

# 반도체 제조 자동화를 위한 gRPC

---

gRPC with Protocol Buffer (SEMI E179)

SEMI E179 - Protocol Buffers Common Component

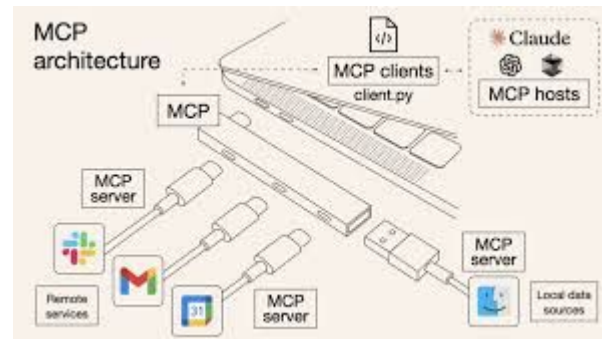
# 발표자 소개



## History

- 여울솔루션 소속 (스타트업)
- SECS/GEM 개발자 수다방 운영
- 반도체 / 디스플레이 제조장비 운영 프로그램 개발

# 오늘의 주제



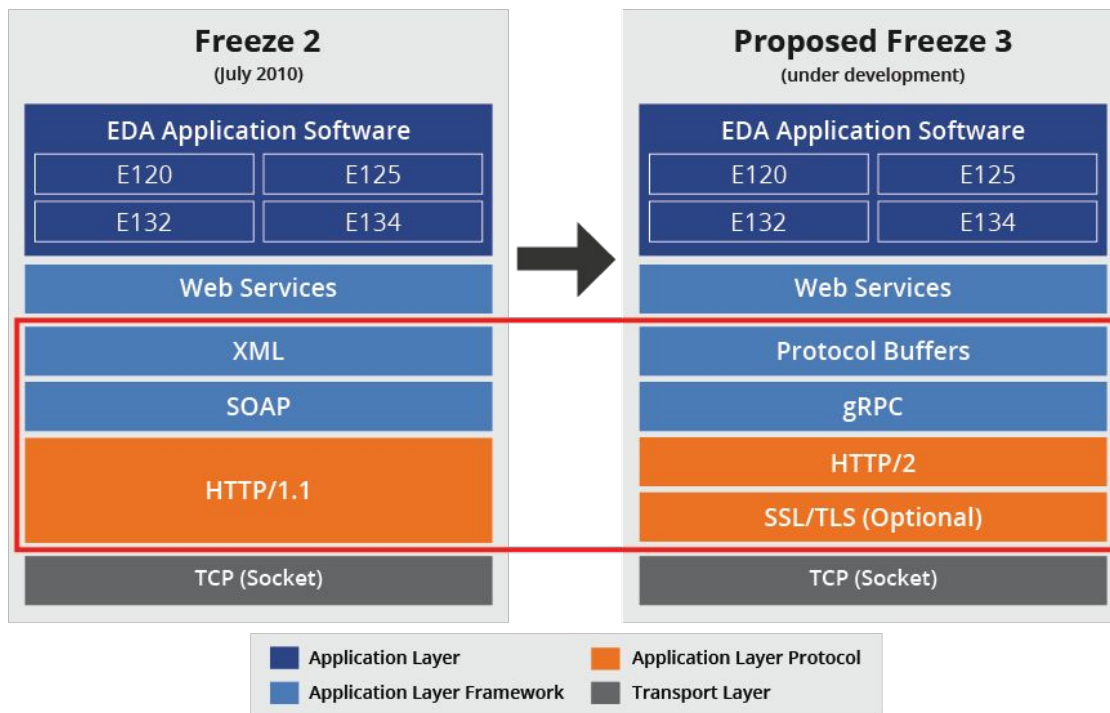
**SEMI E179 -  
Specification for  
Protocol Buffers  
Common Components**



# GEM300 vs EDA 1 / 2 vs EDA 3

GEM/GEM300	EDA 1, 2	EDA 3
TCP/IP binary protocol	HTTP/1, SOAP/XML, Web Services	HTTP/2, gRPC, Protocol Buffers
Command, control and monitor	Data collection, monitor only	Data collection, monitor only
Requires documentation	Self-describing metadata	Self-describing metadata
Single point of communication*	Multiple, independent clients	Multiple, independent clients, <b>passwords</b>
Mature, widespread adoption	Leading global 300mm wafer fabs	<b>Not completed or published yet</b>
Publish/subscribe event, alarm notification, with data	Publish/subscribe event, alarm notification with data	Publish/subscribe event, alarm notification with data
Data polling (trace reporting)	Data polling (traces) with optional triggers to start/stop data collection	Data polling (traces, <b>cached data</b> ) with optional triggers to start/stop data collection

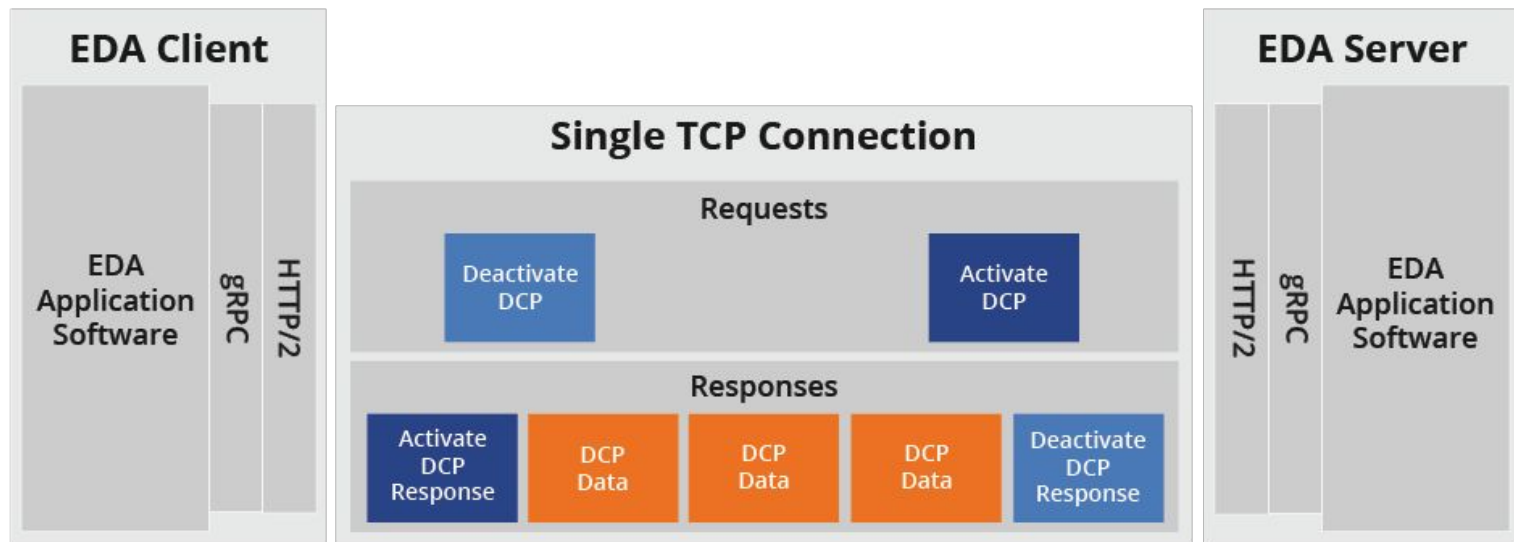
# gRPC Position



출처 : PeerGroup Homepage

<https://www.peergroup.com/article/can-we-get-better-performance-with-eda/>

# gRPC Position



출처 : PeerGroup Homepage

<https://www.peergroup.com/article/can-we-get-better-performance-with-eda/>

## Protocol Buffers란?

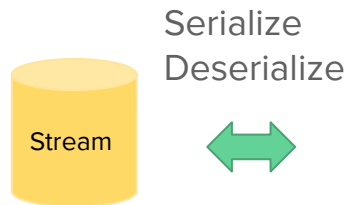
- Google에서 개발한 구조화된 데이터 직렬화 프레임워크
- 언어 및 플랫폼 중립적인 메커니즘  
(C++, Java, Kotlin, Python, Objective-C, C#, Ruby, Go, PHP, Dart, JavaScript)
- 작고 빠른 바이너리 패킷으로 데이터 교환

## 주요 장점:

- 작은 메시지 크기 (XML, JSON 대비 3-10배 작음)
- 빠른 인코딩/디코딩 속도
- 강력한 타입 체크
- 하위 호환성 유지 (이전 버전 메시지도 처리 가능)

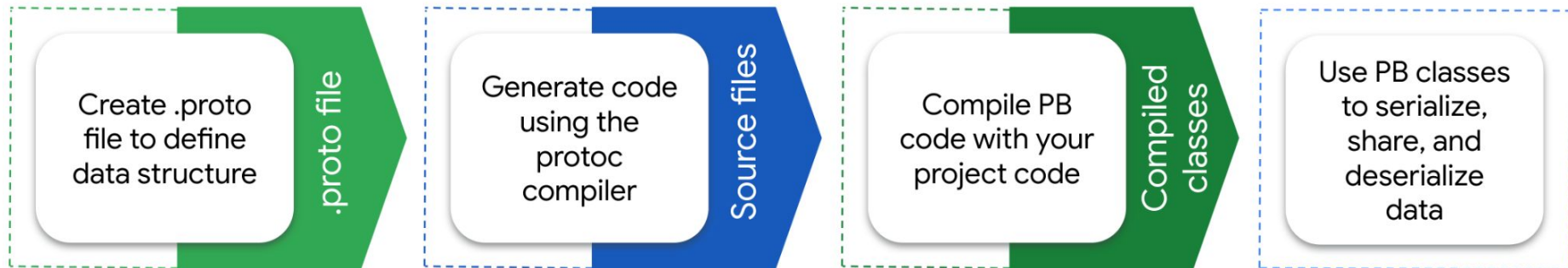
<https://github.com/protocolbuffers/protobuf/>

