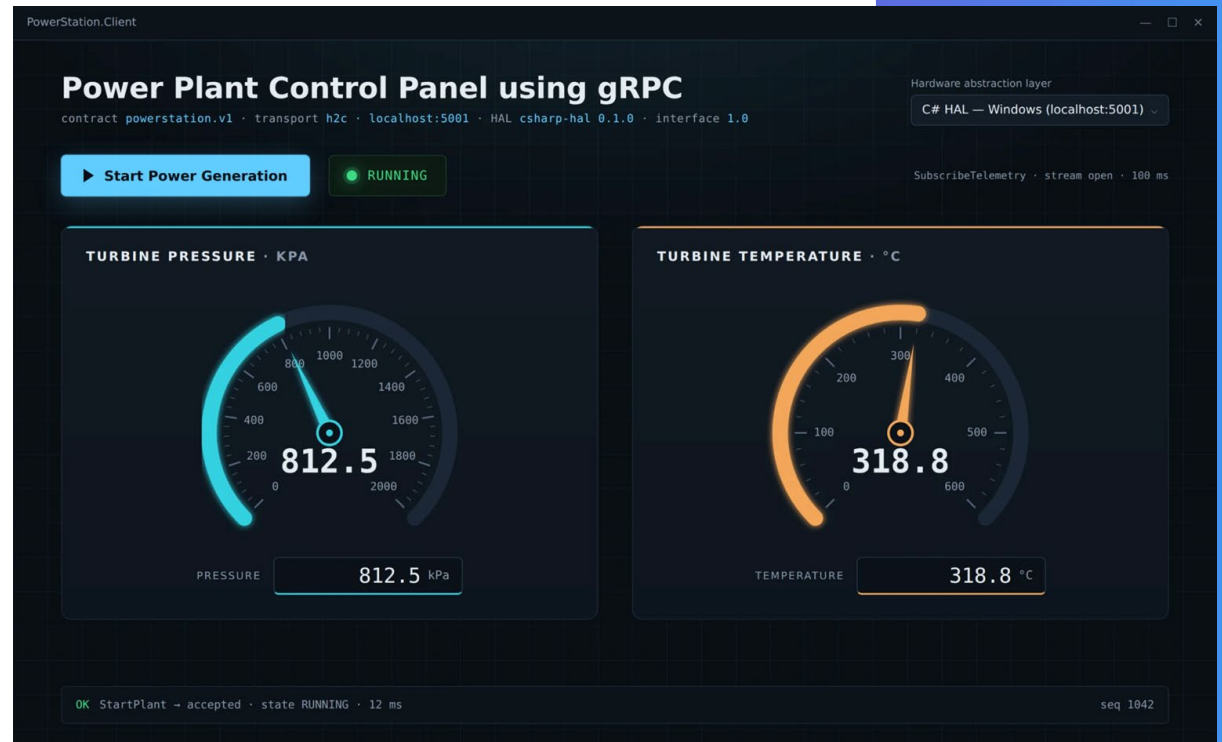




Talking to Hardware With gRPC

A Practical Introduction to gRPC



About ICS

We Build Embedded Systems for What's Next

Accelerating next-gen product development through expert engineering and advanced software solutions.

At a Glance

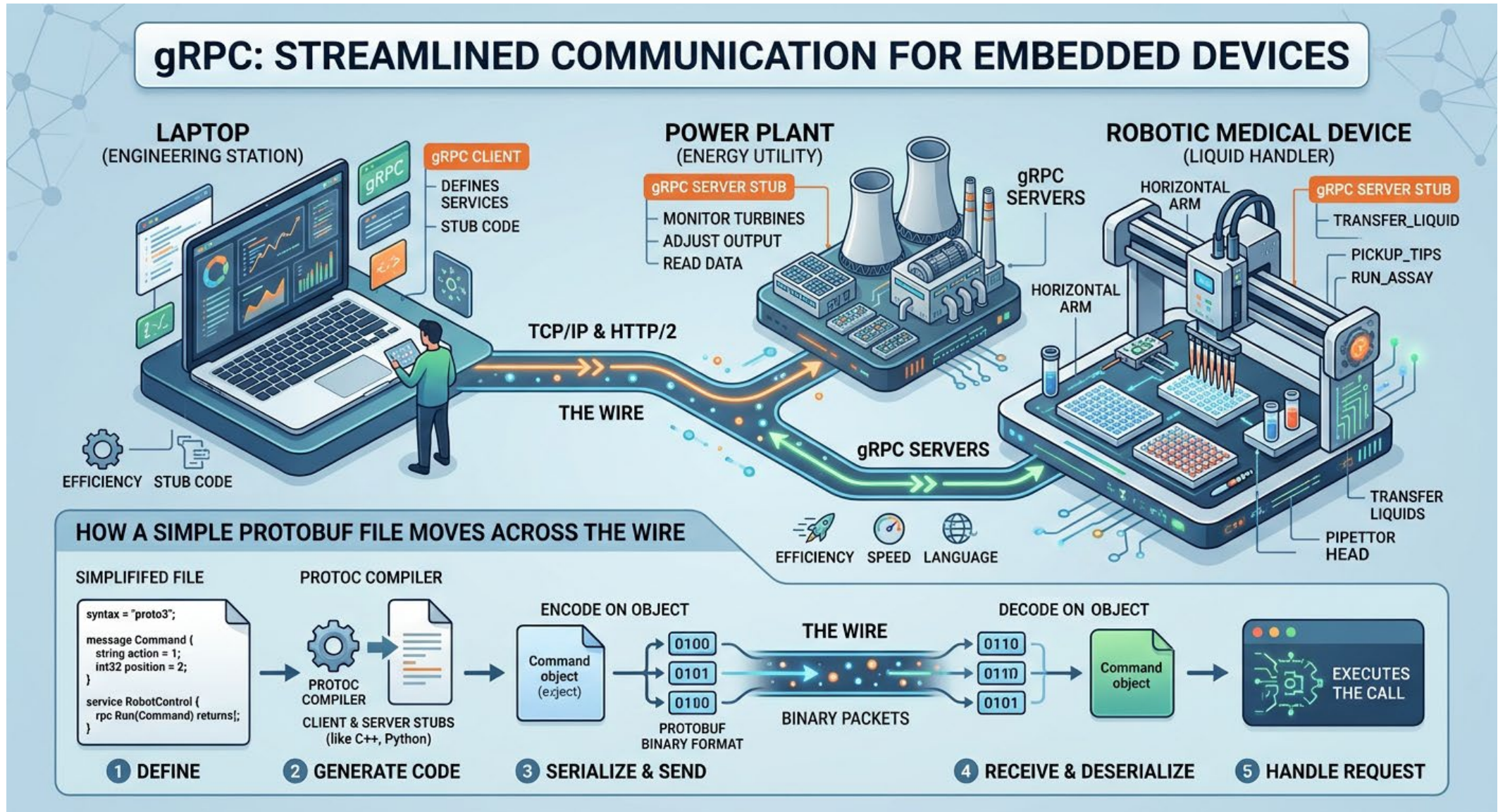
- Founded: 1987
- Headquarters: Waltham, MA (Greater Boston)
- Global Offices: California, Canada, Europe
- Team: 160+ expert engineers & innovators
- Full-stack software development for regulated & high-reliability industries

