

# **Comunicación entre Microservicios C++**

Antonio Espín Herranz

# Contenidos

- Implementación de patrones de mensajería:
  - Uso de **ZeroMQ**, **gRPC** y **RabbitMQ** para comunicación asincrónica.
  - Comparativa entre **REST** y **gRPC** en sistemas de alto rendimiento.
  - Implementación de **WebSockets** para la comunicación en tiempo real.
- Gestión de la **conurrencia** y el **multithreading**:
  - Uso de **std::thread** y **boost::asio** para manejar múltiples solicitudes.
  - Estrategias de sincronización y control de acceso concurrente

# **Patrones de mensajería**

# Comunicaciones

- **ZeroMQ:**

- **No necesita broker:** los procesos se comunican directamente.
- Extremadamente **rápido y ligero**, usado en sistemas financieros y embebidos.
- Menos soporte para persistencia o reintentos automáticos; lo gestionamos todo nosotros.

- **gRPC:**

- Diseñado por **Google** para llamadas RPC eficientes.
- Usa HTTP/2 y **protobuf** para máxima velocidad y eficiencia.
- Perfecto para APIs modernas y microservicios con contratos bien definidos.
- Más enfocado a comunicación síncrona, pero se puede utilizar en asíncrono.

- **RabbitMQ:**

- Ideal para sistemas desacoplados donde los servicios se comunican mediante colas.
- Ofrece persistencia, reintentos, y enrutamiento avanzado.
- Requiere un **broker (servidor RabbitMQ)** que puede ser un punto de fallo si no se gestiona bien.

# Cuando seleccionar

- **RabbitMQ**: proporciona colas, desacoplamiento, y tolerancia a fallos.
- **ZeroMQ**: velocidad extrema y control total sobre la comunicación.
- **gRPC**: API rápida, segura y bien estructurada entre servicios.

# ZeroMQ

# ZeroMQ

- Es una **librería de mensajería ultrarrápida y asíncrona** que permite construir sistemas distribuidos, escalables y concurrentes.
- A diferencia de los sistemas tradicionales de colas de mensajes como RabbitMQ o Kafka, **ZeroMQ no necesita un servidor intermedio**: los procesos se comunican directamente entre sí.
- <https://zguide.zeromq.org/docs>

# Instalación

- `vcpkg install zeromq cppzmq`
- `vcpkg integrate install`
- Con esto Visual Studio ya detectará la librería en los proyectos:
- **`#include <zmq.hpp>`**



# ZeroMQ

- Proporciona **sockets inteligentes** que pueden manejar múltiples patrones de comunicación:
  - **pub-sub** (publicador-suscriptor)
  - **req-rep** (petición-respuesta)
  - **push-pull** (pipeline)
  - **dealer-router** (para patrones más complejos)
- Soporta múltiples **protocolos de transporte**:
  - TCP
  - IPC (comunicación entre procesos)
  - Inproc (dentro del mismo proceso)
  - Multicast

# ZeroMQ

- **Ventajas clave**
  - **Velocidad extrema:** diseñado para alto rendimiento y baja latencia.
  - **Ligero y sin servidor:** no requiere broker central.
  - **Multilenguaje:** disponible en C++, Python, Go, Java, Rust, entre otros.
  - **Flexible:** ideal para arquitecturas de microservicios, sistemas embebidos y telecomunicaciones.

# Casos de uso típicos

Sector	Ejemplo de uso con ZeroMQ
Sistemas embebidos	Comunicación entre sensores y controladores
Telecomunicaciones	Enrutamiento de paquetes y señalización
Trading financiero	Difusión de precios en tiempo real
Robótica	Coordinación entre módulos de percepción y control
Microservicios	Comunicación entre servicios sin necesidad de HTTP





# Uso de ZeroMQ en microservicios

Uso común	Descripción
Comunicación entre servicios	Reemplaza HTTP/REST con mensajería binaria más rápida.
Procesamiento distribuido	Divide tareas entre múltiples workers con <code>PUSH/PULL</code> .
Difusión de eventos	Usa <code>PUB/SUB</code> para enviar actualizaciones a múltiples servicios.
Sistemas embebidos o IoT	Ideal por su bajo overhead y soporte para múltiples protocolos.
Integración con Boost.Asio	Con librerías como <code>azmq</code> puedes combinar ZeroMQ con programación asíncrona <sup>1</sup> .

# Tener en cuenta

- No ofrece persistencia de mensajes por defecto (no es un sistema de colas tradicional).
- Requiere que el desarrollador gestione la **topología de red y la fiabilidad**.
- Es más, una **caja de herramientas** que una solución lista para usar.

# Patrones ZeroMQ

Patrón	Sincronía	Descripción breve
REQ/REP	 Síncrono	Ciclo estricto: enviar → esperar respuesta
PUSH/PULL	 Asíncrono	Distribuye tareas sin esperar respuesta
PUB/SUB	 Asíncrono	Difunde mensajes a múltiples receptores
ROUTER/DEALER	 Asíncrono	Comunicación flexible, no bloqueante, ideal para servidores concurrentes

# Patrón síncrono

- **REQ-REP** (request – reply)
- **Ciclo de comunicación**
  - **Cliente (REQ):**
    - Envía un mensaje con `send()`
    - Se bloquea esperando respuesta con `recv()`
  - **Servidor (REP):**
    - Recibe el mensaje con `recv()`
    - Procesa y responde con `send()`
  - **Cliente (REQ):**
    - Recibe la respuesta
    - Puede enviar otro mensaje (solo después de recibir)

# Test Servidor

```
#include <zmq.hpp>
#include <string>
#include <iostream>

int main() {
    zmq::context_t context(1); // Entorno de ejecución para zeromq, el número 1 indica la cantidad de hilos. 4 → 4 hilos
    zmq::socket_t socket(context, zmq::socket_type::rep);
    socket.bind("tcp://*:5555"); // Con el * acepta conexiones desde cualquier interface de red. Cualquier IP por el puerto 5555

    while (true) {
        zmq::message_t request;
        socket.recv(request, zmq::recv_flags::none);
        std::cout << "Recibido: " << request.to_string() << std::endl;

        std::string reply = "Hola desde el servidor";
        socket.send(zmq::buffer(reply), zmq::send_flags::none);
    }
}
```

**socket\_t** declara el socket  
**socket\_ref** para pasar el socket por parámetro



# Test Cliente

```
#include <zmq.hpp>
#include <string>
#include <iostream>

int main() {
    zmq::context_t context(1);
    zmq::socket_t socket(context, zmq::socket_type::req);
    socket.connect("tcp://localhost:5555");

    std::string mensaje = "Hola servidor";
    socket.send(zmq::buffer(mensaje), zmq::send_flags::none);

    zmq::message_t respuesta;
    socket.recv(respuesta, zmq::recv_flags::none);
    std::cout << "Respuesta: " << respuesta.to_string() << std::endl;
}
```

# Flags para send

- A la hora de enviar un mensaje, se pueden indicar flags. Por ejemplo, para enviar un mensaje en partes.

Flag	Descripción
<code>zmq::send_flags::none</code>	Envío estándar sin opciones especiales.
<code>zmq::send_flags::dontwait</code>	No bloquea si el socket no está listo para enviar. Si no puede enviar, lanza una excepción.
<code>zmq::send_flags::sndmore</code>	Indica que <b>hay más partes del mensaje</b> por enviar. Se usa para mensajes multipartes.

```
socket.send(zmq::buffer("parte1"), zmq::send_flags::sndmore);  
socket.send(zmq::buffer("parte2"), zmq::send_flags::none); // última parte
```

# Flags para recv

- A la hora de recibir mensajes también se puede indicar flags.

Flag	Descripción
<code>zmq::recv_flags::none</code>	Lectura estándar, bloquea hasta recibir.
<code>zmq::recv_flags::dontwait</code>	No bloquea si no hay mensaje disponible. Si no hay datos, lanza excepción o retorna false.

```
Recepción no bloqueante:
zmq::message_t msg;
if (socket.recv(msg, zmq::recv_flags::dontwait)) {
    std::cout << "Recibido: " << msg.to_string() << std::endl;
} else {
    std::cout << "No hay mensaje disponible." << std::endl;
}
```

# Integración con microservicios

- **Define el patrón de comunicación:**
  - **REQ/REP** (Request/Reply): para llamadas tipo cliente-servidor.
  - **PUB/SUB** (Publish/Subscribe): para difusión de eventos.
  - **PUSH/PULL**: para procesamiento en paralelo o distribución de tareas.
  - **ROUTER/DEALER**: para sistemas más complejos con múltiples clientes y servidores.
- Se pueden crear sockets en los microservicios y se conecta o vincula a una dirección.
  - Un microservicio de autenticación puede usar REP para responder a solicitudes de login.
- Enviar y recibir mensajes
  - **Puede ser una cadena o una estructura en json.**
  - **ZeroMQ gestiona la cola de mensajes y la reconexión**

Al crear el socket se indica el tipo:  
rep, req, pull, etc

# Patrones asíncronos

- **PUSH/PULL**

- El **PUSH** envía mensajes sin esperar respuesta.
- El **PULL** recibe mensajes cuando llegan.
- Ideal para distribuir trabajo entre varios workers.