<https://protobuf.dev/>



```
// C++ code
Person john;
fstream input(argv[1],
              ios::in | ios::binary);
john.ParseFromIstream(&input);
id = john.id();
name = john.name();
email = john.email();
```

## 어떻게 작동하는가 ?





# proto3 파일 구조

```
// 프로토콜 버퍼 버전 선언 (필수)
syntax = "proto3";

// 패키지 이름 (선택사항이지만 권장됨)
package semiconductor.equipment;

// 다른 .proto 파일 임포트 (필요시)
import "google/protobuf/timestamp.proto";

// 옵션 설정 (언어별 설정 등)
option csharp_namespace = "Semiconductor.Equipment.Messages";

// 메시지 정의
message EquipmentStatus {
    // 필드 정의
    string equipment_id = 1;
    int32 status_code = 2;
    string status_description = 3;
    google.protobuf.Timestamp timestamp = 4;
}

// 추가 메시지 정의
message EquipmentList {
    repeated EquipmentStatus equipment = 1;
}
```

## 주요 구성 요소

- **syntax:** protobuf 버전을 명시 (proto3 사용)
- **package:** 메시지 타입 간의 이름 충돌을 방지하기 위한 네임스페이스
- **import:** 다른 .proto 파일의 정의를 사용할 때 필요
- **option:** 언어별 설정이나 기타 컴파일 옵션 지정
- **message:** 구조화된 데이터 형식 정의
- **필드 규칙:** **singular**(기본값, 명시하지 않음), **repeated**(배열/리스트)
- **필드 번호:** 각 필드는 고유한 번호를 가지며, 바이너리 인코딩에 사용됨

## proto3 주요 특징

- **필수 필드 제거:** proto2와 달리 모든 필드는 기본적으로 선택사항
- **기본값 사용:** 지정되지 않은 필드는 기본값(0, 빈 문자열 등)을 가짐
- **unknown 필드 처리 단순화:** 알 수 없는 필드는 무시됨
- **map 타입 지원:** 키-값 쌍의 효율적인 표현 가능
- **JSON 매핑 지원:** protobuf와 JSON 간의 변환 규칙 정의

# 기본 데이터 타입

Proto Type	C++ Type	Java/Kotlin Type <sup>[1]</sup>	Python Type <sup>[3]</sup>	Go Type	Ruby Type	C# Type	PHP Type	Dart Type	Rust Type
double	double	double	float	float64	Float	double	float	double	f64
float	float	float	float	float32	Float	float	float	double	f32
int32	int32_t	int	int	int32	Fixnum or Bignum (as required)	int	integer	int	i32
int64	int64_t	long	int/long <sup>[4]</sup>	int64	Bignum	long	integer/string <sup>[6]</sup>	Int64	i64
uint32	uint32_t	int <sup>[2]</sup>	int/long <sup>[4]</sup>	uint32	Fixnum or Bignum (as required)	uint	integer	int	u32
uint64	uint64_t	long <sup>[2]</sup>	int/long <sup>[4]</sup>	uint64	Bignum	ulong	integer/string <sup>[6]</sup>	Int64	u64
sint32	int32_t	int	int	int32	Fixnum or Bignum (as required)	int	integer	int	i32
sint64	int64_t	long	int/long <sup>[4]</sup>	int64	Bignum	long	integer/string <sup>[6]</sup>	Int64	i64
fixed32	uint32_t	int <sup>[2]</sup>	int/long <sup>[4]</sup>	uint32	Fixnum or Bignum (as required)	uint	integer	int	u32
fixed64	uint64_t	long <sup>[2]</sup>	int/long <sup>[4]</sup>	uint64	Bignum	ulong	integer/string <sup>[6]</sup>	Int64	u64
sfixed32	int32_t	int	int	int32	Fixnum or Bignum (as required)	int	integer	int	i32
sfixed64	int64_t	long	int/long <sup>[4]</sup>	int64	Bignum	long	integer/string <sup>[6]</sup>	Int64	i64
bool	bool	boolean	bool	bool	TrueClass/FalseClass	bool	boolean	bool	bool
string	std::string	String	str/unicode <sup>[5]</sup>	string	String (UTF-8)	string	string	String	ProtoString
bytes	std::string	ByteString	str (Python 2), bytes (Python 3)	[]byte	String (ASCII-8BIT)	ByteString	string	List	ProtoBytes

## gRPC란?

- Google에서 개발한 오픈소스 RPC(Remote Procedure Call) 프레임워크
- HTTP/2를 전송 프로토콜로 사용
- Protocol Buffers를 데이터 형식으로 사용

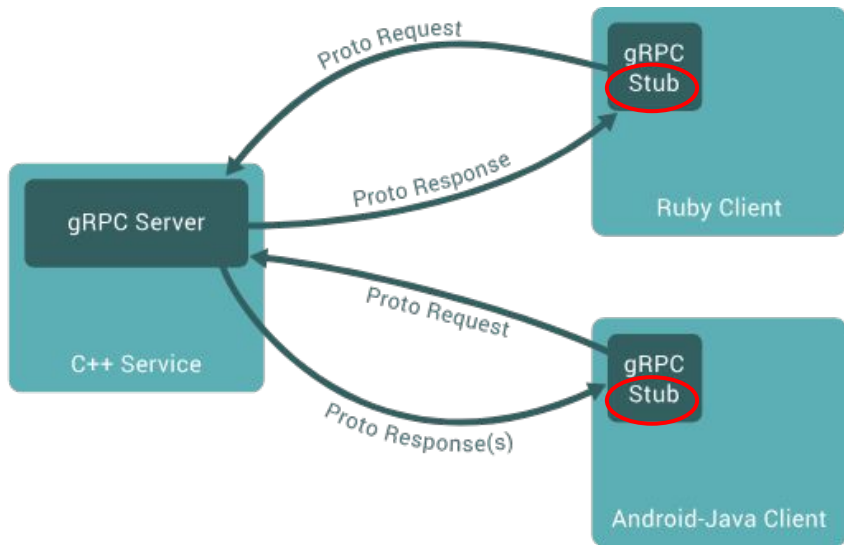
## 주요 특징:

- 양방향 스트리밍 지원
- 다양한 프로그래밍 언어 지원 (C++, C#, Java, Go 등)
- 높은 성능과 낮은 지연 시간
- 멀티플렉싱 지원 (단일 연결로 여러 요청 처리)

## stub란 무엇인가요 ?

**stub** = gRPC 클라이언트가 서버의 메서드를 호출할 수 있도록 만든 **로컬 대리 객체 (프록시)**

- gRPC는 클라이언트와 서버가 **원격 함수 호출(RPC)** 을 통해 데이터를 주고받습니다.
- 이때, 클라이언트는 실제로 서버와 직접 통신하는 것이 아니라, **gRPC가 자동으로 생성해주는 stub 객체**를 통해 서버 함수를 호출합니다.
- stub는 **gRPC 라이브러리**와 **.proto** 파일을 바탕으로 생성됩니다.



## Stub의 종류

1. **블로킹 / 동기식 Stub (Blocking/Synchronous):**
  - 클라이언트에서 호출하면 서버 응답을 받을 때까지 기다림 .
2. **비동기식 Stub (Asynchronous):**
  - 호출 후 바로 리턴하고, 응답은 **콜백** 또는 **Future/Task**로 처리.
3. **스트리밍 Stub (Streaming):**
  - 클라이언트 ↔ 서버 간 **스트리밍 데이터 송수신** 을 지원.



## stub의 역할

### 메서드 인터페이스 제공

- 서버에서 정의한 서비스 함수들을 클라이언트에서 호출할 수 있는 로컬 메서드 형태로 노출합니다.

### 직렬화/역직렬화

- 요청 메시지를 Protocol Buffers로 직렬화하고, 응답 메시지를 자동으로 역직렬화해줍니다.

### 네트워크 통신 처리

- 클라이언트와 서버 간에 gRPC 메시지 교환을 위한 HTTP/2 통신을 자동으로 수행합니다.

### 에러 핸들링

- 호출 결과의 성공/실패 여부를 Status 객체를 통해 전달합니다.