Centers of Excellence

- Medtech Software
- Software Development
- AI/ML Innovation
- UX/UI Design & Human Factors
- IoT & Embedded Software
- Cybersecurity



Overview of what gRPC can do.



One contract, two generated sides

HAND-ROLLED

Desktop App (client) the WinUI in our demo

Hand-written serialiser

A byte layout on a wiki page

Hand-written parser

Device firmware (server) the HAL in our demo.

Two implementations of one format. They drift.

WITH GRPC

Desktop App (client) the WinUI in our demo

Generated client stub

device.proto

Generated service base

Device firmware (server) the HAL in our demo.

One definition. Both implementations fall out of it.

Where gRPC came from

BEFORE

Stubby

Google's internal RPC framework,
used between its own services.
Never released.

AUGUST 2016

gRPC 1.0

The open-source successor reaches
a stable release and a public API.

TODAY

A CNCF project



Governed outside Google, with
maintained implementations across
many languages.

The design goals are the ones you get from running services at scale: a strict contract, generated code, and a transport that multiplexes.

Protocol Buffers: compact binary, not self -describing

JSON —59 bytes

```
{ "sequence" : 1042 ,  
  "pressureKpa" : 812.5 ,  
  "temperatureC" : 318.75 }
```

Every field name is repeated on every message. Punctuation and digits are bytes too. The upside is that you can read it without a schema.

Protobuf —13 bytes

```
08 92 08  
15 00 20 4B 44  
1D 00 60 9F 43
```

One tag byte carries the field number and the wire type, then the value. No names, no quotes. The .proto tells the reader what the numbers mean.

A **message** is a named record of typed fields. Each field carries a **number**. *The number, not the name, is what goes on the wire.*

Anatomy of a .proto file. The “Rosetta Stone”

```
syntax = "proto3";  
package powerstation.v1;  
option csharp_namespace = "PowerStation";  
message StartRequest {  
    string operator_id = 1;  
}  
  
service PowerStation {  
    rpc StartPlant (StartRequest)  
        returns (StartAck);  
}
```

proto3 is the syntax you should be writing

The package namespaces every type on the wire

Language options shape the generated code only
—they never change the wire format

A message is a typed record; every field gets a
number

A service groups rpcs; each takes one message
and returns one

Field numbers and versioning

Remember! This “1” is not the value of “sequence”, it identifies the field over the wire!

```
message TelemetryFrame {  
  uint32 sequence = 1;  
  // v1: float pressure_psi = 2;  
  reserved 2;  
  reserved "pressure_psi";  
  float pressure_kpa = 3;  
}
```

reserved 2 protects the wire

An old device still sends field 2 meaning psi. If a new field took number 2, the receiver would decode those bytes as the new type —no error, just wrong values.

Safe to ship	Breaks the other side
Add a field with a new number	Change an existing field's number
Stop populating a field	Change an existing field's type
Rename a field in the .proto	Reuse a retired number
Mark a retired number reserved	Delete a field without reserving it

reserved "pressure_psi" protects the readers

The name never travels, so this is not about bytes. It stops anyone bringing the old name back under a new number, where old code, logs and JSON output would silently mean something else.

Both sides ship on their own schedule. The field number is the only thing they agree on at runtime.

Assigning Field Numbers:

<https://protobuf.dev/programming-guides/proto3/#~:text=for%20your%20field.,Assigning%20Field%20Numbers,-You%20must%20give>

Keeping two codebases on one contract

A startup check detects drift between two codebases. It does not prevent it.

1

Do not hash the file text

CRLF vs LF, a trailing newline, a comment:
the hash changes, the bytes do not.
Compare a version the vendor bumps on
purpose, not a checksum of the source.

2

Decide the older -firmware policy on paper

A new GUI will meet a device that has not
been updated yet. Refuse, warn, or run
degraded —and make the error say what to
update.

3

Know whether a change is additive or breaking

Pull the .proto from a tagged release and diff
it against the last tag in CI (buf breaking).
Additive: rebuild. Breaking: client code
changes.

THE DEMO'S STARTUP HANDSHAKE ONE UNARY RPC, CALLED BEFORE ANY COMMAND

```
rpc GetDeviceInfo ( DeviceInfoRequest ) returns ( DeviceInfo );  
message DeviceInfo { string firmware = 1; string interface_version = 2; }  
// GUI on connect: refuse or warn if interface_version != expected
```

Every rpc takes one message and returns one message

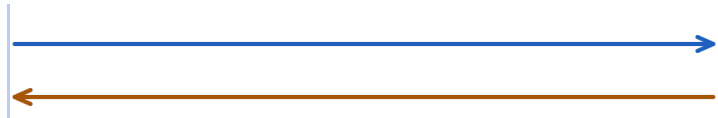
```
message SetPowerRequest {  
    int32 megawatts = 1;  
}  
message SetPowerResponse {  
    bool accepted = 1;  
}  
rpc SetPower (SetPowerRequest)  
    returns (SetPowerResponse);
```

Detailed guide on protocol buffer language:
<https://protobuf.dev/programming-guides/proto3/>

The four rpc types

Unary

Client



Server

Use it when: commands and queries. This is the default.