- **PUB/SUB**

- El **PUB** publica mensajes.
- El **SUB** recibe mensajes si está suscrito.
- No hay acuse de recibo ni respuesta directa.

- **ROUTER/DEALER**

- Más avanzado: permite múltiples clientes y servidores.
- No bloquea como REQ/REP.
- Puedes enviar y recibir en cualquier orden.
- Ideal para sistemas concurrentes o balanceadores de carga.

# Comunicación Asíncrona

- El **emisor** puede enviar mensajes sin esperar respuesta inmediata.
- El **receptor** puede procesar mensajes cuando esté listo, sin bloquear el flujo.
- ZeroMQ gestiona internamente las **colas, buffers y reconexiones**.
- Patrones:
  - PUSH/PULL Distribuye tareas a múltiples workers
  - PUB/SUB Difunde mensajes a múltiples suscriptores
  - ROUTER/DEALER Comunicación flexible entre múltiples clientes y servidores.
  - *REQ/REP* *Bloqueante (síncrono)*

# Ejemplo Worker (PULL)

```
#include <zmq.hpp>
#include <string>
#include <iostream>

int main() {
    zmq::context_t contexto(1);
    zmq::socket_t receiver(contexto, zmq::socket_type::pull);

    std::cout << "Preparado para recibir datos ..." << std::endl;

    receiver.connect("tcp://localhost:5557");
    while (true) {
        zmq::message_t msg;

        receiver.recv(msg, zmq::recv_flags::none);
        std::cout << "Procesando: " << msg.to_string() << std::endl;
    }

    return 0;
}
```

# Ejemplo Dispatcher (PUSH)

```
#include <zmq.hpp>
#include <string>
#include <iostream>

int main() {
    zmq::context_t contexto(1);
    zmq::socket_t sender(contexto, zmq::socket_type::push);

    std::cout << "Preparado para enviar tareas ... " << std::endl;

    sender.bind("tcp://*:5557");
    for (int i = 0; i < 10; i++) {
        sender.send(zmq::buffer("Tarea: " + std::to_string(i + 1)), zmq::send_flags::none);
    }

    return 0;
}
```



# Patrón PUB / SUB

- Patrón **PUB/SUB** permite que un **emisor (publisher)** envíe mensajes a múltiples **receptores (subscribers)** sin saber quiénes son ni cuántos hay.
- **Características clave**
  - **Asíncrono:** el publisher no espera respuesta.
  - **Difusión:** todos los subscribers que estén conectados y suscritos reciben el mensaje.
  - **Filtrado:** los **subscribers** pueden filtrar por temas (**topics**).
- **Ideal para...**
  - Eventos en tiempo real
  - Notificaciones
  - Sistemas de logging
  - Sensores que transmiten datos

El filtro se establece a nivel de cadena  
Que empiece por: ...

# Patrón ROUTER / DEALER

- **ROUTER:**

- Actúa como un servidor inteligente que puede recibir mensajes de múltiples clientes y responder a cada uno individualmente.
- Mantiene el ID de cada cliente.

- **DEALER:**

- Es un cliente avanzado que puede enviar mensajes sin esperar respuesta inmediata.
- Ideal para comunicación asíncrona.

# Patrón ROUTER / DEALER

- **1. Asimetría de roles**

- **ROUTER:** maneja múltiples clientes, conoce sus identidades, y puede enviar mensajes dirigidos.
- **DEALER:** actúa como cliente, no necesita saber a quién habla, y puede enviar/recibir sin bloqueo.
- Esto lo hace ideal para servidores concurrentes y balanceadores de carga.

# Patrón ROUTER / DEALER

- **2. Frames y protocolo interno**

- El protocolo ROUTER-DEALER **no impone una estructura fija**, pero por convención:
  - El ROUTER espera: [identity] [empty frame] [payload]
  - El DEALER envía: [payload], y ZeroMQ **agrega automáticamente** los frames si hay routing\_id.
- **Pero si controlas ambos extremos,**
  - [identity] [payload]

# Patrón ROUTER / DEALER

- **3. Enrutamiento explícito**

- El ROUTER **no decide a quién responder automáticamente.**
- Debes usar el identity recibido para responder correctamente:
  - `router.send(zmq::buffer(client_id), zmq::send_flags::sndmore);`
  - `router.send(zmq::buffer("respuesta"), zmq::send_flags::none);`

- **4. No hay ciclo estricto**

- A diferencia de REQ-REP, el patrón ROUTER-DEALER es **asíncrono**:
- Puedes enviar múltiples mensajes sin esperar respuesta.
- Puedes recibir múltiples mensajes antes de responder.
- Ideal para sistemas con múltiples clientes y lógica de procesamiento más compleja.

# Patrón ROUTER / DEALER

- **5. Identidad como clave**

- La **routing\_id** que asignas al DEALER es **clave** para que el ROUTER sepa a quién responder.
- Si no la asignas, **ZeroMQ** genera una identidad aleatoria.

# gRPC

Los proyectos con C++ 17

# gRPC

- **Google Remote Procedure Call**
- Es un framework de comunicación de alto rendimiento y código abierto que permite a aplicaciones intercambiar datos entre sí de forma eficiente, rápida y estructurada.
- Fue desarrollado por Google y se basa en el protocolo **HTTP/2** y en **Protocol Buffers (protobuf)** para la serialización de datos.



# Introducción

- **Streaming bidireccional**

- gRPC permite:
- **Streaming del servidor:** el servidor envía múltiples respuestas.
- **Streaming del cliente:** el cliente envía múltiples peticiones.
- **Streaming bidireccional:** ambos envían y reciben datos en tiempo real.
- Ideal para chats, dashboards en vivo, sensores, etc.

- **Interoperabilidad entre lenguajes**

- Puedes tener:
- Un servidor en C++
- Un cliente en Python, Go, JavaScript, etc.
- Todo gracias a que comparten el mismo **.proto**.

# gRPC

- Permite definir **servicios** y sus métodos usando archivos **.proto**.
- Un archivo **proto** es un **IDL** (lenguaje de definición de interface) y después se puede **generar código** en C++
- Genera automáticamente el código cliente y servidor en múltiples lenguajes (C++, Go, Java, Python, etc.).
- Usa **llamadas a procedimientos remotos (RPC)** para que una aplicación pueda ejecutar funciones en otra como si fueran locales.

# gRPC: Ventajas

- **Comunicación eficiente**
  - Usa **HTTP/2**, lo que permite multiplexación de conexiones, compresión de cabeceras y streaming bidireccional.
- **Serialización rápida**
  - Utiliza **Protocol Buffers**, que son más compactos y rápidos que JSON o XML.
- **Multilenguaje**
  - Compatible con muchos lenguajes: ideal para arquitecturas de microservicios heterogéneas.
- **Seguridad**
  - Soporta **TLS** para cifrado de extremo a extremo.

# Casos de uso típicos

Sector	Ejemplo de uso con gRPC
Microservicios	Comunicación entre servicios backend
Telecomunicaciones	Transmisión de datos entre nodos de red
IoT / sistemas embebidos	Comunicación eficiente entre dispositivos
Juegos online	Sincronización de estado entre cliente y servidor
Machine Learning	Servir modelos de IA con baja latencia

# Componentes necesarios

- **protoc** es un **compilador** que tomará el archivo **proto** como entrada y generará las estructuras de mensajes y las interfaces de servicio en el lenguaje deseado, en nuestro caso, C++.
- **El archivo .proto** es un archivo **IDL** donde definimos las estructuras de paso de mensajes y las API de rpc de acuerdo con las especificaciones de **Protobuf**.
- **Complemento gRPC** para el lenguaje específico:
  - Existen complementos gRPC separados para diferentes lenguajes de programación y el protocolo los requerirá para generar los archivos fuente según las interfaces de servicio definidas en el IDL.