## C++ 클라이언트

```
cpp
std::unique_ptr<Greeter::Stub> stub = Greeter::NewStub(channel);
HelloRequest request;
request.set_name("철수");

HelloReply reply;
grpc::ClientContext context;

grpc::Status status = stub->SayHello(&context, request, &reply);
if (status.ok()) {
    std::cout << "서버 응답: " << reply.message() << std::endl;
}
```

## C# 클라이언트

```
csharp
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greeter.GreeterClient(channel);

var request = new HelloRequest { Name = "영희" };
var reply = await client.SayHelloAsync(request);

Console.WriteLine($"서버 응답: {reply.Message}");
```

클라이언트 작업

순서

1. channel 생성
2. stub 생성
3. 함수 호출

# RPC 비교

# RPC 비교

구분	gRPC	JSON-RPC	Fast Binary Encoding (FBE)	Cysharp / MagicOnion
전송 프로토콜	HTTP/2 기본, HTTP/3 점진적 지원	전송 독립(HTTP, WebSocket, TCP 등)	직접 선택(TCP/UDP, Zero-MQ 등) – 전송 스택 직접 구성	gRPC(HTTP/2) 기반
직렬화 형식	Protocol Buffers (바이너리)	JSON(텍스트)	자체 바이너리 포맷(가변 길이, zero-copy)	MessagePack + LZ4
IDL/스키마	`.proto` 필수	명시적 IDL 없음(선택적 JSON Schema)	FBE IDL(코드 생성기 포함)	C# interface(Code-First), `.proto` 불필요
스트리밍 지원	서버·클라이언트·양방향 스트리밍	사양에 없음(확장 구현 가능)	가능하지만 직접 구현	StreamingHub로 양방향 실시간 통신
언어·플랫폼	13+ 언어, 브라우저는 gRPC-Web	언어 무관(라이브러리 다양)	C++, C#, Go, Java, JS, Python 등	서버: .NET 8+, 클라이언트: .NET 4.6.1~8, Unity 2022 LTS+
성능 특성	REST/JSON 대비 3-10배 빠른 사례	JSON 파싱 오버헤드로 gRPC보다 느림	C++ 기준 직렬화 $\approx 10$ Mops/s	gRPC와 유사 또는 소폭 낮음
생태계·툴링	로드밸런싱, 트레이싱, mTLS, 관찰성 풍부	가볍지만 표준 툴 부족	전용 툴 제한, 커뮤니티 작음	Unity/게임 특화, Prometheus Exporter
최근 동향	HTTP/3 지원, .NET 9 최적화 진행	사양 2.0 이후 큰 변화 없음	2023-11 릴리스(버그 픽스)	2025-04 릴리스 7.0.3(.NET 7 지원 종료)
추천 시나리오	멀티-언어 마이크로서비스, 모바일·IoT	간단한 제어 채널, CLI↔데몬	초저지연 트레이딩·게임 서버 내부 버스	Unity 게임·실시간 .NET 앱
Site	<a href="https://grpc.io/">https://grpc.io/</a>	<a href="https://www.jsonrpc.org/">https://www.jsonrpc.org/</a>	<a href="https://github.com/chronoxor/FastBinaryEncoding">https://github.com/chronoxor/FastBinaryEncoding</a>	<a href="https://github.com/Cysharp/MagicOnion">https://github.com/Cysharp/MagicOnion</a>



# RPC 비교

## 1. gRPC

- **장점**
  - HTTP/2 멀티플렉싱 + 바이너리(Protobuf)로 낮은 레이턴시와 작은 페이로드.
  - 인터셉터, 미들웨어, 로드밸런싱, TLS/mTLS 등 엔터프라이즈 기능 내장.
  - 서버·클라이언트·양방향 스트리밍, **deadline**, 헤더/트레일러 지원.
  - 최근 HTTP/3 실험 지원으로 모바일-무선 환경 지연 감소 기대 [catchpoint](#).
- **주의할 점**
  - 브라우저는 gRPC-Web(HTTP/1.1) 프록시 필요.
  - **.proto** 관리 및 **Code-Gen** 워크플로가 익숙하지 않다면 러닝커브.
- **추천 시나리오** : 멀티-언어 마이크로서비스, 고성능 백엔드-백엔드 통신, 모바일-IoT.

# RPC 비교

## 2. JSON-RPC

- **장점**
  - 스펙이 매우 단순해 구현·디버깅이 쉽고, 텍스트 기반으로 가시성 높음 [jsonrpc](#).
  - 전송 프로토콜 제한이 없어 **WebSocket**, **IPC**, **HTTP** 등 어디든 적용 가능.
  - 블록체인(이더리움, 비트코인) 노드 **API**처럼 “가벼운 제어 채널”에 널리 사용.
- **제한**
  - 스트리밍·버전 관리·스키마 검증은 사양 외; 직접 설계해야 함.
  - **JSON** 파싱 오버헤드로 대용량 실시간 통신에는 부적합.
- **추천 시나리오** : 간단한 원격 제어, **CLI** ↔ 데몬, 브라우저 ↔ 백엔드 통신에서 “가볍고 사람이 읽기 쉬운” **API**가 필요할 때.

# RPC 비교

## 2. JSON-RPC

- **장점**
  - 스펙이 매우 단순해 구현·디버깅이 쉽고, 텍스트 기반으로 가시성 높음 [jsonrpc](#).
  - 전송 프로토콜 제한이 없어 **WebSocket**, **IPC**, **HTTP** 등 어디든 적용 가능.
  - 블록체인(이더리움, 비트코인) 노드 **API**처럼 “가벼운 제어 채널”에 널리 사용.
- **제한**
  - 스트리밍·버전 관리·스키마 검증은 사양 외; 직접 설계해야 함.
  - **JSON** 파싱 오버헤드로 대용량 실시간 통신에는 부적합.
- **추천 시나리오** : 간단한 원격 제어, **CLI** ↔ 데몬, 브라우저 ↔ 백엔드 통신에서 “가볍고 사람이 읽기 쉬운” **API**가 필요할 때.

# RPC 비교

## 4. Cysharp / MagicOnion

- 특징
  - gRPC + MessagePack을 C# 인터페이스(Code-First)로 감싸 .NET 개발자가 **.proto 없이** 개발 가능.
  - StreamingHub API로 서버↔클라이언트 양방향 실시간 통신(채팅, 멀티플레이) 제공.
  - Unity(IL2CPP)·Blazor·MAUI 등 클라이언트 SDK와 ASP.NET Core 호환.
  - 2025-04-14 릴리스 7.0.3부터 **.NET 8 이상 필수, .NET 7 지원 종료** [github](#).
- 장점
  - Unity/게임 개발자 친화적: 하나의 C# 인터페이스로 서버·클라이언트 동시 코드 생성.
  - MessagePack + LZ4로 JSON-REST 대비 수십 배 작은 페이로드.
  - Prometheus Exporter 내장으로 메트릭 수집 용이 [grafana](#).
- 제한
  - .NET 생태계에 종속; 다언어 호출은 순정 gRPC 사용 권장.
  - 일부 벤치마크에서 순정 gRPC 대비 약간 느린 결과 보고 [reddit](#) (주로 MessagePack 인코딩/디코딩 오버헤드).
- 추천 시나리오 : Unity 게임 백엔드, .NET 실시간 협업 앱, SignalR 대안이 필요할 때.

# RPC 비교

## 선택 가이드

1. 멀티-언어, 대규모 마이크로서비스 → **gRPC**
2. 가볍고 사람이 읽기 쉬운 제어 **API** → **JSON-RPC**
3. 극단적 성능( $\mu$ s, 수백 MB/s+) & 자체 전송 스택 보유 → **FBE**
4. **Unity/.NET** 생태계, 실시간 스트리밍, 코드-퍼스트 → **MagicOnion**

## 참고 자료

- SEMI E179 표준 문서
- Protocol Buffers 공식 문서: <https://protobuf.dev/>
- gRPC 공식 문서: <https://grpc.io/docs/>
- SEMI 표준 관련 정보: <https://www.semi.org/en/standards>