Server streaming

Client



Server

Use it when: telemetry, progress, tailing a log.

Client streaming

Client



Server

Use it when: uploads, batched samples pushed by the caller.

Bidirectional streaming

Client



Server

Use it when: long-lived sessions where both sides talk.

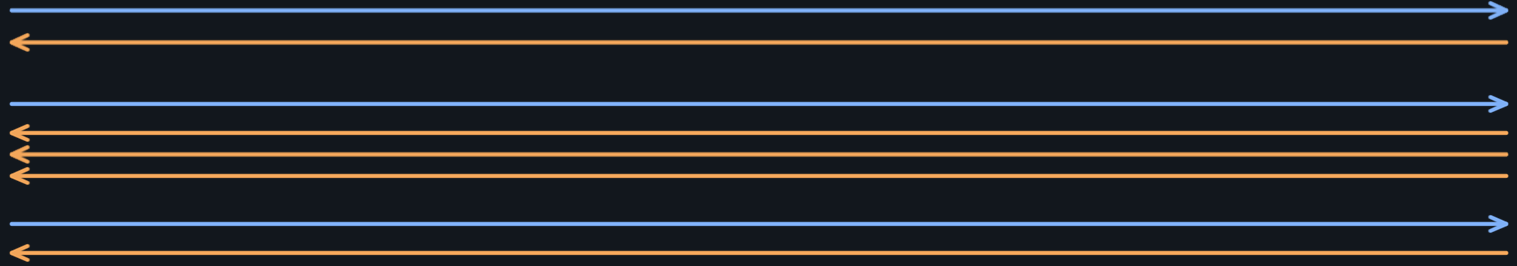
HTTP/2 underneath

ONE TCP CONNECTION

stream 1 · StartPlant

stream 3 · SubscribeTelemetry

stream 5 · StopPlant



Multiplexed streams

Many concurrent calls share one connection. A slow stream does not block the others. Each stream is chopped up into “frames” and each frame has a header. Frames can be interleaved on the connection!

Bits	Field
24	payload length
8	frame type (HEADERS, DATA, RST_STREAM, ...)
8	flags (e.g. END_STREAM)
1 + 31	reserved bit + stream identifier

Binary framing

Each message is a length-prefixed frame, so message boundaries are the transport's job, not yours.

Trailers

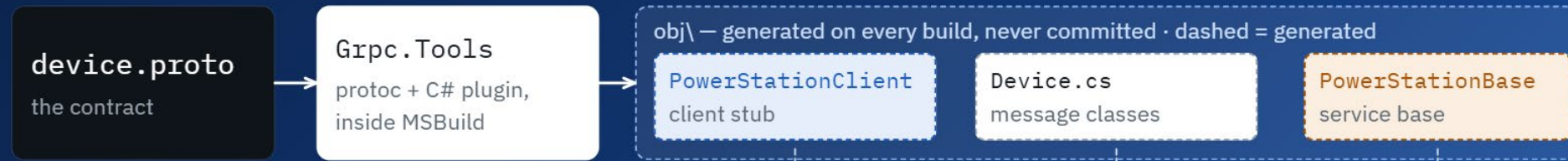
The final status arrives after the body, which is what lets a stream report failure cleanly at the end.

gRPC Execution Path – From Button Click to Endpoint

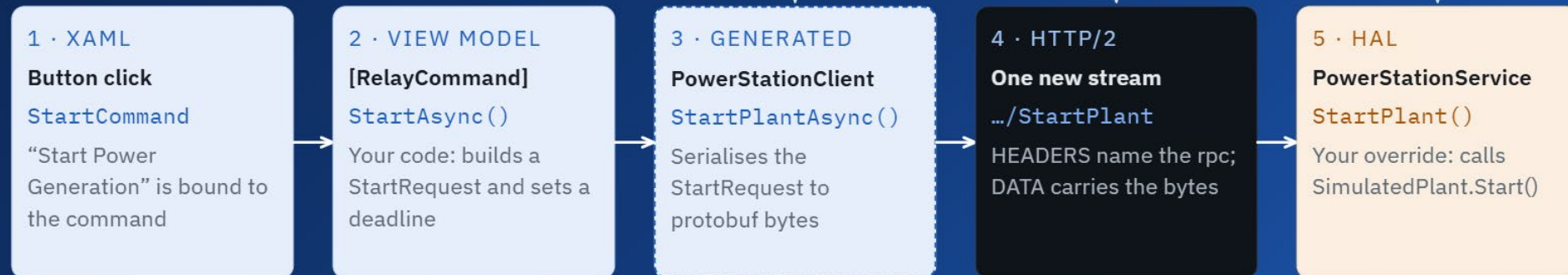
From device.proto to StartPlant()

What the build generates from one file, and the path a single click takes through it.