# Instalación

- Se instala con el gestor de paquetes: **vcpkg**
  - **vcpkg install grpc**
  - **vcpkg integrate install**
- Visual Studio debería de reconocer:  
#include <grpcpp/grpcpp.h>  
#include <grpcpp/server.h>  
#include <grpcpp/server\_builder.h>
- Para la compilación de archivos **.proto**
- Necesitamos instalar:
  - **vcpkg install protobuf**
  - **vcpkg integrate install**

# Instalación

- Dentro de la instalación de **vcpkg** tenemos que localizar el comando **protoc** y añadirlo al **PATH**.
- Por ejemplo:
  - C:\vcpkg\installed\x64-windows\tools\protobuf\**protoc.exe**
- Hay un plugin **grpc\_cpp\_plugin.exe** que se utiliza para generar código C++, y debería de estar también en:
  - C:\vcpkg\installed\x64-windows\tools\grpc\**grpc\_cpp\_plugin.exe**
  - Si no se instaló correctamente lanzar estos comandos:
    - **vcpkg remove grpc**
    - **vcpkg install grpc --recurse**
    - **vcpkg integrate install**

# Pasos para desarrollar con gRPC cliente / servidor para enviar un mensaje

- **1. Definir el servicio con Protocols Buffers**
- Crea un archivo .proto que describa:
  - El servicio (por ejemplo, SaludoService)
  - Los métodos (por ejemplo, DiHola)
  - Los mensajes de entrada y salida (por ejemplo, SolicitudSaludo, RespuestaSaludo)
- Se puede ubicar en una carpeta de la solución aparte.



# Pasos para desarrollar con gRPC cliente / servidor para enviar un mensaje

- **2. Generar el código fuente desde el .proto**
- Usa protoc junto con el plugin de gRPC para generar:
  - Archivos .pb.h y .pb.cc (mensajes)
  - Archivos .grpc.pb.h y .grpc.pb.cc (servicio gRPC)
- Esto se hace en la línea de comandos que invoca el compilador de protobuf.

# Pasos para desarrollar con gRPC cliente / servidor para enviar un mensaje

- **3. Implementar el servidor**
- Crea un archivo fuente para el servidor.
- Incluye:
  - La implementación de la clase del servicio (hereda de la interfaz generada)
  - La lógica del método (por ejemplo, devolver “Hola, Juan”)
  - La configuración del servidor (puerto, credenciales, etc.)

# Pasos para desarrollar con gRPC cliente / servidor para enviar un mensaje

- **4. Implementar el cliente**
- Crea un archivo fuente para el cliente.
- Incluye:
  - La creación del stub (cliente gRPC)
  - La construcción del mensaje de solicitud
  - La llamada al método remoto
  - La recepción y manejo de la respuesta

# Pasos para desarrollar con gRPC cliente / servidor para enviar un mensaje

- **5. Compilar el proyecto**
- Usa CMake o Visual Studio para compilar:
  - Los archivos generados por protoc
  - El servidor y el cliente
  - Las dependencias de gRPC y protobuf
- **6. Ejecutar**
- Primero ejecuta el **servidor** para que escuche en el puerto.
- Luego ejecuta el **cliente** para enviar la solicitud y recibir la respuesta.

# ¿Cómo se usa?

- El servicio se define en un archivo **.proto**

```
service Saludo {  
  rpc DiHola (Mensaje) returns (Respuesta);  
}
```
- Se compila con **protoc**
  - Implementar el servidor y el cliente en el lenguaje elegido.
- gRPC genera código a partir de contratos.
- El archivo **.proto** define la estructura y servicios y **protoc genera la parte de comunicación.**
- La lógica de que hace el cliente y el servidor la escribimos nosotros.

# Estructura básica del archivo: **saludo.proto**

```
syntax = "proto3";           // Versión del lenguaje de Protobuf  
package saludo;              // Nombre del paquete (opcional pero recomendable)
```

```
service Saludo {             // Definición del servicio gRPC  
  rpc DiHola (Solicitud) returns (Respuesta); // Método RPC  
}
```

```
message Solicitud {          // Mensaje de entrada  
  string nombre = 1;  
}
```

```
message Respuesta {         // Mensaje de salida  
  string mensaje = 1;  
}
```

# Generar código

**protoc**

**--cpp\_out=.**

**--grpc\_out=.**

**--plugin=protoc-gen-grpc=D:\vcpkg\installed\x64-windows\tools\grpc\grpc\_cpp\_plugin.exe**

**saludo.proto**

- El comando genera:
  - **saludo.pb.h** y **saludo.pb.cc**: clases para los mensajes (Solicitud, Respuesta)
  - **saludo.grpc.pb.h** y **saludo.grpc.pb.cc**: clases para el servicio (**Saludo::Service**, **Saludo::Stub**)

# Escribir el servidor

- Crea una clase que herede de **Saludo::Service**.
- Implementa la lógica del método.
- Usa `grpc::ServerBuilder` para iniciar el servidor.



# Escribir el cliente

- Usa **Saludo::Stub** para conectarte al servidor.
- Llama a **DiHola()** pasando una instancia de **Solicitud**.
- Recibe la respuesta en una instancia de **Respuesta**.

# En Visual Studio

- Crear una solución:
  - En la carpeta Proto mantener el fichero .proto y los ficheros generados, se llamarán:
    - **saludo.pb.h**
    - **saludo.pb.cc**
    - **saludo.grpc.pb.h**
    - **saludo.grpc.pb.cc**
  - Crear dos proyectos en la solución:
    - **Servidor y Cliente**
  - En cada proyecto hay que vincular los ficheros generados con **protoc**
  - Se pueden agregar como elementos existentes con botón derecho sobre cada proyecto.
  - Después en propiedades de cada proyecto, ir a C/C++ → General → Directorios de inclusión adicionales y añadir **\$(SolutionDir)Proto**

# saludo.pb.h

- **Encabezado de Protocol Buffers**
- Define las clases C++ para los mensajes que has declarado en el .proto, como:
  - **Solicitud**
  - **Respuesta**
- Incluye métodos para:
  - Acceder y modificar campos (set\_nombre(), nombre(), etc.)
  - Serializar y deserializar los mensajes
  - Comparar, copiar, limpiar, etc.

# saludo.pb.cc

- **Implementación de Protocol Buffers**
- Contiene el código fuente que implementa las clases declaradas en **saludo.pb.h**
- Incluye la lógica de serialización, construcción de objetos, y manejo interno de campos

# saludo.grpc.h

- **Encabezado de gRPC**
- Define las interfaces para el servicio gRPC que declaraste en el .proto, por ejemplo:
  - Clase abstracta Saludo::**Service** con métodos virtuales como DiHola(...)
  - Clase Saludo::**Stub** para que el cliente invoque el servicio
- También incluye definiciones para el contexto de llamada (grpc::ServerContext, grpc::ClientContext)

# saludo.grpc.pb.cc

- **Implementación de gRPC**
- Implementa el **stub** del cliente y el registro del servicio en el servidor
- Contiene el código que conecta los métodos RPC con el transporte gRPC
- Maneja la serialización de mensajes entre cliente y servidor usando Protocol Buffers

- **Tipos de datos más ricos**

- int32, int64, float, double, bool, bytes
- enum para valores constantes y repeated para listas

```
message Usuario {  
    string nombre = 1;  
    int32 edad = 2;  
    repeated string intereses = 3;  
}
```

- **Streaming de datos (gRPC permite streaming bidireccional)**

```
service Chat {  
    rpc Conversacion(stream Mensaje) returns (stream Mensaje);  
}
```

- **Mensajes anidados y reutilizables:**

```
message Direccion {  
    string calle = 1;  
    string ciudad = 2;  
}
```

```
message Perfil {  
    string nombre = 1;  
    Direccion direccion = 2;  
}
```

# IDL

- Enums para estados o tipos:

```
enum Estado {  
    ACTIVO = 0;  
    INACTIVO = 1;  
    SUSPENDIDO = 2;  
}
```

- Opciones personalizadas y extensiones

- Opciones para definir metadatos, como validaciones o documentación extra.

```
message Producto {  
    string nombre = 1 [(validate.rules).string.min_len = 3];  
}
```



# IDL

- **Los números de los campos son identificadores únicos (en formato binario)**
- **Permite que sean compactos y eficientes:**
  - **Solo se transmiten los números**, no los nombres. Esto:
    - Reduce el tamaño del mensaje.
    - Permite renombrar campos sin romper compatibilidad.
    - Hace que el protocolo sea más rápido y eficiente.
- ***Es crítico para la compatibilidad, cambiar el número rompe la compatibilidad con versiones anteriores.***

# IDL

- Los números deben estar entre 1 y  $2^{29} - 1$  (excepto los reservados).
- No se deben reutilizar ni cambiar una vez en uso.
- Se pueden reservar números: reserved 3,4,5;
- Los números se reservan dentro del mensaje.
  - Se suelen reservar cuando se elimina un campo y su número correspondiente, en este caso es mejor reservarlo para no volver a utilizarlo.