# gRPC Conf

## 2024

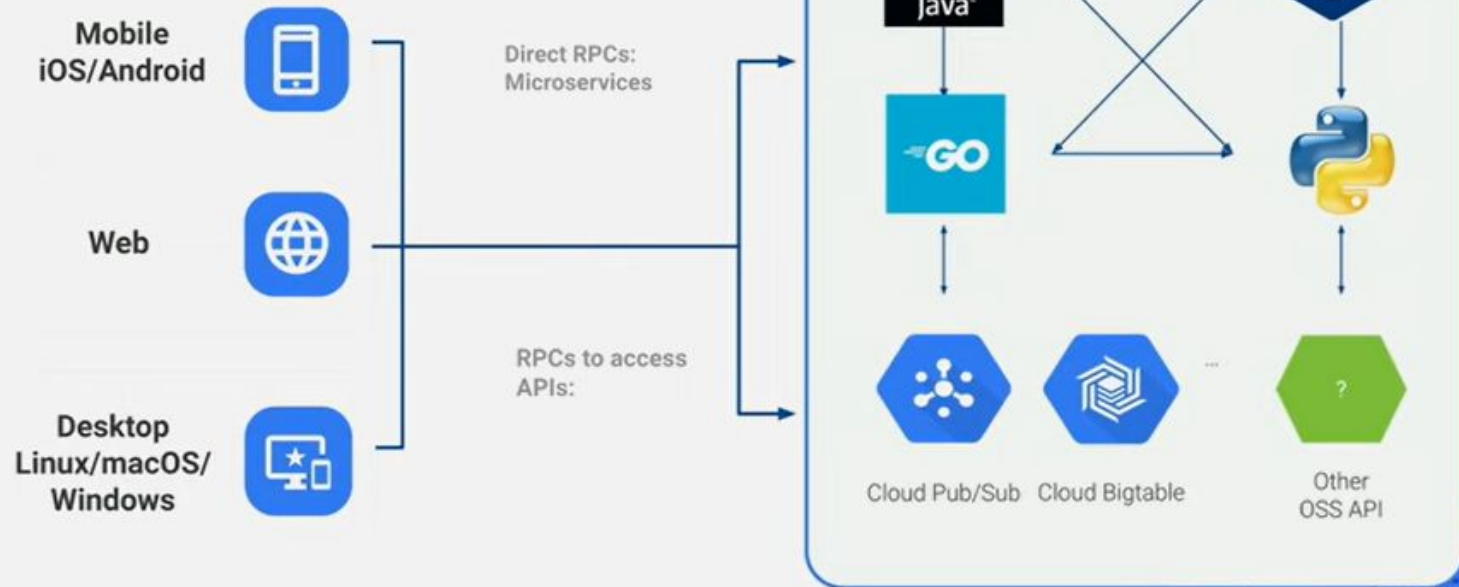
<https://www.youtube.com/@grpcio>

## gRPC Overview

*Yifei (Ivy) Zhuang, Software Engineer, Google  
gRPC maintainer (Java)*



# gRPC Use Cases



# Why gRPC is outstanding

Multi Languages  
Multi Platform

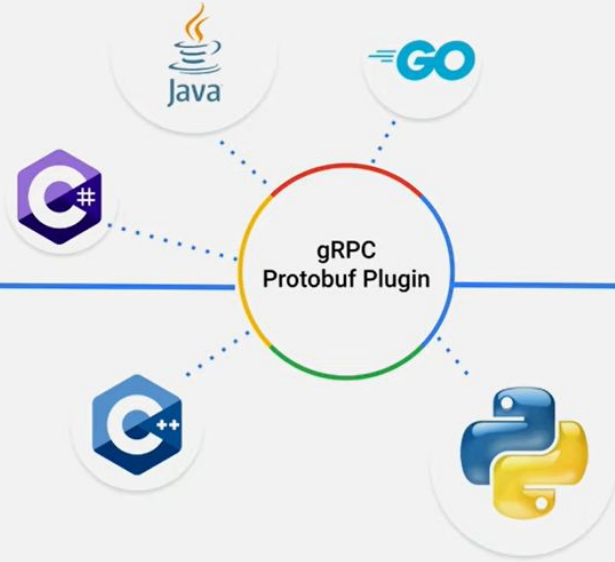
High  
Performance

Pluggable  
Transport  
Serialization  
Interceptors

Feature Rich  
Suitable for  
microservices



# gRPC Key Design Decisions



**High Performance**  
Binary encoding  
Efficient parsing  
Reduces message sizes

Java  
Go  
Python  
C  
C++  
Python  
Ruby  
Node  
Dart  
Objective-C  
PHP  
C#  
Kotlin  
**Rust coming soon**  
Windows  
Linux  
MacOS  
Web  
iOS  
Android  
Mobile  
Vehicle  
AWS  
Azure  
...

# gRPC Key Design Decisions - HTTP/2

## HTTP/2

### Improvements

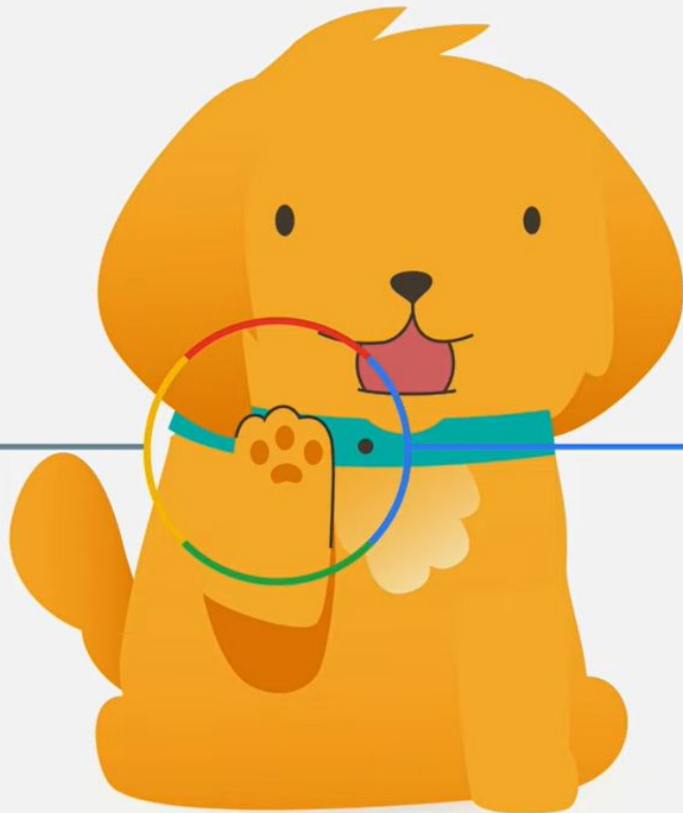
- Reduce TCP connections
- Binary
- Header compression

### High Performance

- Reduce latency
- Resource utilization

### Compatible

- With load balancers and proxies



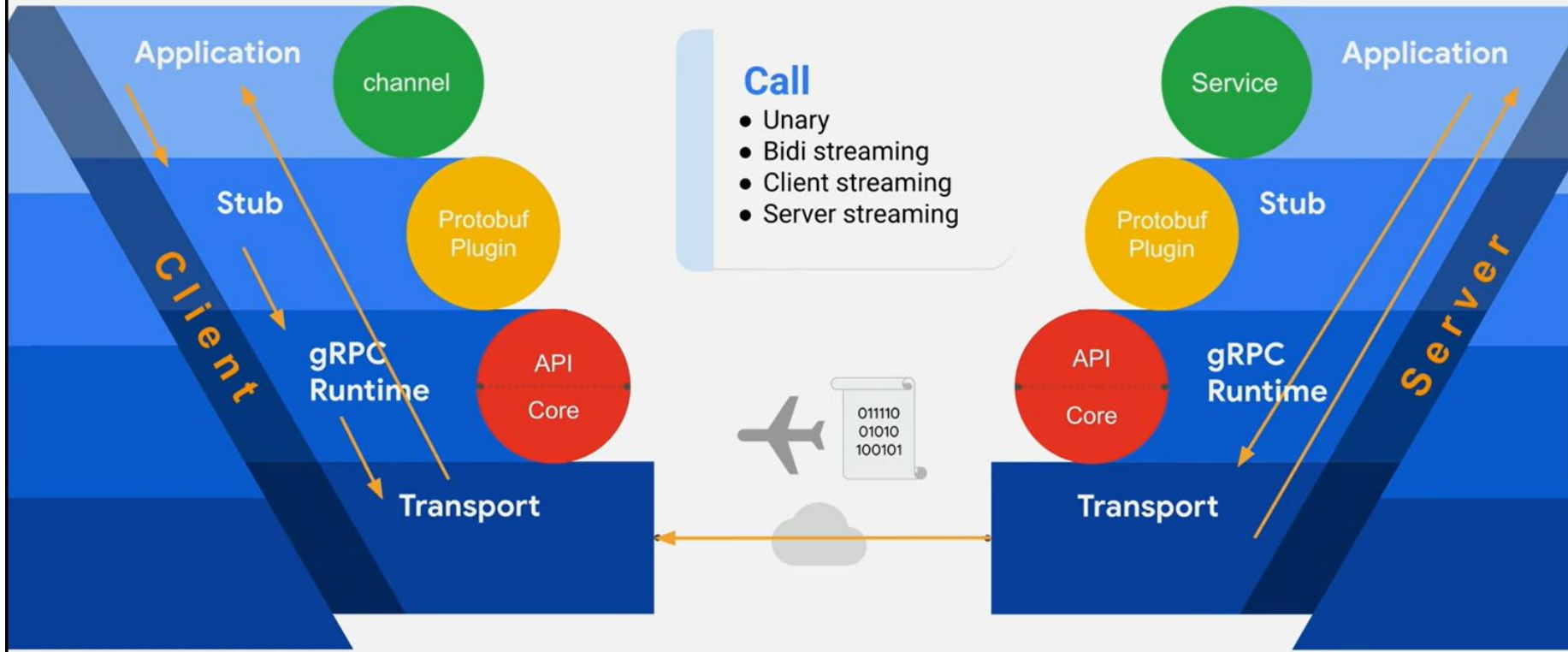
# Channel

It is an abstract of the endpoint you can send or receive messages.