BUILD TIME — EVERY COMPILE



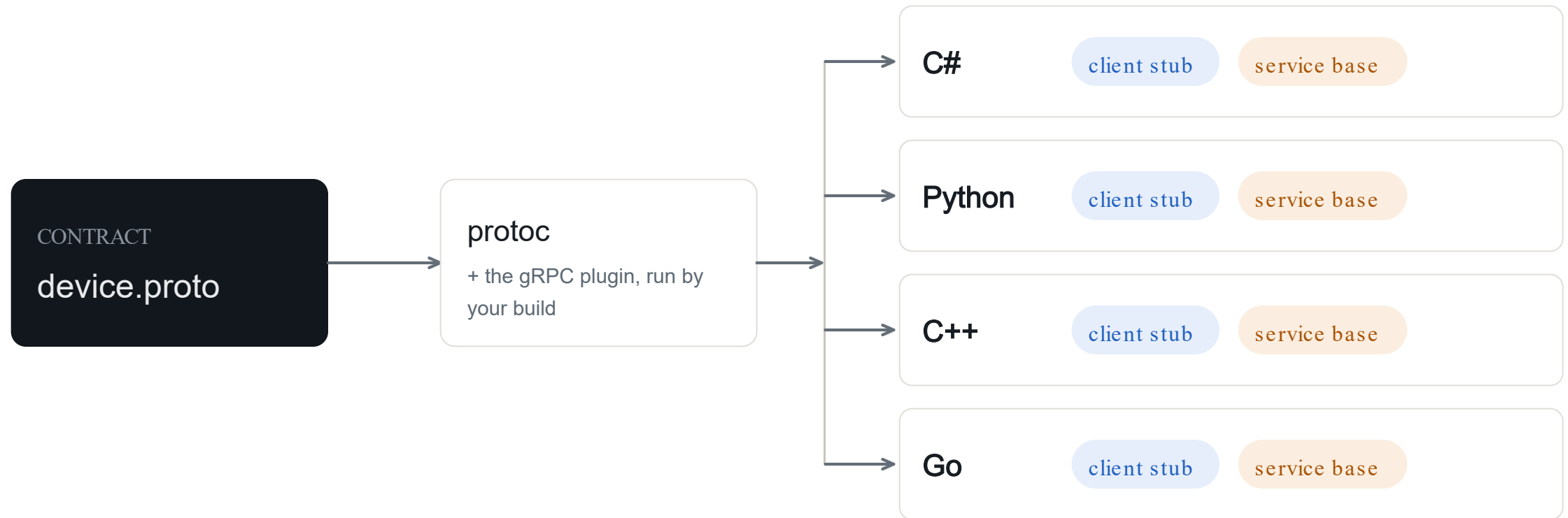
RUN TIME — PRESSING START



StartAck comes back on the same stream, then the trailer `grpc-status: 0` → the await completes and the pill shows STARTING

The code generation pipeline

One definition compiles into every language you need — both sides of it.



A stub you call on one side; an abstract base class you inherit from on the other. You never write the wire code.

The 3 Generated class files.

Three generated pieces, three jobs

What protoc writes into obj\ from device.proto — and the one thing you do with each.

DeviceGrpc.cs • client

PowerStationClient

The caller: create it, call it

- One method per rpc — StartPlantAsync() and the rest
- Returns awaitable calls, e.g. AsyncUnaryCall<StartAck>
- Holds no connection: wraps your channel, cheap and thread-safe

```
// you write
var client = new
    PowerStationClient(channel);
```

Device.cs • both sides

Message classes

The data: fill it, send it

- One class per message: StartRequest, StartAck, TelemetryFrame...
- Fields become properties: operator_id → OperatorId
- Writes and reads itself as protobuf bytes

```
// you write
var req = new StartRequest
    { OperatorId = "harry" };
```

DeviceGrpc.cs • HAL

PowerStationBase

The answerer: inherit, override

- One virtual method per rpc, same types as the client
- Not overridden? It answers UNIMPLEMENTED
- The server routes each :path to your override

```
// you write
class PowerStationService
    : PowerStationBase
```

How the two sides agree: DeviceGrpc.cs also holds one method descriptor per rpc — its path, /powerstation.v1.PowerStation/StartPlant, and the Device.cs types it carries. The client and the base are both built on it.

Channels, stubs, and calls

The channel is the library's; the stub is generated from your .proto; a call is a method on it.

C# —GRPC.NET.CLIENT

```
// CHANNEL : the gRPC library
var ch = GrpcChannel
    .ForAddress(addr);

// STUB : generated from .proto
var client =
    new PowerStationClient    (ch);

// CALL : one method per rpc
var ack = await client
    . StartPlantAsync    (req);
```

PYTHON—GRPCIO

```
# CHANNEL: the gRPC library
ch = grpc.insecure_channel(
    addr)

# STUB: generated from .proto
stub = device_pb2_grpc \
    . PowerStationStub    (ch)

# CALL : one method per rpc
ack = stub.    StartPlant    (req)
```

The other half: implementing the service

You *instantiate* the stub; you *inherit* the service base and override one method per rpc. The signature mirrors the client's, plus one server-only parameter —the context —carrying the deadline, cancellation token, and caller metadata.

server side

C# HAL—ASP.NET CORE

```
// 1. inherit the GENERATED base
class PowerStationService
    : PowerStation. PowerStationBase
{
    // the simulated hardware, injected by DI
    private readonly PlantSim _plant;
    public PowerStationService (PlantSim plant)
        => _plant = plant;

    // 2. override one method per rpc
    public override Task<StartAck> StartPlant (
        StartRequest req, ServerCallContext ctx)
    {
        _plant.Start();
        return Task.FromResult( new StartAck());
    }
}
```

PYTHON HAL—GRPCIO

```
# 1. inherit the GENERATED base
class PowerStationService (
    device_pb2_grpc. PowerStationServicer
):
    # the simulated hardware, passed in
    def __init__ ( self , plant):
        self ._plant = plant

    # 2. override one method per rpc
    def StartPlant ( self , request,
        context):
        self ._plant.start()
        return device_pb2.StartAck()
```

What you get for free

SETTING A DEADLINE

```
// C#  
var ack = await client.StartPlantAsync(  
    req,  
    deadline: DateTime.UtcNow  
        .AddSeconds( 5));
```

```
# Python  
ack = stub.StartPlant(req, timeout=  
    5)
```

A deadline is an absolute point in time that travels with the call, not a local timer.

Deadlines

A time budget on every call

Cancellation

Client cancels, server's token fires

Status codes

A standard set, not a string in a body

Metadata

Key/value headers and trailers per call

TLS

Configured on the channel, once

Status codes and error handling

Status code	What it tells the caller
INVALID_ARGUMENT	The request was malformed. Retrying it unchanged will not help.
FAILED_PRECONDITION	The system is not in a state where this operation is valid.
DEADLINE_EXCEEDED	The caller's time budget ran out before a reply arrived.
UNAVAILABLE	The server is unreachable or the connection dropped. Safe to retry.
CANCELLED	The caller ended the call.

```
catch ( RpcException ex) when
    (ex.StatusCode == StatusCode.Unavailable)
{ ShowBanner( "HAL is not reachable" ); }
```

ON THE CLIENT THIS SURFACES AS AN EXCEPTION
RPCException IN C#, RPCError IN PYTHON
LIST OF CODES [HTTPS://GRPC.IO/DOCS/GUIDES/STATUS -CODES/](https://grpc.io/docs/guides/status-codes/)

Constraints worth knowing before you commit

Browsers cannot speak native gRPC

JavaScript in a browser has no access to HTTP/2 frames or trailers.