# Aplicaciones comunes entre gRPC / C++

- **Sistemas distribuidos y microservicios**

- Comunicación entre servicios en arquitecturas modernas.
- Sustituto de REST cuando se necesita eficiencia.
- Ejemplo: backend de una plataforma de streaming o e-commerce.

- **Redes y telecomunicaciones**

- Transmisión de datos en tiempo real.
- Control de dispositivos de red, routers, switches.
- Ejemplo: gestión de infraestructura 5G o SDN (Software Defined Networking).

# Aplicaciones comunes entre gRPC / C++

- **IoT y dispositivos embebidos**

- Comunicación entre sensores, gateways y servidores.
- gRPC es ligero y eficiente, ideal para dispositivos con recursos limitados.

- **Gaming y simulaciones**

- Sincronización de estado entre cliente y servidor.
- Envío de eventos en tiempo real (movimiento, colisiones, etc.).
- Ejemplo: juegos multijugador con servidores dedicados.

- **Machine Learning y procesamiento de datos**

- Comunicación entre modelos, servidores de inferencia y clientes.
- Ejemplo: un cliente C++ que envía imágenes a un servidor Python con TensorFlow.

# Aplicaciones comunes entre gRPC / C++

- **Finanzas y trading algorítmico**

- Envío de datos de mercado en tiempo real.
- Ejecución de órdenes con baja latencia.
- Ejemplo: plataformas de trading de alta frecuencia.

- **Automoción y robótica**

- Comunicación entre módulos de control (sensores, actuadores, navegación).
- Ejemplo: vehículos autónomos que intercambian datos entre subsistemas.

# gRPC vs REST

Característica	gRPC	REST
Protocolo	HTTP/2	HTTP/1.1
Serialización	Binaria (Protobuf)	Texto (JSON)
Rendimiento	Alto	Medio
Streaming	Bidireccional	Limitado
Tipado	Fuerte (IDL)	Débil
Soporte multilenguaje	Excelente	Bueno

# Servicio con streaming

```
syntax = "proto3";  
package chat;
```

```
service ChatService {  
  rpc Conversacion(stream Mensaje) returns (stream Mensaje);  
}
```

```
message Mensaje {  
  string usuario = 1;  
  string texto = 2;  
}
```

# Servidor simplificado

```
class ChatServiceImpl final : public ChatService::Service {
    Status Conversacion(ServerContext* context,
        ServerReaderWriter<Mensaje, Mensaje>* stream) override {
        Mensaje mensaje;
        while (stream->Read(&mensaje)) {
            std::cout << "Recibido de " << mensaje.usuario() << ": " << mensaje.texto() << std::endl;

            Mensaje respuesta;
            respuesta.set_usuario("Servidor");
            respuesta.set_texto("Hola " + mensaje.usuario() + ", recibí tu mensaje.");
            stream->Write(respuesta);
        }
        return Status::OK;
    }
};
```



# Cliente simplificado

```
std::unique_ptr<ChatService::Stub> stub = ChatService::NewStub(channel);  
ClientContext context;  
std::shared_ptr<ClientReaderWriter<Mensaje, Mensaje>> stream(stub->Conversacion(&context));
```

```
Mensaje mensaje;  
mensaje.set_usuario("Cliente1");  
mensaje.set_texto("¡Hola servidor!");  
stream->Write(mensaje);
```

```
Mensaje respuesta;  
while (stream->Read(&respuesta)) {  
    std::cout << "Respuesta: " << respuesta.texto() << std::endl;  
}
```

# Enlaces

- Configurar **gRPC** en **Windows**:
  - <https://sanoj.in/2020/05/07/working-with-grpc-in-windows.html>

# RabbitMQ

# RabbitMQ

- **RabbitMQ** es un sistema de **mensajería intermedia (message broker)** que permite a diferentes aplicaciones comunicarse entre sí de forma **asíncrona, confiable y escalable**.
- Funciona como un **intermediario** que recibe mensajes de un productor (emisor) y los entrega a uno o varios consumidores (receptores), siguiendo distintos patrones de distribución.

# RabbitMQ

- **Desacoplar servicios:** los emisores no necesitan saber quién consume los mensajes.
- **Distribuir carga:** balancea el trabajo entre múltiples consumidores.
- **Persistencia:** puede almacenar mensajes hasta que sean entregados.
- **Escalabilidad:** permite añadir más productores o consumidores sin cambiar la lógica del sistema.

# RabbitMQ

- RabbitMQ se basa en el protocolo **AMQP (Advanced Message Queuing Protocol)** y utiliza tres componentes clave:
  - **Producer:** envía mensajes.
  - **Exchange:** decide cómo enrutar los mensajes.
  - **Queue:** almacena los mensajes hasta que un consumidor los procesa.

# Patrones de Uso

Patrón	Descripción
Work Queue	Distribuye tareas entre múltiples trabajadores
Publish/Subscribe	Un mensaje se envía a múltiples receptores
Routing	Mensajes se envían según claves específicas
Topic	Enrutamiento basado en patrones de temas
RPC	Simula llamadas remotas entre servicios

# Casos de Uso

- Procesamiento de tareas en segundo plano (ej. generación de PDFs, envío de correos).
- Comunicación entre microservicios.
- Sistemas de monitoreo y logging.
- Integración entre sistemas heterogéneos (Java, Python, C++, etc.).
- Control de flujo en sistemas embebidos o IoT.



# Lenguajes que soportan RabbitMQ

- RabbitMQ tiene clientes oficiales y comunitarios para:
- C++
- Python
- Java
- Go
- Node.js
- Rust, entre otros

# RabbitMQ

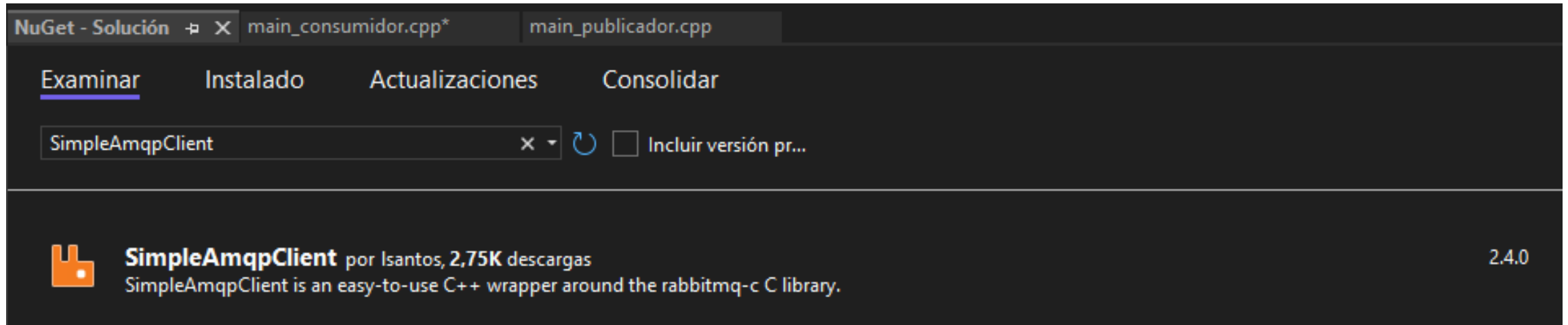
- Hay que tener **Erlang** y luego descargar e instalar RabbitMQ
  - <https://www.rabbitmq.com/docs/download>
- Mejor opción con **Docker**:
  - # latest RabbitMQ 4.x
  - `docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:4-management`***
- Para conectar microservicios mediante el protocolo AMQP: 5672
- Panel de administración Web: <http://localhost:15672>
  - User / Pass por defecto es: **guest / guest**

# Instalación de las librerías

- A diferencia de ZeroMQ y gRPC. RabbitMQ no es una librería, es un servidor de mensajería.
- Con vcpkg se pueden instalar librerías cliente compatibles con RabbitMQ
- **vcpkg install librabbitmq (librería oficial) ← en C**
- **vcpkg integrate install**
  - Depende de Boost y rabbitmq-c, pero el gestor vcpkg ya instala las dependencias.

# SimpleAmqpClient

- Utilizar NuGet para instalar los paquetes necesarios en la solución.
- Botón derecho sobre la solución → Administrar paquetes NuGet para la solución.
- En examinar y a la derecha botón instalar.