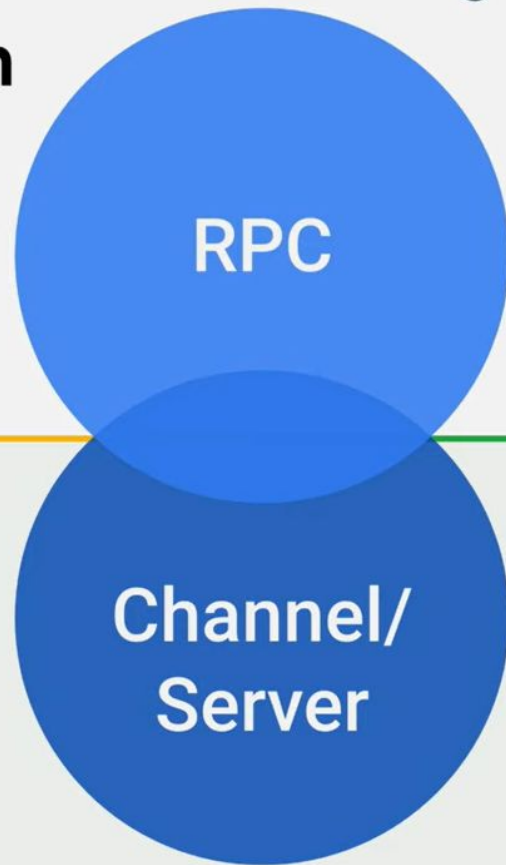
```
ManagedChannel channel = Grpc.newChannelBuilder("dns:///example.com",  
    InsecureChannelCredentials.create()).build();
```

# gRPC Life Cycle



# gRPC Life Cycle - Termination

- Normal/abnormal termination
- Client and server generally agree, client may see more errors



```
//preexisting calls continue, new calls cancelled  
channel.shutdown();  
  
//preexisting and new calls cancelled  
channel.shutdownNow();  
  
channel.awaitTermination(1, TimeUnit.MINUTES);  
server.awaitTermination(1, TimeUnit.MINUTES);
```

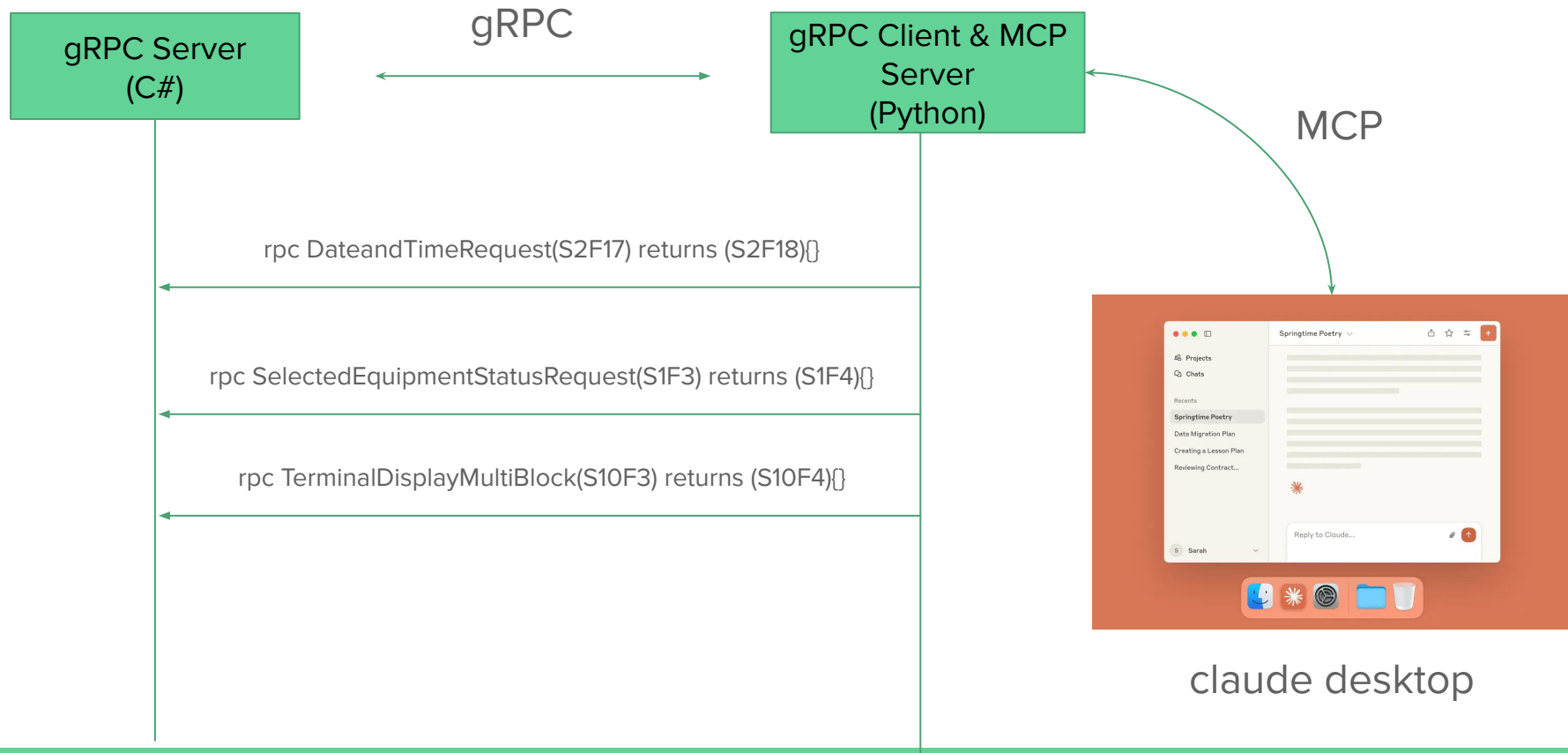
# Demo



# Demo Overview



for Test



# Demo Overview

Form1

IP  Port


EQP STATUS

TerminalMsg

이제 그만 쉬고 일해라

Claude — Control+Alt+Space





Equipment Status Check ▾

 장비의 현재 상태가 "STOP"이면 "이제 그만 쉬고 일해라" 라는 터미널 메시지를 보내주고 "RUN" 상태이면 "그래 열심히 해라. 수고" 라는 터미널 메시지를 보내주세요.

장비의 현재 상태를 확인한 후 상태에 맞는 메시지를 보내드리겠습니다.

현재 장비의 상태는 "STOP"입니다. 요청하신 대로 "이제 그만 쉬고 일해라" 메시지를 보내드리겠습니다.

메시지가 성공적으로 전송되었습니다. 장비가 "STOP" 상태이므로 "이제 그만 쉬고 일해라"라는 메시지를 보냈습니다.

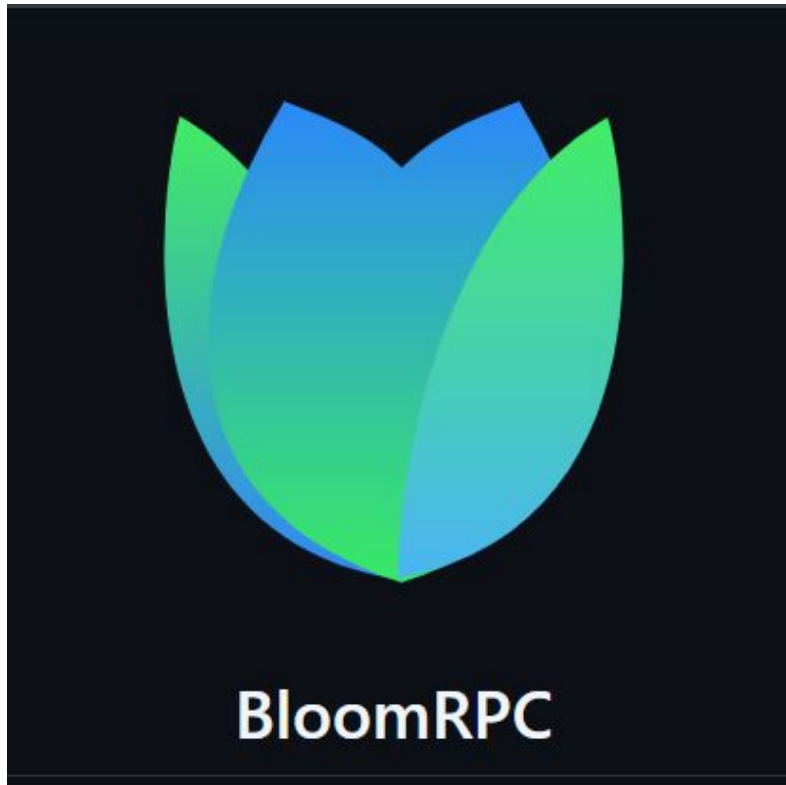
    재시도 ▾

Claude는 실수를 할 수 있습니다. 응답을 반드시 다시 확인해 주세요.

Claude에게 답변하기

Claude 3.7 Sonnet ▾

# gRPC client gui tool



- .proto 파일을 드래그 앤 드롭으로 쉽게 불러올 수 있음
- 서비스와 메서드를 트리 구조로 시각적으로 보여줌
- 요청/응답 데이터를 JSON 형식으로 쉽게 편집 가능
- 단방향(Unary) 및 스트리밍 호출 모두 지원
- 메타데이터 추가 및 TLS/SSL 설정 가능
- 여러 환경에서의 테스트를 위한 워크스페이스 관리 기능 제공
- 이러한 기능들을 통해 개발자는 gRPC 서비스를 쉽게 테스트하고 디버깅할 수 있습니다.

BloomRPC는 오픈소스 프로그램으로 github에서 다운로드 받을 수 있습니다. <https://github.com/bloomrpc/bloomrpc>

# Claude Desktop (MCP client) <https://claude.ai/download>



시작하기

## 데스크톱에서 Claude를 만나보 세요 BETA

필요할 때 언제든지 곁에 있습니다—Claude는 작업 흐름 속에서 조용히 기다립니다. 탭 전환이 필요합니다.

