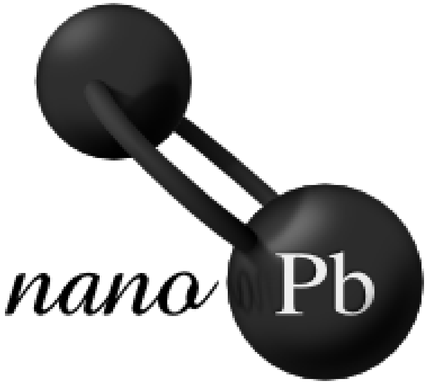


Google Protocol Buffers for Embedded IoT

Integration in a medical device project

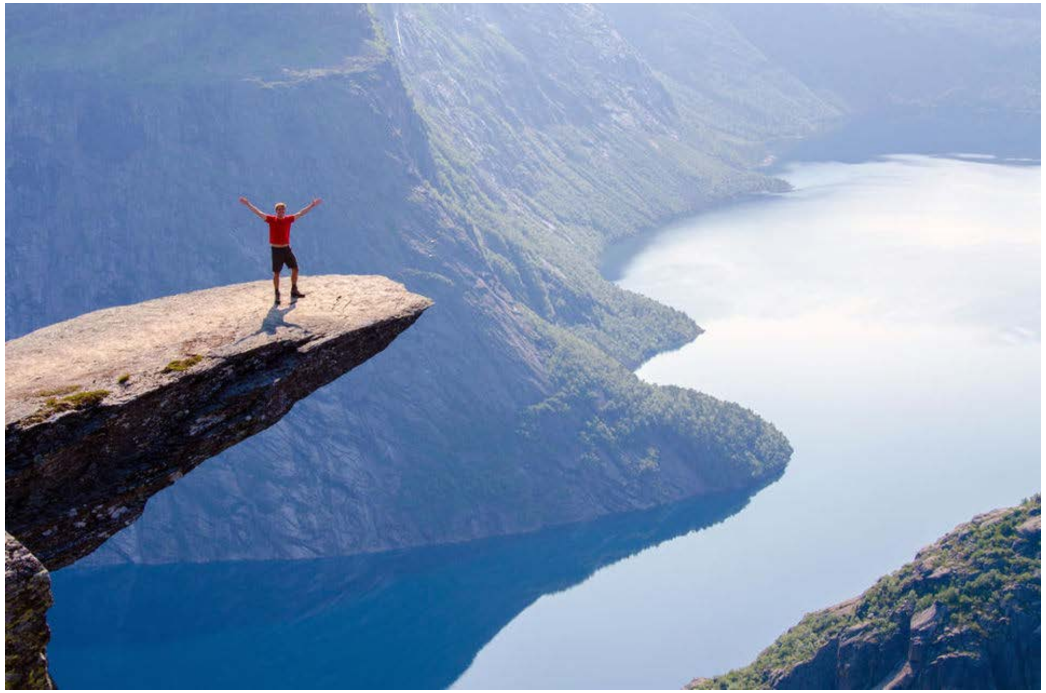
Quick look

1: Overview

- What is it and why is useful
- Peers and alternatives
- Wire format and language syntax
- Libraries for Embedded