- Se puede instalar en Docker.
- **#include <SimpleAmqpClient/SimpleAmqpClient.h>**
- **La opción mas moderna y eficiente: AMQP-CPP → OJO C++ 17**
  - **vcpkg install amqpcpp**
  - **vcpkg integrate install**

# Ejemplos

- Utilizarlo en:
  - Procesamiento de tareas en segundo plano.
  - Arquitectura de microservicios.
  - Sistemas que requieren alta fiabilidad.
  - Integración de sistemas heterogéneos.
  - Control de flujo y balanceo de carga.
- No utilizarlo en:
  - Comunicación en tiempo real → WebSockets o gRPC.
  - Sistemas simples que no requieren desacoplamiento.
  - Modelos de comunicación más ligeros.

# Librerías AMQP

- **AMQP (Advanced Message Queuing Protocol)** es el protocolo estándar que usa RabbitMQ para enviar y recibir mensajes.
- RabbitMQ es un **broker** que implementa AMQP, pero **no proporciona directamente una librería oficial para C++**.

# **Comparativa entre REST y gRPC**



# Comparativa

- Son dos enfoques típicos para la comunicación entre servicios, especialmente en arquitecturas distribuidas y microservicios:

Característica	REST (HTTP/JSON)	gRPC (HTTP/2 + Protobuf)
 Protocolo	HTTP/1.1	HTTP/2
 Formato de datos	JSON	Protocol Buffers (binario)
 Definición de API	Manual (OpenAPI/Swagger opcional)	Automática con archivos <code>.proto</code>
 Comunicación	Sincrónica (por defecto)	Sincrónica y asíncrona (streaming bidireccional)
 Validación	Manual	Validación automática por esquema <code>.proto</code>
 Compatibilidad	Universal (navegadores, herramientas web)	Limitada en navegadores, ideal para backend
 Rendimiento	Medio	Alto (más rápido y eficiente)
 Multilenguaje	Muy buena	Excelente (con generación automática)
 Seguridad	TLS, OAuth, JWT	TLS, autenticación personalizada

# REST

- **Ideal para:**

- APIs públicas o abiertas.
- Aplicaciones web y móviles.
- Sistemas donde la compatibilidad con navegadores es clave.

- **Evítalo si:**

- Necesitas rendimiento extremo o streaming bidireccional.
- Quieres evitar la sobrecarga de JSON en servicios internos

# gRPC

- **Ideal para:**

- Comunicación entre microservicios backend.
- Sistemas embebidos, telecomunicaciones, alto rendimiento.
- Streaming de datos en tiempo real (IoT, juegos, ML).

- **Evítalo si:**

- Tu cliente es un navegador (gRPC no funciona directamente en ellos).
- No quieres depender de herramientas como protoc para generar código.

# Uso por sectores

Sector	REST	gRPC
Web pública	Ok	Limitado en navegadores
Microservicios	Ok	Más eficiente
Sistemas Embebidos	Pesado	Ligero y rápido
Telecomunicaciones	No	Streaming, binario, eficiente
IA / ML	Para Dashboards	Para inferencia distribuida

# Implementación

- **gRPC no usa directamente los verbos HTTP como POST, GET, PUT o DELETE.**
- **Se puede modelar esas operaciones típicas de un microservicio** en C++ usando gRPC, pero con un enfoque diferente.
  - Hay que diseñar un fichero .proto que sea equivalente a REST

# Ejemplo .proto

```
service ProductoService {  
    rpc CrearProducto (Producto) returns (Respuesta);    // POST  
    rpc ObtenerProducto (ProductoID) returns (Producto); // GET  
    rpc ActualizarProducto (Producto) returns (Respuesta); // PUT  
    rpc EliminarProducto (ProductoID) returns (Respuesta); // DELETE  
}
```

# Peticiones

REST (HTTP)	gRPC (RPC)
GET /producto/123	ObtenerProducto(ProductID)
POST /producto	CrearProducto(Producto)
PUT /producto/123	ActualizarProducto(Producto)
DELETE /producto/123	EliminarProducto(ProductID)



# Compilar fichero .proto

- Al compilar se generan clases C++ con métodos virtuales para su implementación:

```
class ProductoServiceImpl final : public ProductoService::Service {  
    grpc::Status CrearProducto(grpc::ServerContext* context,  
                               const Producto* request,  
                               Respuesta* response) override {  
        // lógica de creación  
        return grpc::Status::OK;  
    }  
};
```

# Ventajas

- Tipado fuerte y validación automática.
- Comunicación binaria, más rápida que JSON.
- Streaming bidireccional si lo necesitas.
- Generación automática de cliente y servidor.

# WebSockets

# WebSockets

- Dentro de la librería boost.beast con **WebSockets** podemos implementar:
  - Un **cliente WebSocket**: conectar con un servidor WebSocket remoto (para consumir datos en tiempo real).
  - Un **Servidor WebSocket**: crear un servidor que escuche conexiones WebSocket entrantes y gestionar múltiples clientes simultáneamente.

# Realizar el handshake

- Gestiona el **handshake HTTP** inicial que convierte una conexión HTTP en una conexión WebSocket.
- Permite personalizar los encabezados del handshake para añadir autenticación, tokens, etc.

# Enviar y recibir mensajes

- Lectura y escritura síncrona (read, write)
- Lectura y escritura asíncrona (async\_read, async\_write)
- Compatible con mensajes de texto y binarios
- Se pueden utilizar buffers dinámicos o estáticos, y gestionar los mensajes con precisión.

# Soporte para WebSocket seguro (WSS)

- Integración con SSL / TLS mediante Boost.Asio
- Establecer conexiones seguras usando certificados y clases privadas
- Ideal para aplicaciones que requieren confidencialidad (como chats, trading, IoT)

# Control de flujo y gestión de errores

- Manejo de errores detallado con `boost::system::error_code`
- Control de cierre de conexión (close) con códigos estándar WebSocket
- Detección de desconexiones, timeouts, y errores de protocolo



# Personalización avanzada

- Puedes acceder directamente a los **encabezados HTTP** del handshake
- Configurar opciones como:
  - **Fragmentación de mensajes**
  - **Tamaño máximo de buffer**
  - **Control de ping/pong** para mantener viva la conexión

# Integración con otras tecnologías

- Compatible con **Boost.Asio coroutines** (co\_spawn, awaitable)
- Puedes combinar WebSockets con **HTTP/REST**, **TCP**, o **SSL** en una misma aplicación
- Ideal para servidores híbridos que ofrecen tanto APIs como canales WebSocket

# Ejemplos de uso

- Chat en tiempo real
- Streaming de datos financieros
- Juegos multijugador
- Comunicación entre dispositivos IoT
- Actualización en vivo de interfaces web.

# Ejemplo: Cliente WebSocket con Boost.Beast

- Pasos:
  - Resolver el host y el puerto
  - Establecer la conexión TCP
  - Realiza el handshake WebSocket
  - Enviar un mensaje
  - Recibe la respuesta
  - Cierra la conexión
- Disponemos de un servidor público por el puerto 80:
- **echo.websocket.events**
- Para probar los WebSockets

- `asio::io_context ioc; // Definir el contexto de in-out`
- `tcp::resolver resolver(ioc); // Resolver DNS:`
- **`auto const results = resolver.resolve("echo.websocket.events", "80");`**
- `websocket::stream<tcp::socket> ws(ioc); // Crear el WebSocket:`
- **`// Conectar al Servidor: intenta conectar con el primer endpoint disponible`**
- `asio::connect(ws.next_layer(), results.begin(), results.end()); ws.handshake("echo.websocket.events", "/"); // Handshake WebSocket:`
- `std::string msg = "Mensaje de Boost.Beast"; // Enviar el mensaje:`
- `ws.write(asio::buffer(msg));`
- `beast::flat_buffer buffer; // Leer la respuesta:`
- `ws.read(buffer);`
- `std::cout << "Respuesta del Servidor: " << beast::make_printable(buffer.data()) << std::endl;`
- `beast::flat_buffer buffer2; // Leer una segunda respuesta:`
- `ws.read(buffer2);`
- `std::cout << "Respuesta 2 del Servidor: " << beast::make_printable(buffer2.data()) << std::endl;`
- `ws.close(websocket::close_code::normal); // Cerrar la conexión:`

# results

- **results** es un objeto de tipo **tcp::resolver::results\_type**, que es básicamente una colección de **tcp::endpoint + metadatos**.
- De cada elemento representa una posible dirección IP y puerto a la que puedes conectarte.
- De cada elemento puedes extraer:
  - Dirección IP (`endpoint.address()`): Por ejemplo, 93.184.216.34
  - Puerto (`endpoint.port()`): En este caso, 80 (puerto HTTP)
  - Familia de protocolo (`endpoint.protocol()`): Por ejemplo, `tcp::v4()` o `tcp::v6()`
  - Nombre del host y servicio (si accedes a los metadatos): Puedes obtener el nombre original que se resolvió (`host_name()`, `service_name()`)

# Inspeccionar

```
for (auto const& entry : results) {  
    auto endpoint = entry.endpoint();  
    std::cout << "IP: " << endpoint.address().to_string()  
        << ", Puerto: " << endpoint.port()  
        << ", Protocolo: " << (endpoint.protocol() == tcp::v4() ? "IPv4" : "IPv6")  
        << std::endl;  
}
```

# Ejemplo: Servidor WebSocket con Boost.Beast

- Este servidor:
  - Escucha en un puerto TCP.
  - Acepta conexiones WebSocket.
  - Lee mensajes del cliente.
  - Los devuelve tal cual (eco).
  - Cierra la conexión cuando el cliente lo solicita.