A browser client needs gRPC-Web, which requires a translating layer in front of the service.

gRPCWeb supports unary and server-streaming calls only.

Target .NET 10 (LTS) —in support until 14 Nov 2028.

.NET Framework is a dead end for gRPC

Client only. There is no gRPC server for .NET Framework at all—Kestrel's HTTP/2 stack is modern .NET.

The client needs WinHttpHandler, TLS only, and Windows 11 or Windows Server 2019 or later.

No client or bidirectional streaming on Windows Server 2019 or 2022.

The older Grpc.Core package that did run there is in maintenance mode and being deprecated.

.NET 8 (LTS) and .NET 9 (STS) both leave support on 10 Nov 2026.

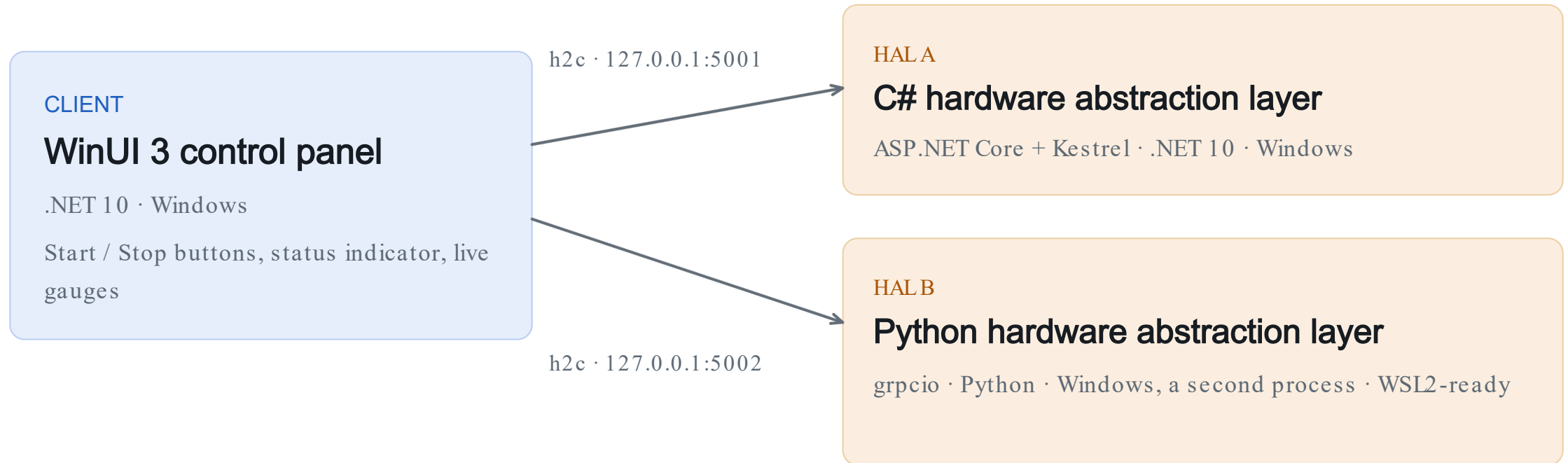
.NET Framework is the Windows-only 4.x runtime that stopped at 4.8 and gets security fixes only; modern .NET (5 through 10) is the cross-platform successor and is where all development happens.

The Demo Application



A power station control panel: one contract, two interchangeable hardware abstraction layers, three processes, one machine.

Demo architecture



device.proto

C# client stub

C# service base

Python service base

one file, three generated artefacts

The contract: device.proto

```
service PowerStation {
  rpc GetDeviceInfo (DeviceInfoRequest) returns (DeviceInfo);
  rpc StartPlant (StartRequest) returns (StartAck);
  rpc StopPlant (StopRequest) returns (StopAck);
  rpc SubscribeTelemetry (SubscribeRequest)
    returns (stream TelemetryFrame); // the only difference
}

message DeviceInfo { string firmware = 1; string interface_version = 2; }
message StartRequest { string operator_id = 1; }
message StartAck { bool accepted = 1; string state = 2; }
message TelemetryFrame {
  uint32 sequence = 1; float pressure_kpa = 2;
  float temperature_c = 3; string state = 4;
}
```

What gets generated, in three languages

C# —CLIENT AND HAL

Grpc.Tools

Runs inside the MSBuild build. Both C# projects regenerate on every compile, and no generated code is checked in.

PYTHON—HAL

python -m grpc_tools.protoc

Produces device_pb2.py for the messages and device_pb2_grpc.py for the service.

THE TYPE NAMES EACH SIDE ACTUALLY USES

Side	Generated type you write against
C# client	Power Station. Power StationClient —call its methods
C# HAL	Power Station. Power StationBase —inherit and override
Python HAL	device_pb2_grpc. Power StationServicer —subclass and implement

Build Action	Protobuf compiler
Class Access	Public
Compile Protobuf	Yes
File Name	device.proto
Full Path	C:\development\gRPCWebir
gRPC Stub Classes	Client only