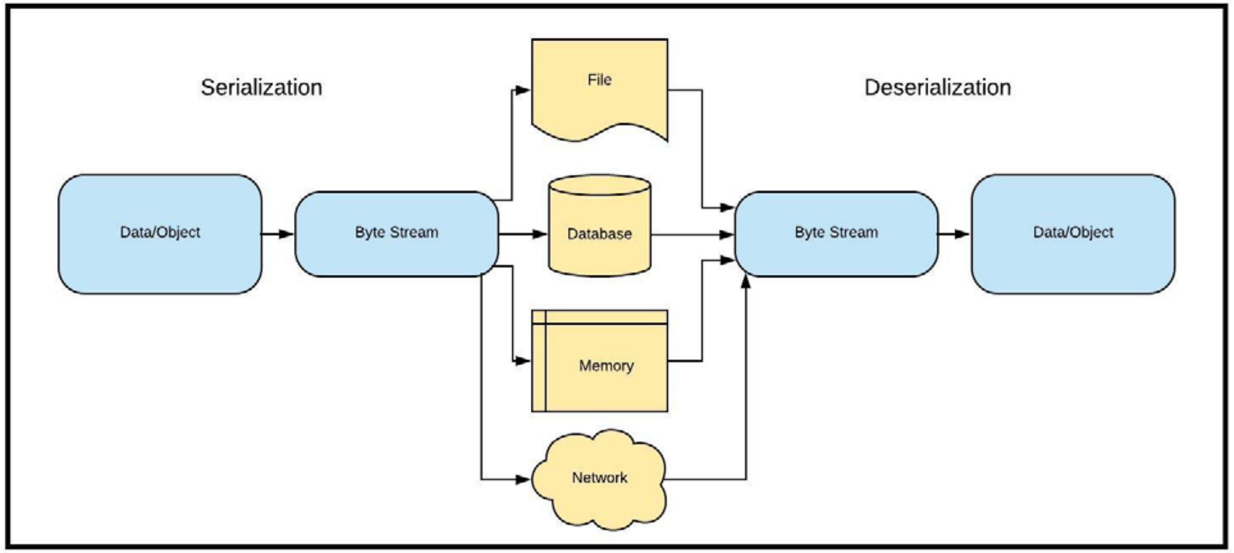
2: Project integration

- Why did we choose it
- How it was used and integrated
- Custom communications stack
- Lessons learned



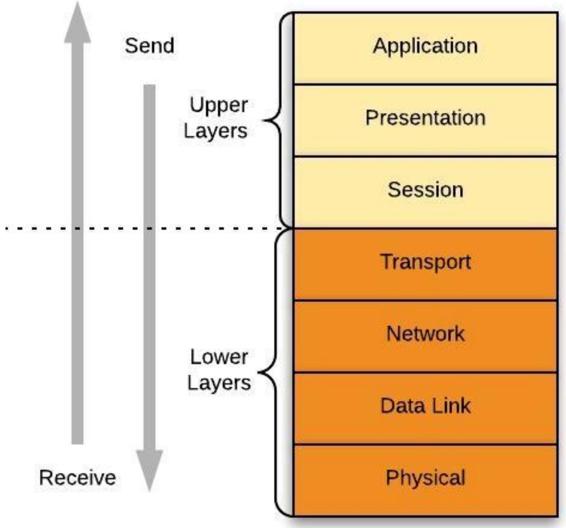
The what?

- Portable data interchange format for serialization across machine boundaries
- Used in producer/ consumer scenarios like:
  - Data blobs storage
  - Networks
  - PC to embedded devices
  - Multi-processor/ controller
- Specified wire exchange format
- Implementation not mandated
- Official implementation available
- Under BSD



The why?

- Language and platform portable
- Full scalar data type coverage
- Wire size efficient (but not optimal)
- Fast runtime performance (but not optimal)
- Basis for a “Remote Procedure Protocol”
- Backbone of an OSI stack (Layers 2-5)
- Excellent documentation
- No magic tricks, straightforward spec