# Testear Servidor WebSocket

- Disponemos de una herramienta online para testear el servidor:
- <https://piehost.com/websocket-tester>
- Y luego nos conectamos a:
- `ws://localhost:8080`

# **Web Socket Secure**

## **WSS**

# WSS (WebSocket Secure)

- Necesitamos hacer algunos cambios:
  - Crear un contexto `ssl:context`
  - Cambiar el tipo WebSocket stream
  - Realizar el handshake SSL antes del handshake WebSocket
  - El puerto tiene que ser 443
- 
- Necesitamos la herramienta **openssl**, y generar un certificado

# Tipos de certificado

Tipo de certificado	¿Quién lo emite?	¿Dónde se usa?	¿Requiere hardware?
<b>DNI electrónico (DNle)</b>	Gobierno (Policía Nacional en España)	Trámites oficiales, firma electrónica	Sí, lector de tarjetas
<b>Certificado FNMT</b>	Fábrica Nacional de Moneda y Timbre	Administración pública, firma digital	No
<b>Certificado local (OpenSSL)</b>	Tú mismo con OpenSSL	Desarrollo, pruebas, servidores propios	No
<b>Certificados SSL/TLS</b>	Autoridades de certificación (CA)	Sitios web seguros (HTTPS)	No
<b>Certificados de firma de código</b>	CA como DigiCert, Sectigo	Firmar software, garantizar integridad	No

## CA: Autoridad de certificación

# openssl

- Comando: ***openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365***
- El comando:
  - Crea un certificado autofirmado (no emitido por una CA).
  - Clave privada RSA de 2048 bits
  - Pide una serie de datos
- Así como una contraseña para encriptar la clave:
  - Se genera como resultado dos archivos:
    - **key.pem → clave privada**
    - **cert.pem → tu certificado público autofirmado**
  - **Estos dos ficheros son necesarios para WSS**

# openssl

- Permite activar HTTPS / WSS
- Cifrar comunicaciones entre dispositivos
- Probar servicios sin tener que tener certificados oficiales
- Al comando se le puede añadir un parámetro para evitar que nos pida la información
- -subj “...”

# openssl

Campo	Descripción
C	País (Country)
ST	Estado o provincia (State)
L	Localidad o ciudad (Locality)
O	Organización (Organization)
OU	Unidad organizativa (Organizational Unit)
CN	Nombre común (Common Name)
emailAddress	Correo electrónico

- Cada inicial lleva una barra / delante del campo.
- En el comando si añadimos **-nodes** no encripta la clave privada **key.pem**

```
openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365 -nodes \  
-subj "/C=ES/ST=Madrid/L=Madrid/O=AntonioTech/OU=IoT/CN=raspberry.local/emailAddress=antonio@ex.com"
```

# Servidor WSS

- **El primer paso es configurar el contexto para SSL.**
- Tenemos que indicar los dos ficheros generados anteriormente.

```
boost::asio::ssl::context ctx(boost::asio::ssl::context::tlsv12);
```

```
ctx.set_options(  
    boost::asio::ssl::context::default_workarounds |  
    boost::asio::ssl::context::no_sslv2 |  
    boost::asio::ssl::context::no_sslv3 |  
    boost::asio::ssl::context::single_dh_use  
);
```

```
ctx.use_certificate_file("cert.pem", boost::asio::ssl::context::pem);  
ctx.use_private_key_file("key.pem", boost::asio::ssl::context::pem);
```



# Diferencias entre: `boost::asio::io_context` / `boost::asio::ssl::context`

- **`boost::asio::io_context`**

- El motor principal de I/O (asíncronas)
- Coordinar eventos en conexiones TCP, temporizadores y lectura / escritura de sockets.
- Ejecutar los handlers cuando ocurren eventos

- **`boost::asio::ssl::context`**

- Contexto de configuración para TLS/SSL, para cifrar comunicaciones con HTTPS / WSS
- Certificados a utilizar
- Que claves privadas cargar
- Protocolos TLS como TLS 1.2 o 1.3)
- Opciones de seguridad

```
void run_server_1_mensaje(net::io_context& ioc, ssl::context& ctx, unsigned short port) {  
    tcp::acceptor acceptor(ioc, tcp::endpoint(tcp::v4(), port));
```

## Servidor WSS

```
for (;;) {  
    tcp::socket socket(ioc);  
    acceptor.accept(socket);  
  
    ssl::stream<tcp::socket> ssl_stream(std::move(socket), ctx);  
    ssl_stream.handshake(ssl::stream_base::server);  
  
    websocket::stream<ssl::stream<tcp::socket>> ws(std::move(ssl_stream));  
    ws.accept();  
  
    beast::flat_buffer buffer;  
    ws.read(buffer);  
    ws.text(ws.got_text());  
    std::cout << "Mensaje recibido: " << beast::make_printable(buffer.data()) << std::endl;  
  
    ws.write(buffer.data());  
}
```

# Servidor WSS - Pasos

- **Pasos**

- **1 – Inicializar io\_context, configurar SSL/TLS e indicar el puerto**

- **2 – Inicializar el servidor**

- `tcp::acceptor acceptor(ioc, tcp::endpoint(tcp::v4(), port));`

- Crea un acceptor TCP para escuchar por el puerto indicado

- Y utilizamos IPv4

- **3 – Bucle principal para aceptar clientes:**

- `for (;;) {`

- `tcp::socket socket(ioc); // Acepta conexiones`

- `acceptor.accept(socket); // y crea un socket TCP`

# Servidor WSS – Pasos II

- **4 – HandShake TLS**

- **// Crea un stream SSL sobre TCP**
- `ssl::stream<tcp::socket> ssl_stream(std::move(socket), ctx);`
- **// Realiza el handshake TLS como servidor**
- `ssl_stream.handshake(ssl::stream_base::server);`

- **5 – HandShake WebSocket**

- **// Crea un stream WebSocket sobre el canal TLS**
- `websocket::stream<ssl::stream<tcp::socket>> ws(std::move(ssl_stream));`
- **// Realiza el handshake WebSocket, para completar la conexión WSS**
- `ws.accept();`

# Servidor WSS – Pasos III

- **6 – lectura del mensaje**

- **// El buffer se utiliza para los datos entrantes**
- `beast::flat_buffer buffer;`
- **// Lee un mensaje del buffer.**
- `ws.read(buffer);`

- **7 – Procesamiento y eco**

- `// Configura el mensaje para tratarlo como texto no binario`
- `ws.text(ws.got_text());`
- **// Imprime el mensaje recibido en un formato legible**
- `std::cout << "Mensaje recibido: " << beast::make_printable(buffer.data()) << std::endl;`
- **// De Vuelta al cliente, hace el eco**
- `ws.write(buffer.data());`

# Cliente WSS - Pasos

- Utilizaríamos: **Boost.Beast + Boost.Asio + OpenSSL**
- **Pasos:**
  - **1- Configurar el contexto SSL**
  - `boost::asio::ssl::context ctx(boost::asio::ssl::context::tlsv12);`
  - **// Para certificados autofirmados**
  - `ctx.set_verify_mode(boost::asio::ssl::verify_none);`
  - **2- Resolver y conectar**
  - `boost::asio::io_context ioc;`
  - `tcp::resolver resolver(ioc);`
  - `auto const results = resolver.resolve("localhost", "9002");`
  - `boost::asio::ssl::stream<tcp::socket> stream(ioc, ctx);`
  - `boost::asio::connect(stream.next_layer(), results);`
  - `stream.handshake(boost::asio::ssl::stream_base::client);`

# Cliente WSS – Pasos II

- **Pasos:**

- **3 – Handshake WebSocket**

- `beast::websocket::stream<boost::asio::ssl::stream<tcp::socket>>`  
`ws(std::move(stream));`
    - `ws.handshake("localhost", "/");`

- **4 – Enviar y recibir**

- `ws.write(boost::asio::buffer("Hola servidor"));`
    - `beast::flat_buffer buffer;`
    - `ws.read(buffer);`
    - `std::cout << "Respuesta: " << beast::make_printable(buffer.data()) << std::endl;`

# **Concurrencia & multithreading**



# std::thread / Boost.Asio

- **std::thread** es la clase estándar de C++ para crear y manejar hilos. Te permite ejecutar funciones en paralelo.
  - Necesitas manejar sincronización con std::mutex, std::condition\_variable, etc.
  - No escala bien para miles de conexiones simultáneas (como en servidores web).
- **Boost.Asio** es una librería para **programación asíncrona y basada en eventos**, ideal para manejar múltiples conexiones de red sin bloquear hilos.
  - Manejo eficiente de miles de conexiones con pocos hilos.
  - Compatible con std::thread, std::future.
  - Ideal para microservicios, servidores HTTP, y sistemas embebidos.