Windows

Windows (arm64)

macOS

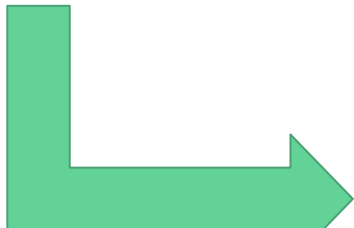


# proto 파일 만들기

```
<S10F5 P Terminal Display, Multi-Block (VTN)
  <LIST 2
    <BINARY 1 TID>
    <LIST n
      <ASCII 255 TEXT>
    >
  >
>

<S10F6 S Terminal Display, Multi-Block Acknowledge (VMA)
  <BINARY 1 ACKC10>
>
```

SML 파일을 proto 파일로 변환

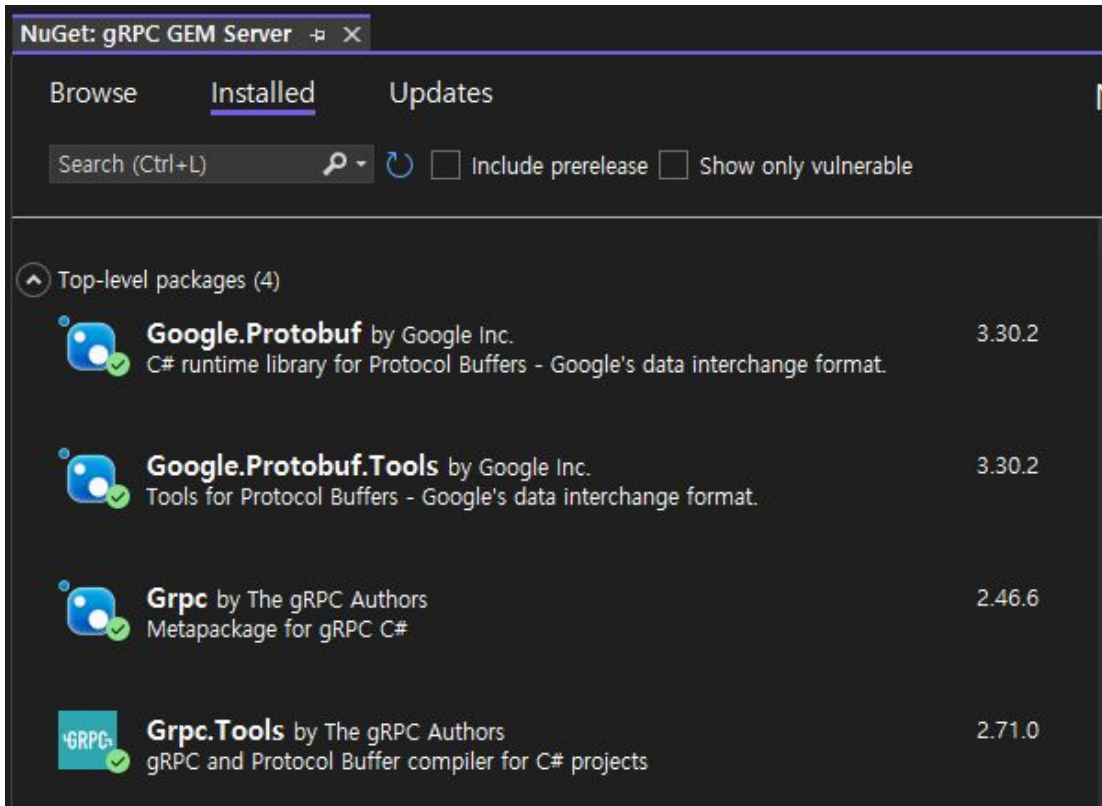


> sml 파일과 proto 파일을 학습해서 sml에  
해당하는 나머지 proto 파일을 완성해줘.

```
SEMI_GEM.proto x
1  syntax = "proto3";
2
3  package SEMI_GEM;
4
5  import "google/protobuf/descriptor.proto";
6  import "google/protobuf/empty.proto";
7
8  import "protos/semi_e179-1224.proto";
9
10 service SEMI_GEM_Service{
11   > ...
12
13   // Stream 10
14   > ...
15
16   // HOST → EQ
17   rpc TerminalDisplayMultiBlock (S10F5) returns (S10F6) {}
18 }
19
20 > ...
21
22 // Terminal Display Multi-Block
23 message S10F5 {
24   int32 TID = 1;
25   repeated string TEXT_LIST = 2;
26 }
27
28 message S10F6{
29   int32 ACKC10 = 1;
30 }
```

# C# gRPC Server

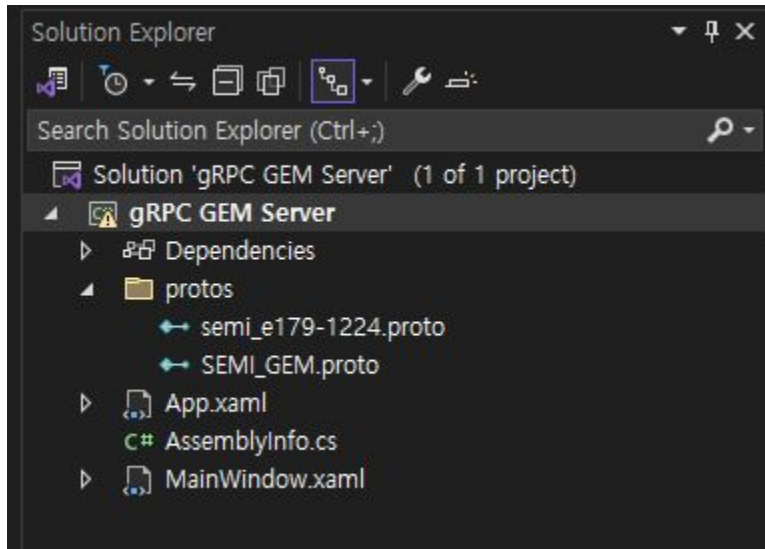
# C# gRPC Server (개발환경 설정)



C# 프로젝트 생성 후 다음

4가지 패키지를 설치

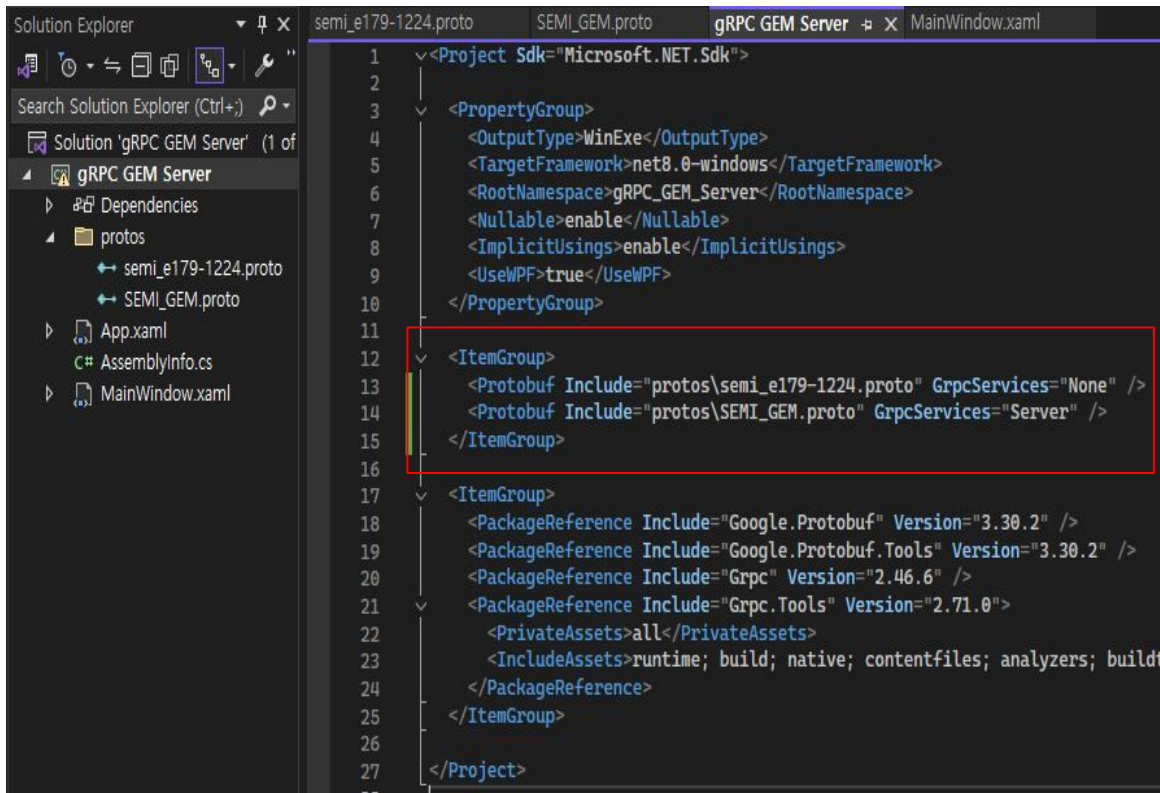
# C# gRPC Server (개발환경 설정)



1. project 폴더 아래에 protos 폴더 생성
2. proto 파일을 복사해 놓음



# C# gRPC Server (개발환경 설정)



C#의 경우 proto 파일을 프로젝트에 포함 시킨 후 Gpc.Tools 패키지를 사용하면 컴파일 후 코드가 자동 생성 됨.