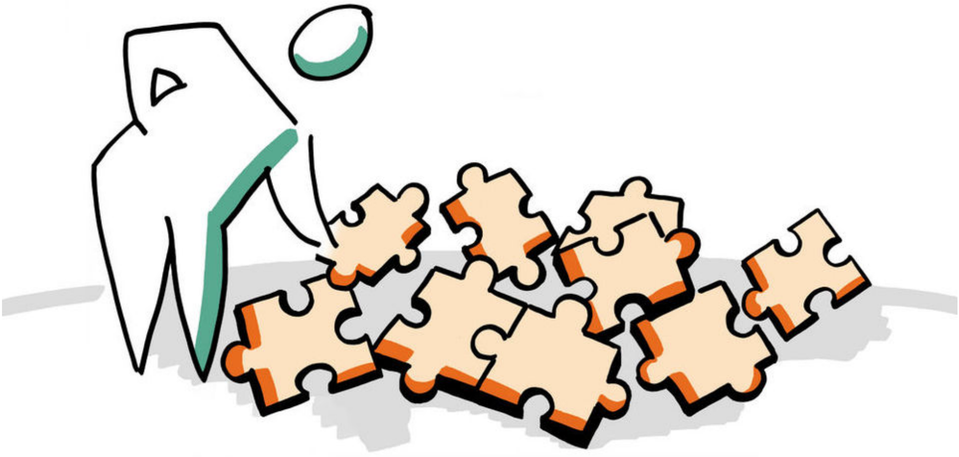


What is it not?

- No encryption/ decryption features
- No compression beyond encoding
- No RPC framework built in

- … but these things can be added

- It’s not self describing, the contract must be available!



What could it replace?

Serialization has been around the block…

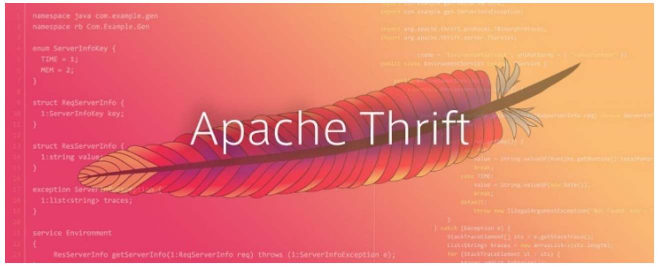
- Tried and “true” methods include:
- Soap, Corba, COM/ DCOM
  - ✓ Flexible and comprehensive
  - x Heavy weight integration/ use
  - x Not necessarily language/ platform neutral
- JSON, XML, Raw Text
  - ✓ Highly portable
  - ✓ Human readable (kind of)
  - x Not cheap to parse
  - x Not cheap to encode
  - x Not cheap to store
  - x Not cheap to send

$

What are it’s peers?

Apache Thrift

- From an ex-googler (2007)… similarities
- was internal at Facebook, now open source
- Similar to PB in performance
- Similar feature set to PB and even more languages
- Full built in RPC layer
- Less documentation
- Still slightly less efficient



The latest craze

Apache Avro

- JSON Schema always available
- Payload can be binary or JSON
- Schema robust to changes (alias’s, defaults)
- Can read/ write PB and Thrift!
- Similar speed/ space to PB
- Comparatively limited language support
- Only recently stable (2016)

But wait… there's more!

We need to go faster…

- "Protocol Buffers" spawned streamlined "zero copy" serialization formats.

- Why do we even need to encode/ decode?! Why can't we **mmap**the data?!

  - April 2013:Cap'nProto from author of "Protocol Buffers" v2
  - December 2013: SBE (Simple Binary Encoder) for financial tradin
  - June 2014: Flat Buffers from Google for game development

- By nature faster than Google's PB implementation, but beware the gotchas…

You could…

Roll your own

- ✓ Specify only what you need
- ✓ Can be faster
- ✓ Can be smaller
- x But …. lions, tigers, and bears… oh my!