# Comparativa

Herramienta	Ideal para...	Evítalo si...
<code>std::thread</code>	Tareas paralelas simples, procesamiento	Necesitas escalabilidad o IO intensiva
<code>Boost.Asio</code>	Servidores concurrentes, IO no bloqueante	Tu aplicación es muy simple o CPU-bound

## CPU-bound:

El programa **consume mucho tiempo de CPU** realizando cálculos intensivos.

El cuello de botella está en la **velocidad de procesamiento**, no en la espera por datos externos.

Aumentar el número de núcleos o la frecuencia del procesador puede mejorar el rendimiento.

# **std::thread**

# Contenidos

- Clase thread
- Paso de parámetros a los hilos.
- Regiones críticas, interbloqueos, condiciones de carrera.
- Mecanismos de sincronización en hilos:
  - Mutex
- Variables de condición.
- Esquema productor / consumidor.
- Futures y tareas asíncronas.

# threads

- Soporte en C++11
- Para trabajar con hilos, incluir el fichero .H
  - **#include <thread>**
- Para compilar con g++:
  - **g++ -std=c++11 fichero.cpp -o fichero -lpthread**
- Para compilar con make:
  - `set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -lpthread")`
  - `set(CMAKE_CXX_STANDARD 11)`
  - `set(CMAKE_CXX_STANDARD_REQUIRED ON)`

# Lanzamiento de Hilos

- En C++11 un hilo se puede lanzar de 3 formas distintas:
  - Con una función.
    - La función puede tener parámetros o no.
  - Con un objeto de una clase que implemente el operador ()
    - También puede ser una estructura con la implementación de dicho operador.
  - Con una función lambda.

# Con una función

- Primero se define una función:

```
void funcion_hello(){  
    int i;  
    for (i = 0 ; i < 10 ; i++)  
        std::cout << "Hello " << i << std::endl;  
}
```

```
std::thread h1(funcion_hello);  
h1.join();
```

- También se puede inicializar el hilo con las {}  
 std::thread h2 {función\_hello}

# Con una clase + operador ()

```
class MiFuncion {  
    public:  
        void operator()(){  
            for (int i = 0 ; i < 10 ; i++)  
                std::cout << "Operador () " << i << std::endl;  
        }  
};
```

```
std::thread h2( (MiFuncion()) );  
h2.join();
```

// Ojo, se instancia la clase MiFuncion() se necesitan los paréntesis extras.



# Con una función lambda

```
std::thread h3([]{  
    for (int i = 0 ; i < 10 ; i++)  
        std::cout << "Lambda " << i << std::endl;  
});  
  
h3.join();
```

# Condiciones de carrera

```
int x = 42;  
void f () { ++x; }  
void g() { x=0; }  
void h() { cout << "Hola" << endl; }  
void i () { cout << "Adios" << endl; }
```

// La variable x la comparten dos hilos sin ningún tipo de protección.

```
void carrera() {  
    thread t1{ f };  
    thread t2{ g };  
    t1.join () ;  
    t2.join () ;  
    thread t3{ h };  
    thread t4{ i };  
    t3.join ();  
    t4.join () ;  
}
```

# Paso de argumentos a un hilo

- A un hilo se le pueden pasar un número indeterminado de argumentos.
- La función que ejecute el hilo tiene que tener todos esos argumentos.
- Al instanciar el hilo se le manda como primer parámetro la función que tiene que ejecutar.

```
void funcion(int x, std::string s){  
    std::cout << "Parametro int: " << x << std::endl;  
    std::cout << "Parametro string: " << s << std::endl;  
}
```

```
// suele hacer un casting automático de const char * a std::string  
std::thread hilo(funcion, 1, std::string("hola"));  
hilo.join()
```

# Paso de argumentos a un hilo

- La definición de la clase thread:
- El constructor recibe una función y un número indeterminado de argumentos, que pueden ser 0 o n
- **thread thread( Function&& *f*, Args&&... *args* );**
- Un hilo termina cuando finaliza la rutina que ejecuta (por ejemplo, realiza un proceso n veces y termina) y llama a la instrucción **return**.

# Paso de parámetros por referencia

- Cuando queremos pasar un parámetro a un hilo por referencia se tiene que indicar en la construcción del hilo.
- Para ello se dispone de la función `std::ref(param)`
- `#include <functional>`
- `#include <thread>`
- **void** f ( registro & r) ;
- **void** g( registro & s) {
  - `thread t1{ f ,s};` // Copia de s
  - `thread t2{ f , std::ref (s) };` // Referencia a s
  - `Thread t3 {[&] { f (s) ; }};` // Referencia a s, con la lambda también se puede indicar.

# Esperar a que termine un hilo: **join()**

- Siempre se lanza un hilo principal (desde main) y a partir de este se van creando el resto de hilos.
- Para esperar a que un hilo termine se dispone del método **join()**.
- Sólo se puede llamar una vez al método **join**.
- Se dispone de la función en thread: **joinable()** se aplica sobre un objeto thread y devuelve true / false para indicar si se puede hacer join a un hilo o no.

# Vectores de hilos

- Los hilos se pueden combinar con la clase **vector** para tener varios hilos.

```
#include <vector>
```

```
#include <thread>
```

```
class Hilo {  
    public:  
    void operator()(){  
        // Muestra el identificador del hilo  
        std::cout << "Dentro del hilo: " << std::this_thread::get_id() << " esta ejecutando" << std::endl;  
    }  
};
```

```
std::vector<std::thread> hilos;
```

```
// Creamos 10 hilos y se añaden al vector:
```

```
for (int i = 0 ; i < 10 ; i++)  
    hilos.push_back(std::thread( (Hilo()) ));
```

```
// Ahora esperamos a que acaben todos los hilos:
```

```
std::cout << std::endl << "Esperamos por todos los hilos" << std::endl;
```

```
for (auto &h : hilos)  
    h.join();
```

# mutex

- Al igual que en POSIX los **mutex** (cerrojo) nos sirven para sincronizar el acceso de varios hilos a un recurso compartido para evitar condiciones de carrera y que se corrompa la memoria.

- La 1ª forma: más propensa a errores se puede olvidar el desbloqueo del mutex:

```
#include <mutex>
```

```
miMutex.lock();    // Adquiere el cerrojo
```

```
// Actualizar el recurso;
```

```
miMutex.unlock();    // Libera el cerrojo
```

- La 2ª forma: es más segura, se evita el posible error de la primera forma. El mutex se libera automáticamente.

```
std::lock_guard<std::mutex> guard(miMutex);
```

```
// Actualizar el recurso y después se libera automáticamente.
```

- La 3ª forma: es equivalente a lock\_guard → unique\_lock
- **unique\_lock**<mutex> milock {miMutex};
- // Actualizar el recurso y después se libera automáticamente.



# lock\_guard vs unique\_lock

- **lock\_guard** y **unique\_lock** son más o menos lo mismo; lock\_guard es una versión restringida con una interfaz limitada.
- **lock\_guard** siempre tiene un candado desde su construcción hasta su destrucción.
- **unique\_lock** puede crearse sin bloqueo inmediato, puede desbloquearse en cualquier momento de su existencia y puede transferir la propiedad del bloqueo de una instancia a otra.
- Por lo tanto, siempre utilizaremos lock\_guard, a menos que se necesiten las capacidades de unique\_lock.
- Una variable condition\_variable necesita a unique\_lock.