1. 프로젝트 설정
2. ItemGroup에 Protobuf 추가
3. Build

# C# gRPC Server (procedure implement)

ServiceBase를 상속 받아 Service를 위한 함수 작성

```
1 reference
public class My_GEM_gRPC_Service : SEMI_GEM_Service.SEMI_GEM_ServiceBase
{
    // terminal display multiblock
    3 references
    public override Task<S10F6> TerminalDisplayMultiBlock(S10F5 request, ServerCallContext context)
    {
        // Process the terminal display multiblock request and prepare the response
        Program.mainForm.DisplayTerminal(request.TEXTLIST.ToList());

        S10F6 response = new S10F6
        {
            ACKC10 = 0
        };

        return Task.FromResult(response);
    }
}
```

# C# gRPC Server (Run/Stop Server)

```
4 references
public partial class fmMain : Form
{
    private Grpc.Core.Server rpcServer;
    private ConcurrentDictionary<int, string> _v = new ConcurrentDictionary<int, string>();

    1 reference
    public fmMain() ...

    1 reference
    private void btnStart_Click(object sender, EventArgs e)
    {
        if (rpcServer != null)
        {
            if (MessageBox.Show("gRPC server is already running. Do you want shutdown?") == DialogResult.OK)
            {
                rpcServer?.ShutdownAsync().Wait();
                rpcServer = null;
                btnStart.Text = "START";
            }
        }
        else
        {
            rpcServer = new Grpc.Core.Server
            {
                Services = { SEMI_GEM_Service.BindService(new My_GEM_gRPC_Service()) },
                Ports = { new ServerPort(tbIPAddr.Text, int.Parse(tbPort.Text), ServerCredentials.Insecure) }
            };
            rpcServer.Start();
            btnStart.Text = "STOP";
        }
    }
}
```

`insecure`는 TLS(SSL)를 사용하지 않고 통신하는 설정을 의미

# MCP Server & gRPC Client

# MCP Server 만들기 - uv 설치

```
# install uv
```

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

# MCP Server 만들기 - 프로젝트 생성 및 가상환경 셋업

# Create a new directory for our project

```
uv init secsgem  
cd secsgem
```

# Create virtual environment and activate it

```
uv venv  
.venv\Scripts\active
```

# Install dependencies

```
uv add mcp[cli] httpx
```

# Create our server file

```
new-item secsgem.py
```

# MCP 서버 만들기 - gRPC client 구현

# Install grpc, protobuf packages

```
uv pip install grpcio protobuf grpcio-tools
```

# Compile proto files

```
python -m grpc_tools.protoc \  
    --proto_path=protos \  
    --python_out=. \  
    --grpc_python_out=. \  
    protos/SEMI_GEM.proto protos/semi_e179-1224.proto
```

# secsgem.py

```
from typing import Any
import grpc
import asyncio
from mcp.server.fastmcp import FastMCP
from datetime import datetime
from protos import semi_gem_pb2, semi_gem_pb2_grpc

# SECSGEM MCP 서버 초기화
mcp = FastMCP("secsgem")

# gRPC 클라이언트 설정을 위한 상수
GRPC_SERVER_ADDRESS = "127.0.0.1:50051" # gRPC 서버 주소
```



```

# gRPC 연결을 위한 설정
class GRPCClient:
    def __init__(self, address: str):
        """gRPC 클라이언트 초기화"""
        self.address = address
        self.channel = None
        self.stub = None

    async def connect(self):
        """gRPC 서버에 연결"""
        try:
            self.channel = grpc.aio.insecure_channel(self.address)
            self.stub = semi_gem_pb2_grpc.SEMI_GEM_ServiceStub(self.channel)
            print(f"Successfully connected to gRPC server at {self.address}")
        except Exception as e:
            print(f"Failed to connect to gRPC server: {str(e)}")

    async def close(self):
        """gRPC 연결 종료"""
        if self.channel:
            await self.channel.close()
            print("gRPC connection closed")

    async def get_equipment_status(self) -> str:
        """gRPC를 통해 장비의 상태 데이터 요청 (S1F3/S1F4)"""
        try:
            request = semi_gem_pb2.S1F3(
                svid_list=[1] # VID 1번 요청
            )
            response = await self.stub.SelectedEquipmentStatusRequest(request)
            return f"Equipment Status: {response.sv_list[0]}"
        except Exception as e:
            return f"Error getting equipment status: {str(e)}"

# gRPC 클라이언트 인스턴스 생성
grpc_client = GRPCClient(GRPC_SERVER_ADDRESS)

```

```
@mcp.tool()
async def get_equipment_time() -> str:
    """장비의 현재 시간을 가져옵니다.

    반환값:
        str: ISO 8601 형식의 시간 문자열 (YYYY-MM-DDThh:mm:ss.sss)
    """
    # gRPC 클라이언트가 연결되어 있지 않으면 연결
    if grpc_client.stub is None:
        await grpc_client.connect()

    # 장비 시간 요청
    return await grpc_client.get_date_time()
```

```
@mcp.tool()
async def send_terminal_msg(msg : str) -> str:
    """장비에 터미널 메시지를 전달 합니다.

    매개변수:
        msg (str): 장비에 전송할 메시지

    반환값:
        str: 메시지 전송 결과
    """
    # gRPC 클라이언트가 연결되어 있지 않으면 연결
    if grpc_client.stub is None:
        await grpc_client.connect()

    # 터미널 메시지 전송
    return await grpc_client.send_terminal_message(msg)
```

```
if __name__ == "__main__":  
    print("Starting SECSGEM MCP server...")  
    # 서버 초기화 및 실행  
    mcp.run(transport='stdio')
```

# MCP Server 만들기 - claude desktop에 설치

```
{  
  "mcpServers": {  
    "secsgem": {  
      "command": "uv",  
      "args": [  
        "--directory",  
        "C:\\ABSOLUTE\\PATH\\TO\\PARENT\\FOLDER\\secsgem",  
        "run",  
        "secsgem.py"  
      ]  
    }  
  }  
}
```

# MCP Server 만들기 - 참고

MCP Server 만들기 튜토리얼 (영어)

<https://modelcontextprotocol.io/quickstart/server>

MCP Server 공식 샘플코드 (영어)

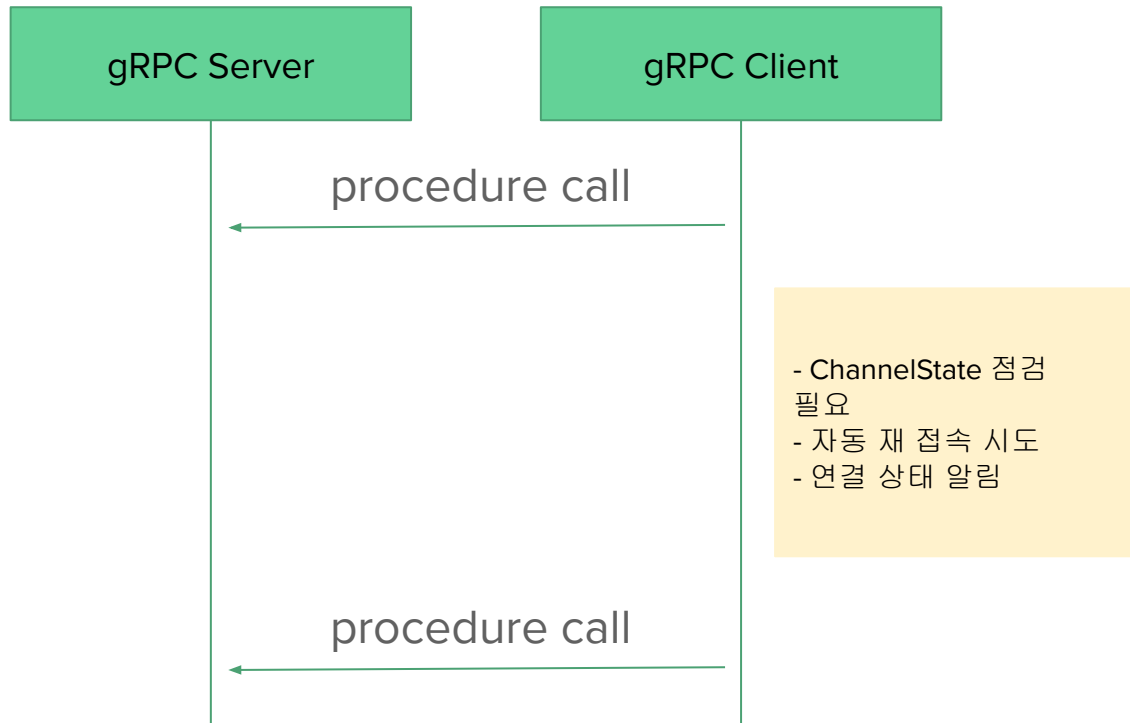
<https://github.com/modelcontextprotocol>

MCP 소개 (한글)