Short history

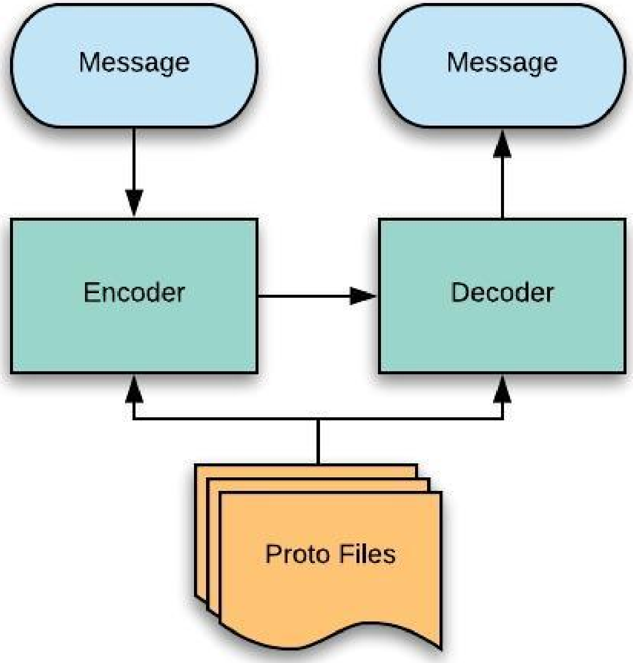
It’s been awhile!

- Developed internally at Google circa 2001
- Used as core glue for google services
- Released open source (2008) to public as stable and well tested
- Google’s implementation’s security has been verified
- Reward for finding XSS exploit in Google maps w/ Protobuf

IDL

Overview and integration

- Interface definition language specified by Google
- Two versions Proto2/ Proto3
- Files have .proto suffix
- Google compiler converts IDL to boilerplate
- Runtime libraries decode and encode streams
- Simple to read and write