Look at the service: `grpcurl`

No client code: the .proto alone is enough to call a HAL. Change the port; the other HAL answers. Good way to test endpoints before developing frontend.

```
# 1 · what does this HAL offer? No reflection needed —the file is enough
grpcurl -plaintext -import-path Contract -proto device.proto `
  127.0.0.1:5001 describe

# 2 · unary: press Start from a shell
grpcurl -plaintext -import-path Contract -proto device.proto -d '{"operatorId":"harry"}' 127.0.0.1:5001
powerstation.v1.PowerStation/StartPlant

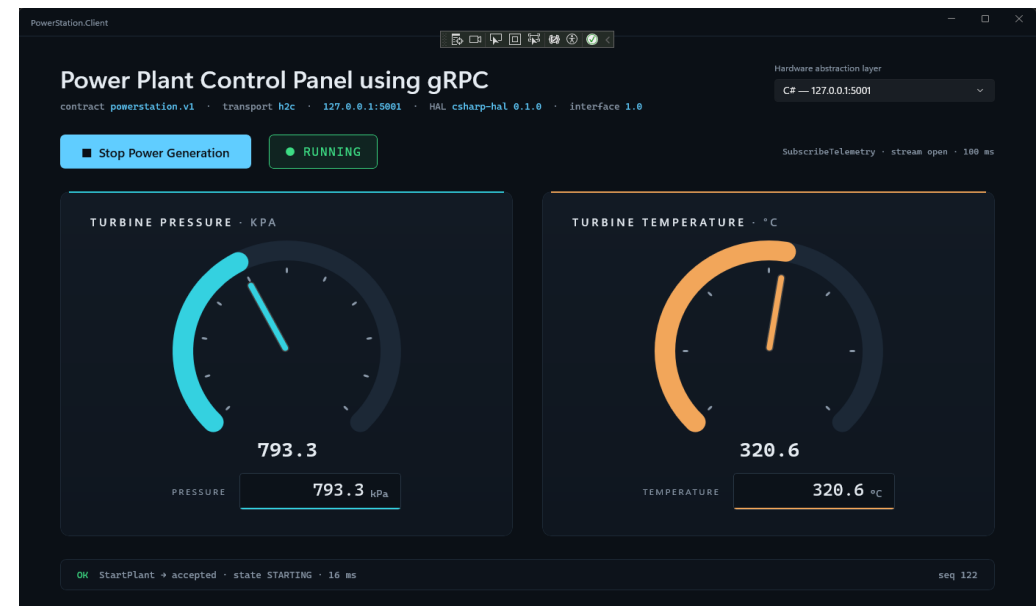
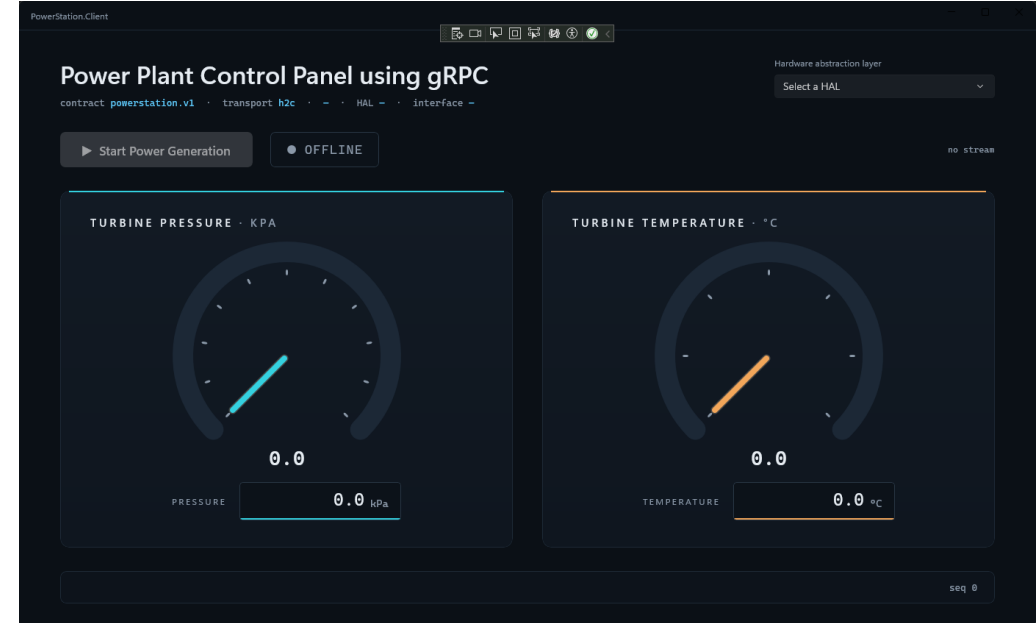
#3 unary: StopPlant
grpcurl -plaintext -import-path Contract -proto device.proto -d '{"operatorId":"harry"}' 127.0.0.1:5001
powerstation.v1.PowerStation/StopPlant

# 3 · server streaming: one JSON object per frame until Ctrl-C
grpcurl -plaintext -import-path Contract -proto device.proto -d '{"intervalMs":500}' 127.0.0.1:5001
powerstation.v1.PowerStation/SubscribeTelemetry
```

The unary call: pressing Start

- 1 Button click raises the start command on the view model
- 2 It calls StartPlantAsync on the generated client
- 3 StartRequest is serialised to protobuf bytes
- 4 A new HTTP/2 stream opens on the existing channel
- 5 The HAL's override of PowerStationBase.StartPlant is dispatched
- 6 The simulated plant transitions IDLE to STARTING to RUNNING
- 7 StartAck is serialised back on the same stream
- 8 The await returns and the status indicator updates