<https://wikidocs.net/268790>

# gRPC 실무 적용 시 고려 사항

# Health Monitoring



```
namespace Grpc.Core
{
    /// <summary>
    /// Connectivity state of a channel.
    /// Based on grpc_connectivity_state from grpc/grpc.h
    /// </summary>
    public enum ChannelState
    {
        /// <summary>
        /// Channel is idle
        /// </summary>
        Idle,

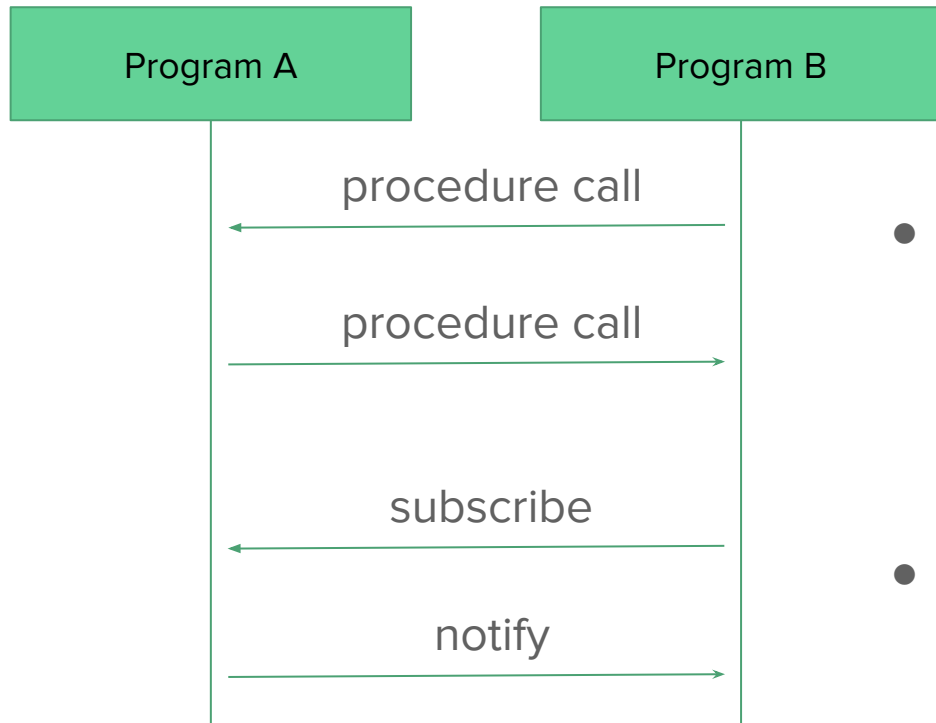
        /// <summary>
        /// Channel is connecting
        /// </summary>
        Connecting,

        /// <summary>
        /// Channel is ready for work
        /// </summary>
        Ready,

        /// <summary>
        /// Channel has seen a failure but expects to recover
        /// </summary>
        TransientFailure,

        /// <summary>
        /// Channel has seen a failure that it cannot recover from
        /// </summary>
        Shutdown
    }
}
```

# Both Direction Service



- 양방향 모두 Server/Client 구현
- Stream기능을 이용하여 콜백 구현



# Field Network Test



# Security



구분	결과
암호화	없음(평균)
무결성/위·변조 방지	없음
상대방 인증	없음
사용 권장 범위	로컬 테스트, 내부-VPC 트래픽, 이미 mTLS 프록시가 감싸는 환경 등

## 1. 'Insecure 모드'란?

gRPC는 기본적으로 **TLS(Transport-Layer Security)** 입니다.

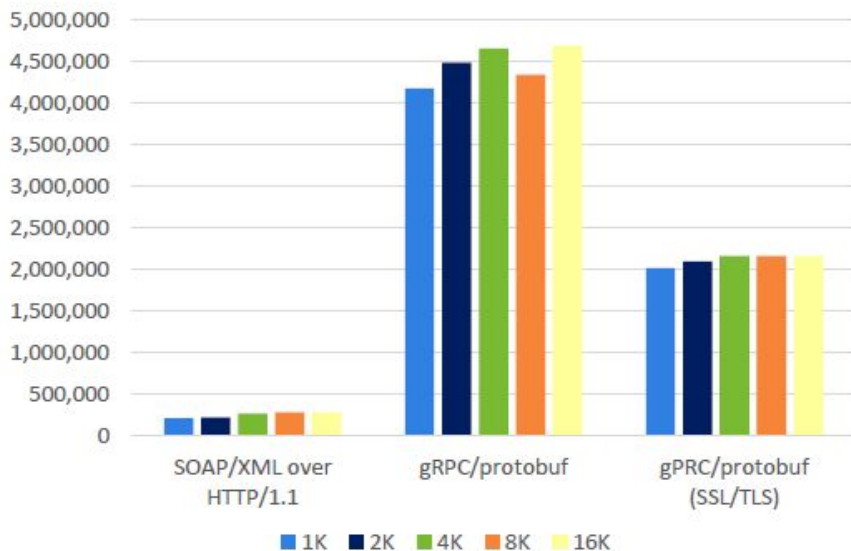
**insecure** 모드를 쓰면 이 TLS 계층을 **완전히 비활성화** → 모든 데이터가 평문으로 흐르고, 서버/클라이언트 신원 확인도 안 됩니다.

## 2. 언제(언제만) 써야 하나?

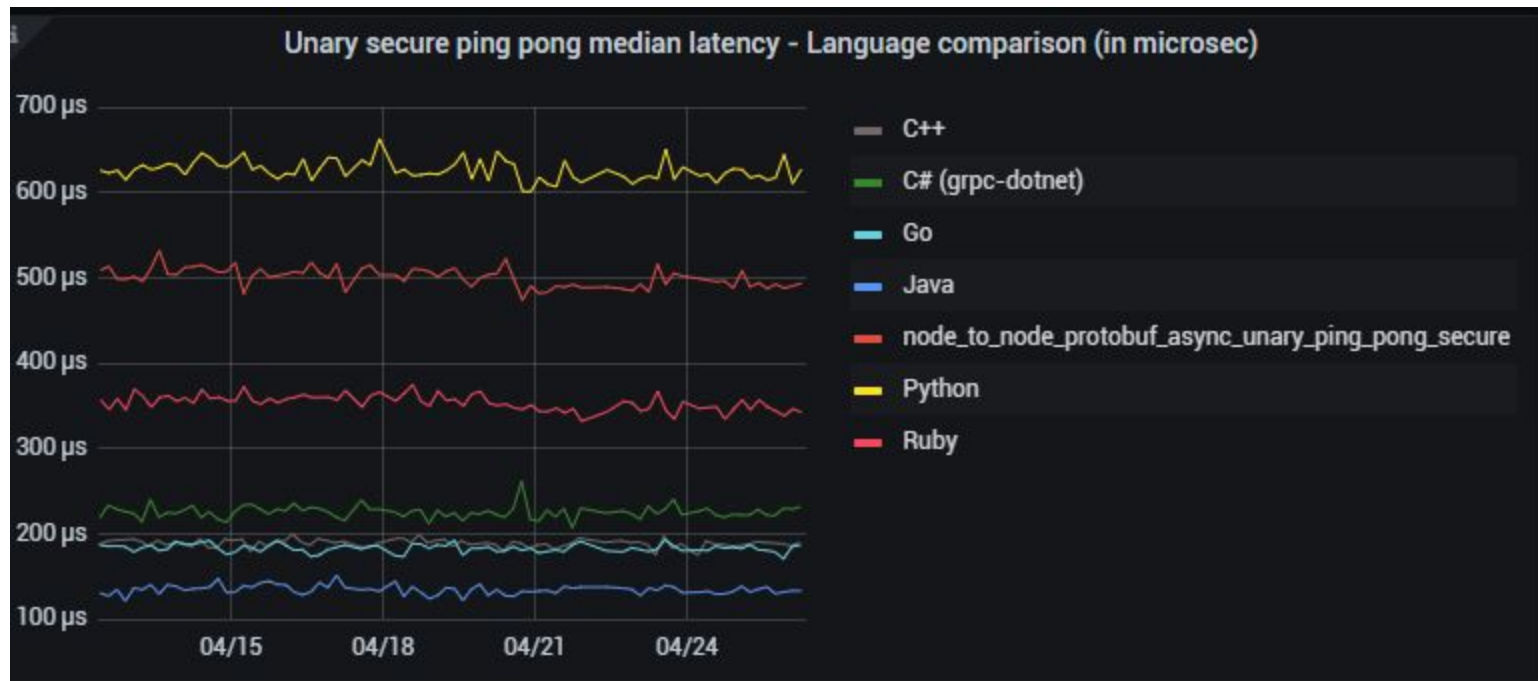
- **단위 테스트 / 로컬 개발**  
빠른 빌드-실행 루프가 필요할 때.
- **서비스-메시/프록시 (mTLS) 내부**  
Istio, Linkerd, Envoy 같은 프록시가 이미 mTLS를 제공 → 애플리케이션 계층에서 중복 암호화를 건너뛰어 성능·복잡도 감소.
- **폐쇄형 사설망**  
물리적으로 외부 접속이 불가능한 사설망 + 네트워크 ACL이 충분히 강력할 때.

그 외 인터넷이나 불특정 네트워크 구간에서 **절대 사용 금지**. 데이터 탈취·위조·중간자 공격에 그대로 노출됩니다. Datadog SAST 를 역시 **WithInsecure()** 사용을 보안 위험으로 표시합니다.

# Performance



# Performance - Language comparison



출처 : <https://grpc.io/docs/guides/benchmarking/>