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.

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
Sintético	Usa pruebas artificiales para estresar componentes
Aplicativo	Usa cargas reales simuladas (ej. peticiones HTTP)
Comparativo	Compara tu sistema con otros (competencia o versiones)
Interno	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
`perf`	Análisis de CPU, eventos, y llamadas
`Valgrind`	Fugas de memoria, profiling básico
`gprof`	Perfil por función (requiere recompilación)
`Callgrind` + `KCachegrind`	Visualización de llamadas
`strace`	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.