# detach: Hilos no asociados

- Se puede indicar que un hilo sigue ejecutando después de que el destructor se ejecute con **detach()**.
- Útil para tareas que se ejecutan como demonios.

```
void actualiza () {  
    for (;;) {  
        muestra_reloj(stead_clock::now());  
        this_thread :: sleep_for(second{1});  
    }  
}  
  
void f () {  
    thread t { actualiza };  
    t.detach();  
}
```

# Problemas con hilos no asociados

- Inconvenientes:
  - Se pierde el control de qué hilos están activos.
  - No se sabe si se puede usar el resultado generado por un hilo.
  - No se sabe si un hilo ha liberado sus recursos.
  - Se podría acabar accediendo a objetos que han sido destruidos.

# Variables de condición

- Mecanismo para sincronizar hilos en acceso a recursos compartidos:
  - wait(): Espera en un mutex.
  - notify\_one(): Despierta a un hilo en espera.
  - notify\_all(): Despierta a todos los hilos en espera.
- Productor / Consumidor
  - **class** peticion ;
  - queue<peticion> cola; // Cola de peticiones
  - condition\_variable cv;
  - mutex m;
  - **void** productor();
  - **void** consumidor();

# Consumidor

```
void consumidor() {  
    for (;;) {  
        unique_lock<mutex> l{m};  
        while (cv.wait( l ) ) ;  
        auto p = cola. front ( ) ;  
        cola.pop();  
        l.unlock() ;  
        procesa(p);  
    };  
}
```

- Efecto de **wait**
  - Libera el cerrojo y espera una notificación.
  - Adquiere el cerrojo al despertarse.

# Producer

- **void** productor() {
    - **for** (;;) {
      - petición p = genera();
      - unique\_lock<mutex> l{m};
      - cola.push(p);
      - cv.notify\_one() ;
    - }
  - }
- Efecto de notify\_one()
    - Despierta a uno de los hilos que están esperando en la condición.

# Tareas asíncronas y future

- Una tarea **asíncrona** permite el lanzamiento simple de la ejecución de una tarea:
  - **En otro hilo** de ejecución.
  - Como una **tarea diferida**.
- Un **future** es un objeto que permite que un hilo pueda devolver un valor a la sección de código que lo invocó

# Invocación de tareas asíncronas

```
#include <future>
```

```
#include <iostream>
```

```
int main() {
```

```
    std :: future<int> r = std :: async(tarea, 1, 10);
```

```
    otra_tarea() ;
```

```
    std :: cout << "Resultado= " << r.get() << std :: endl;
```

```
    return 0;
```

```
}
```



# Uso de futuros

- **Idea general:**
  - Cuando un hilo necesita pasar un valor a otro hilo pone el valor en una **promesa**.
  - La implementación hace que el valor esté disponible en el correspondiente **futuro**.
- Acceso al **futuro** mediante **f.get()**:
  - Si se ha asignado un valor → obtiene el valor.
  - En otro caso → el hilo llamante se bloquea hasta que esté disponible.
  - Permite la transferencia transparente de excepciones entre hilos.

# **Boost.Asio**

# ¿Qué es Boost.Asio?

- **Boost.Asio** es una librería de C++ para **programación asíncrona y basada en eventos**, especialmente útil para:
  - Redes TCP/UDP
  - Timers
  - Serialización
  - Multithreading
  - I/O no bloqueante
- Está diseñada para construir aplicaciones **eficientes, escalables y concurrentes**, como servidores web, microservicios, sistemas embebidos o clientes de red.

# Características

- **Modelo asíncrono:** evita bloqueos usando callbacks.
- **Sin dependencias externas:** todo se basa en C++ estándar y Boost.
- **Multiplataforma:** funciona en Windows, Linux, macOS.
- **Integración con `std::thread` y `std::future`** para concurrencia moderna.
- **Timers y señales:** ideal para tareas periódicas o eventos del sistema.

# Ejemplo

```
boost::asio::io_context io;  
tcp::acceptor acceptor(io, tcp::endpoint(tcp::v4(), 1234));  
  
while (true) {  
    tcp::socket socket(io);  
    acceptor.accept(socket);  
    // Manejar la conexión  
}
```

# Uso de Boost.Asio

Sector	Aplicación típica
Microservicios	Servidores HTTP, gRPC, ZeroMQ integrados
Sistemas embebidos	Comunicación entre sensores y controladores
Telecomunicaciones	Procesamiento de paquetes y señalización
Juegos en red	Sincronización de estado y eventos

# Cuando elegir Boost.Asio

- Manejar miles de conexiones simultáneas sin bloquear hilos.
- Trabajar en sistemas de alto rendimiento o embebidos.
- Solución ligera y sin dependencias externas como gRPC o RabbitMQ

# Multicliente con boost.asio

- Tenemos que utilizar
    - **async\_read** y **async\_write**
    - Para no bloquear el flujo del servidor.
  - El servidor escucha por un puerto:
  - Acepta múltiple conexiones TCP
  - Leer solicitudes HTTP
  - Responder un mensaje
  - Cierra la conexión después de enviar la respuesta.
- **Flujo de ejecución:**
  - **main():**
    - Crea el io\_context, el endpoint y el acceptor.
    - Llama a do\_accept() para comenzar a aceptar conexiones.
    - Ejecuta el bucle de eventos con ioc.run().
  - **do\_accept():**
    - Espera una conexión entrante.
    - Cuando llega, crea una nueva session y llama a start().
  - **session::start():**
    - Inicia la lectura de la solicitud HTTP con http::async\_read.
  - **session::handle\_request():**
    - Procesa la solicitud y prepara una respuesta HTTP
  - **session::write\_response():**
    - Envía la respuesta al cliente y cierra el socket.



# Objetos del Servidor

Objeto / Clase	Descripción
<code>asio::io_context</code>	Motor de eventos de Boost.Asio. Coordina operaciones asíncronas.
<code>tcp::endpoint</code>	Representa la dirección IP y el puerto donde el servidor escucha (en este caso, <code>0.0.0.0:8080</code> ).
<code>tcp::acceptor</code>	Acepta conexiones entrantes de clientes TCP.
<code>tcp::socket</code>	Canal de comunicación con el cliente. Se usa para leer y escribir datos.
<code>session</code>	Clase que maneja la comunicación con un cliente individual. Se crea una instancia por conexión.
<code>beast::flat_buffer</code>	Buffer de Boost.Beast para almacenar datos leídos del socket.
<code>http::request&lt;http::string_body&gt;</code>	Representa la solicitud HTTP recibida del cliente.
<code>http::response&lt;http::string_body&gt;</code>	Representa la respuesta HTTP que se enviará al cliente.

# Multicliente

- La clave está en cómo se manejan las conexiones:
  - Usa `tcp::acceptor::async_accept()` para aceptar conexiones **de forma asíncrona**.
  - Cada vez que se acepta una conexión, se crea una nueva instancia de **session** con **`std::make_shared<session>(...)`**.
  - Dentro de **`do_accept()`**, se llama recursivamente a sí mismo para seguir aceptando más clientes:
    - `do_accept(acceptor);` // vuelve a aceptar otra conexión
  - Cada session maneja su propia conexión de forma independiente usando **`shared_from_this()`**.

# session::start()

- **No son un hilo en sí**, pero **funcionan como una unidad de trabajo asíncrona**, muy parecida a lo que haría un hilo en un servidor multiciente.
  - *Se crea una sesión por cada conexión entrante*
- Usa operaciones **asíncronas** (async\_read, async\_write) que se ejecutan dentro del asio::io\_context.
- No crea un hilo nuevo, pero se comporta como una tarea independiente que puede ejecutarse en paralelo con otras sesiones si el **io\_context** tiene varios hilos activos.

# session

- Cada cliente tiene su propia session, con su propio socket, buffer, solicitud y respuesta.
- Esto permite que múltiples clientes se atiendan en paralelo, sin interferencias entre ellos.
- Como las operaciones son asíncronas, no necesitas un hilo por cliente, lo que permite escalar a miles de conexiones

# Dentro de la session

- Representa una sesión HTTP individual.
  - Leer solicitud HTTP
  - Procesarla
  - Enviar respuesta
  - Cerrar la conexión

# Varios threads

- **Se pueden usar varios núcleos del procesador:**

```
std::vector<std::thread> threads;  
for (int i = 0; i < std::thread::hardware_concurrency(); ++i) {  
    // Construye el thread dentro del vector para evitar copias  
    threads.emplace_back([&ioc] { ioc.run(); });  
}  
for (auto& t : threads) t.join();
```