling

- Análisis del rendimiento, es una técnica que permite estudiar el comportamiento de un programa mientras se ejecuta.
- Permite identificar cuellos de botella, optimizar recursos y mejorar el rendimiento general.

# ¿Qué es el profiling?

- Es el proceso de **medir y analizar** aspectos internos de una aplicación como:
  - Tiempo de ejecución por función o método
  - Uso de CPU y memoria
  - Accesos a disco o red
  - Frecuencia de llamadas
  - Fugas de memoria o ciclos innecesarios
- El resultado es un **perfil de ejecución**, que te muestra qué partes del código consumen más recursos o tiempo.

# ¿Para qué sirve?

- Optimizar código crítico: Saber qué funciones ralentizan el sistema
- Reducir consumo de recursos: CPU, RAM, disco
- Detectar fugas de memoria
- Mejorar escalabilidad: Preparar el sistema para más carga
- Tomar decisiones informadas: Qué refactorizar, qué paralelizar, qué cachear

# Benchmarking

- Es una técnica que consiste en **medir, comparar y analizar el rendimiento** de un sistema, proceso o componente frente a otros similares, con el objetivo de identificar oportunidades de mejora y adoptar las mejores prácticas.
- **Consiste en:**
  - Ejecutar pruebas de rendimiento sobre un programa, función o sistema
  - Medir métricas como tiempo de ejecución, uso de CPU, memoria, latencia, etc.
  - Comparar esos resultados con otras versiones, competidores o estándares



# Tipos de benchmarking técnico

Tipo	Descripción breve
<b>Sintético</b>	Usa pruebas artificiales para estresar componentes
<b>Aplicativo</b>	Usa cargas reales simuladas (ej. peticiones HTTP)
<b>Comparativo</b>	Compara tu sistema con otros (competencia o versiones)
<b>Interno</b>	Compara entre áreas o módulos dentro de tu propio sistema

# ¿Para que sirve?

- Detectar cuellos de botella
- Validar mejoras de rendimiento
- Justificar decisiones tecnológicas
- Optimizar recursos (CPU, RAM, red)
- Aumentar la escalabilidad

# Herramientas Linux

Herramienta	Función principal
<code>perf</code>	Análisis de CPU, eventos, y llamadas
<code>Valgrind</code>	Fugas de memoria, profiling básico
<code>gprof</code>	Perfil por función (requiere recompilación)
<code>Callgrind</code> + <code>KCachegrind</code>	Visualización de llamadas
<code>strace</code>	Trazado de llamadas al sistema

# Herramientas Windows

Herramienta	Función principal
Visual Studio Profiler	Integrado, muy completo
Windows Performance Analyzer (WPA)	Análisis profundo del sistema
Intel VTune	Avanzado, multiplataforma
Process Explorer	Monitoreo en tiempo real
Very Sleepy	Profiling ligero por función

- En Windows podemos instalar los comandos de Linux en WSL (ojo utilizar **WSL 2**).

# Herramientas

- **valgrind**

- Detecta fugas de memoria, errores de acceso, y profiling básico.
- <https://valgrind.org/>

- **gprof**

- Genera perfiles de tiempo de ejecución por función

- **perf** (Linux)

- Herramientas de bajo nivel para análisis de CPU y eventos.

# valgrind: instalación en WSL2

```
sudo apt update
```

```
sudo apt install -y build-essential make g++ wget
```

```
wget https://sourceware.org/pub/valgrind/valgrind-3.25.1.tar.bz2
```

```
tar -xvjf valgrind-3.25.1.tar.bz2
```

```
cd valgrind-3.25.1
```

```
./configure
```

```
make
```

```
sudo make install
```

# valgrind: instalación en WSL2

- Una vez instalado, añadir esta línea al fichero: **.bashrc**
  - **export PATH=\$PATH:/usr/local/bin**
- **Prueba:**
  - **valgrind --version**
  - **Debería responder:**
  - **valgrind-3.25.1**

# valgrind

- Podemos **detectar**:
  - Fugas de memoria en las clases que utilizan memoria dinámica
  - Accesos inválidos a memoria.
  - Uso incorrecto de new / delete o malloc / free
  - Consumo excesivo del Heap
- **Aplicarlo**:
  - `valgrind --tool=memcheck ./microservicio`



# gprof: instalación en WSL2

- **sudo apt update**
- **sudo apt install binutils**
- **gprof --version**
  - GNU gprof (GNU Binutils for Ubuntu) 2.38
  - Based on BSD gprof, copyright 1983 Regents of the University of California.
  - This program is free software. This program has absolutely no warranty.

# gprof

- Permite detectar:
  - Qué funciones consumen más tiempo
  - Cuántas veces se llaman
  - Relación entre funciones (quien llama a quien)
- **Aplicarlo:**
  - Compilar con la opción `-pg`:
    - `g++ -pg -O2 main.cpp -o microservicio`
  - Ejecutar el programa normalmente:
    - `./microservicio`
  - Generar el perfil:
    - `gprof microservicio gmon.out > perfil.txt`

# perf: instalación en WSL2

```
sudo apt update
```

```
sudo apt install -y build-essential flex bison libdwarf-dev libelf-dev libnuma-dev libunwind-dev \
libnewt-dev libdw-dev libssl-dev libperl-dev python-dev-is-python3 binutils-dev libiberty-dev \
libzstd-dev libcap-dev libbabeltrace-dev git
```

```
git clone https://github.com/microsoft/WSL2-Linux-Kernel --depth 1
cd WSL2-Linux-Kernel/tools/perf
```

```
sudo apt install flex bison libelf-dev python3 libtraceevent-dev
```

```
sudo ln -s /usr/bin/python3 /usr/bin/Python
```

```
make NO_LIBTRACEEVENT=1 NO_JEVENTS=1 -j$(nproc)
```

```
sudo cp ./tools/perf/perf /usr/local/bin/perf
```

**perf --version**

# perf

- Permite **detectar**:
  - Uso de CPU por función o hilo
  - Llamadas al sistema (syscalls)
  - Latencia en operaciones de red o disco
  - Contención entre hilos.
- Aplicarlo:
  - **perf** record ./microservicio
  - **perf** report

# Relación entre las 3 herramientas

- **PASOS:**

- 1. Simular carga con **wrk** o **curl**.
- 2. Ejecutas **valgrind** para detectar fugas.
- 3. Usas **gprof** para ver qué funciones dominan el tiempo.
- 4. Usas **perf** para ver si hay contención o latencia en llamadas externas.

# Recompilar

- Para utilizar estas 3 herramientas, tenemos que recompilar los fuentes con el compilador de g++ para poder utilizar las herramientas.