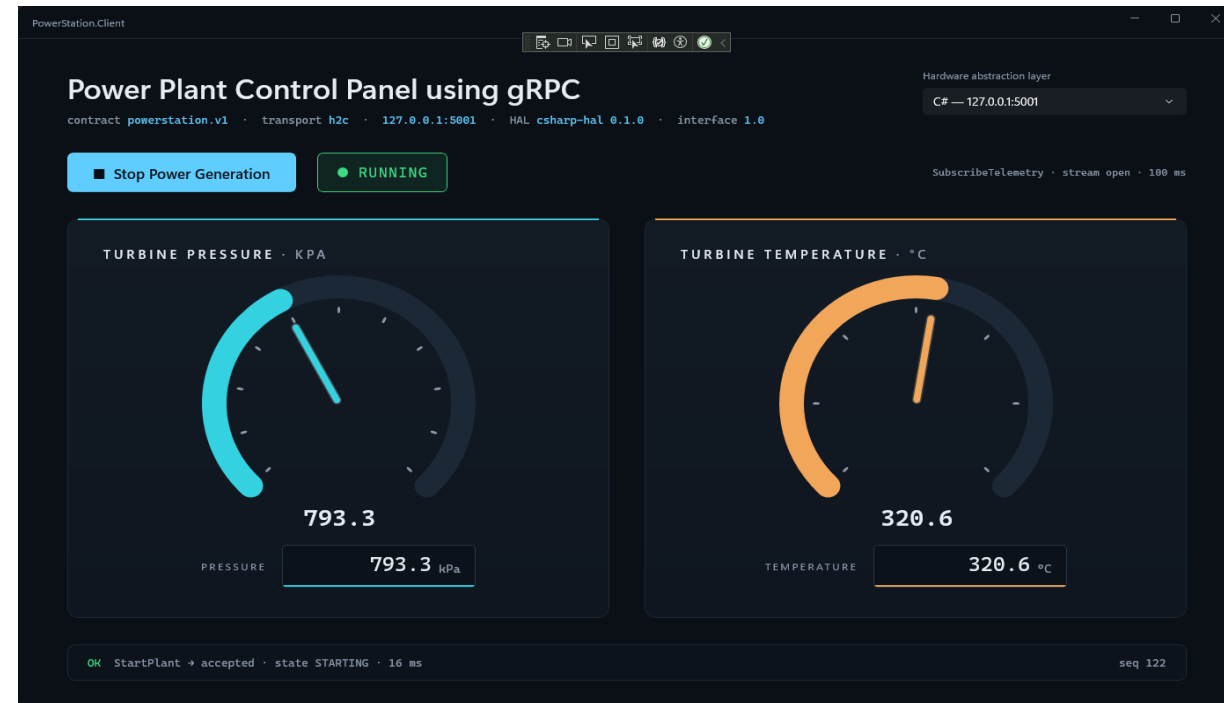
Eight steps, and the only two you write are the button handler and the override.



The streaming call: subscribing to telemetry

- 1 The client calls `SubscribeTelemetry` with a cancellation token
- 2 One HTTP/2 stream opens on the same channel—and stays open
- 3 The HAL's override receives a stream writer and loops
- 4 Each `TelemetryFrame` is one message, length-prefixed on that stream
- 5 The client awaits each frame as it arrives and updates the gauges
- 6 The loop runs until the client cancels or the deadline expires

The server method does not return a value. It returns when it is finished writing.



Same channel, same connection. The subscription is one long-lived call.

Look at the wire: Wireshark

The gRPC and Protobuf dissectors are built in. Point Wireshark at device.proto and it will decode the gRPC traffic.

HEADERS: path /PowerStation/SubscribeTelemetry

The rpc name is literally the URL path

DATA one length-prefixed SubscribeRequest

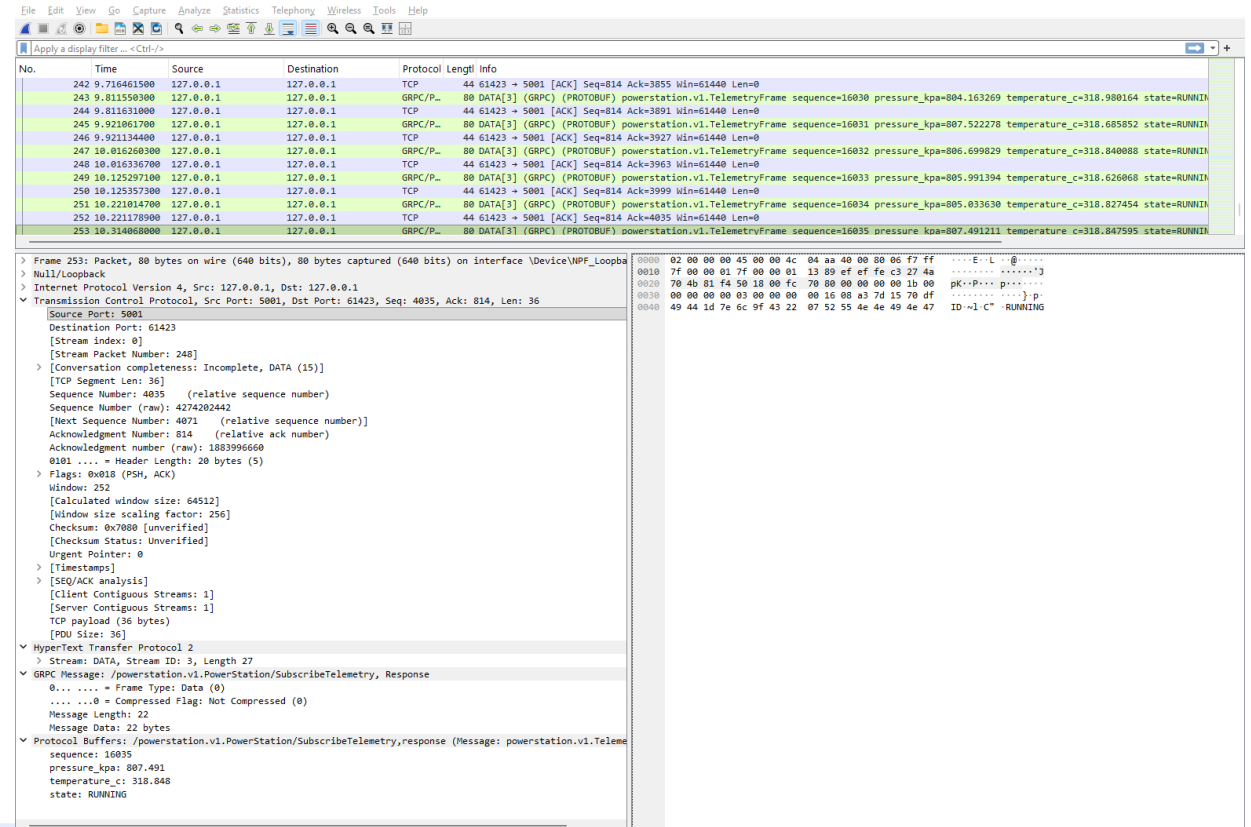
Decoded field by field, by name and type

DATA × n one TelemetryFrame per frame, *all on one stream ID*

Slide 10's multiplexing claim, on screen

HEADERS trailers: grpc-status: 0

Status travels in trailers, after the last message



1 · CAPTURE

Loopback via Npcap. Plaintext h2c means nothing to decrypt.

2 · DECODE AS

TCP 5001 and 5002 → HTTP2. Wireshark won't guess odd ports.

3 · PROTOBUF SEARCH PATHS

Add the folder holding device.proto. Field numbers become field names.

Server streaming is not publish/subscribe

A broker (MQTT, NATS, a bus)	A gRPC server-streaming call
Topics exist independently of any client	Nothing exists outside the call
A late subscriber can get retained or replayed messages	A late subscriber gets frames from the moment it calls
Fan-out is the broker's job	Fan-out is your server writing to N open streams
Delivery guarantees, QoS levels, persistence	Frames arrive while the call is alive. That is the guarantee
A separate process to deploy, secure and monitor	No extra infrastructure

It is one long-lived call whose response happens to arrive in pieces. When the call ends, the subscription ends.

Stream lifetime, cancellation, and the UI thread

```
using var cts = new CancellationTokenSource();
var call = client.SubscribeTelemetry(
    req, cancellationToken: cts.Token);

await foreach (var f in
    call.ResponseStream.ReadAllAsync())
{
    dispatcherQueue.TryEnqueue(
        () => UpdateGauges(f));
}
// Stop button: cts.Cancel();
```

Cancelling

Stop cancels the token. The server's cancellation token fires, its write loop exits, and the call ends with CANCELLED.

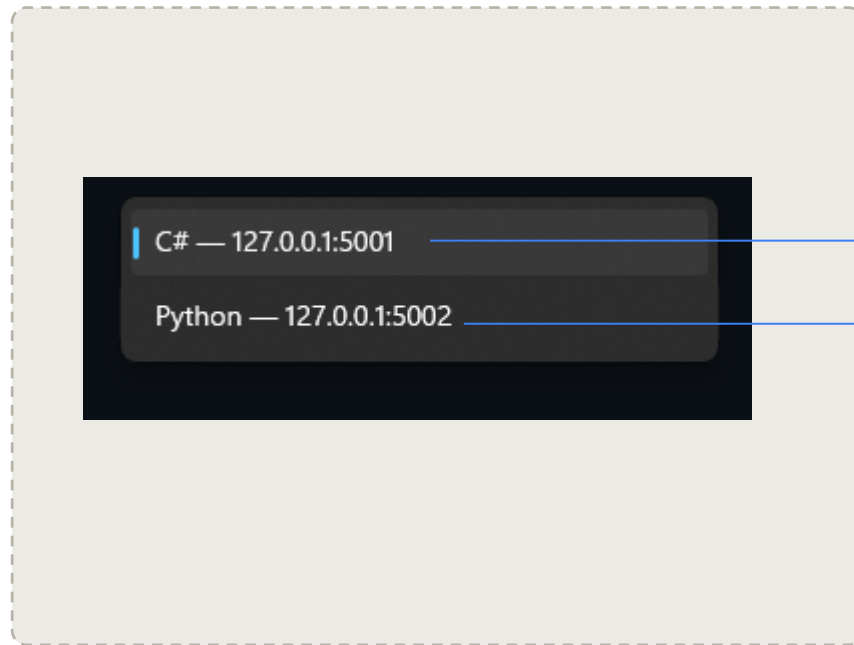
The UI thread

Frames arrive on a background thread. Touch the gauges only through `DispatcherQueue.TryEnqueue`.

Sampling rate

The HAL decides how fast to push. The UI must not be the throttle —drop frames rather than block the stream.

Language independence, demonstrated



The selector is the only control in the demo that is not part of the power station.

```
// one entry per backend process
Hals = [
    new( "C#" ,      "http://127.0.0.1:5001"      ),
    new( "Python" ,  "http://127.0.0.1:5002"      ),
];
// selecting one swaps the channel
_channel = GrpcChannel
        .ForAddress(Selected.Address);
```

Two processes, one machine, two ports. Selecting a HAL swaps the channel's address; nothing else changes.

In WSL2 the string is identical —loopback forwarding hides the OS boundary.

Failure modes, live

What we do on stage	Status code
Kill the HAL process mid-stream	UNAVAILABLE
Start the client before the HAL	UNAVAILABLE
Let a short deadline expire on Start	DEADLINE_EXCEEDED
Press Stop while telemetry is flowing	CANCELLED
Send Start while the plant is stopping	FAILED_PRECONDITION

Every one of these is a code the client can branch on, not a timeout you invented and a log line somebody has to read.

The last row is ours: `FAILED_PRECONDITION` is thrown by the HAL's state machine, not by the framework.

```
FAILED_PRECONDITION StopPlant → Cannot stop while plant is STARTING
```

```
seq 872
```

What this demo deliberately leaves out

Client and bidirectional streaming

When the caller also has a continuous source, or both sides need to talk on one call.

Interceptors

Cross-cutting logging, metrics, retries and auth, on every call without touching the methods.

Authentication and call credentials

mTLS on the channel, plus per-call tokens carried in metadata.

Load balancing and name resolution

When one name has to resolve to more than one backend.

Server reflection

Lets tools such as grpcurl discover the service at runtime without the .proto.

Health checking

A standard probe that orchestrators and supervisors already understand.

None of them changes the contract.

When to use gRPC —and when not to

Reach for it at the seams inside your system, not at the front door.

Good fit

- Service-to-service calls inside a system you own
- Device and instrument control
- High-frequency telemetry back to an operator
- Polyglot teams that must not hand -sync types
- Contracts that have to stay in step across releases

Poor fit

- Public, browser-facing APIs
- Anything that wants human -readable payloads on the wire
- Teams with no appetite for a build -time codegen step
- A single endpoint that is the whole integration surface
- Cases where a broker's durability is the actual requirement

WHERE TO GO NEXT

grpc.io/docs

Quick starts and guides for every supported language, including the C# and Python ones used in the demo.

protobuf.dev — proto3 language guide

The reference for field numbers, reserved, default values and the rules for evolving a message.

[gRPCPowerPlantDemo.zip](#)

device.proto, the WinUI 3 client, and both hardware abstraction layers.



Any questions?