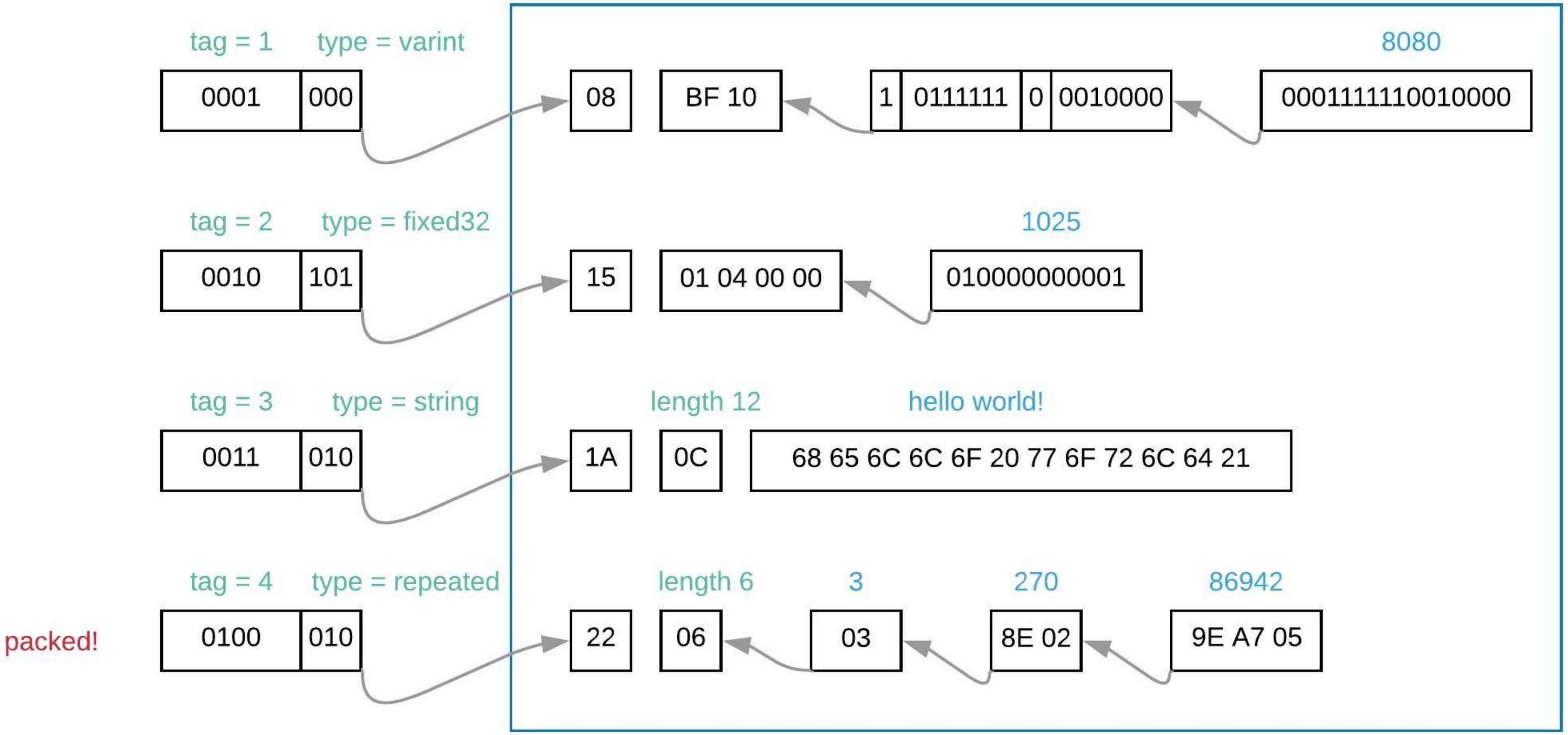
IDL syntax

Proto2 example

```
//Generic definitions in other proto
import “base.proto”

enum OperationType
{
        IDLE = 0;
        SLEEP = 1;
        DOWORK = 2;
}

message OperationRequest
{
        required OperationType requestType = 1;
        optional uint32 timeout = 2;
        optional bool sleepOnFinish = 3;
}
```

```
message Result
{
        required string resultString=1;
        optional uint32 timestamp=2;
}

message OperationResponse
{
        repeated Result result=1;
}
```

Wire format



tag = 1 type = varint

0001 | 000

08 | BF 10 | 1 | 0111111 | 0 | 0010000

8080

0001111110010000

tag = 2 type = fixed32

0010 | 101

15 | 01 04 00 00

1025

010000000001

tag = 3 type = string

0011 | 010

1A | 0C | 68 65 6C 6C 6F 20 77 6F 72 6C 64 21

length 12

hello world!

tag = 4 type = repeated

packed!

0100 | 010

22 | 06 | 03 | 8E 02 | 9E A7 05

length 6

3

270

86942

Above covers signed numbers (two possible encodings) and floats (@fixed32).
Separate type for 64-bit fixed sized numbers

But we didn’t use Google’s tools

A bit on the heavy side

Google C++ implementation under BSD

- But it depends on STL…
- Dynamic allocation
- … code space starts around 100KB or more

You could strip it down,

… but Embedded was not the target market!

PB for embedded

Alternatives

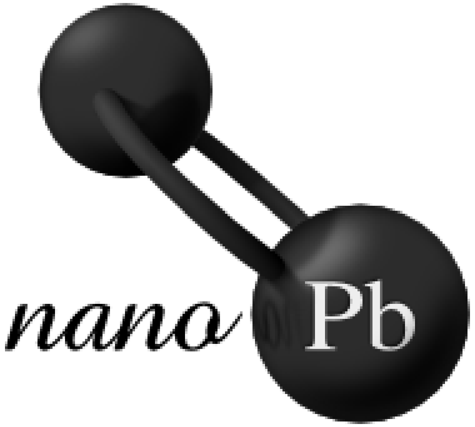
- Various C libraries from third parties over the years:
  - … lwpb, Protobuf-embedded-c, empb

- The actives ones:
  - Protobuf-c
    - New BSD license
    - Feature complete to google specification
    - Dynamic allocations, but customizable with allocators
    - > 20 KB rom

The winner is…

Nanopb

- Z-lib license
- Fully static by default
- Smallest @: ~ 2-10 rom KB and ~ 300 bytes ram
- Favors small size over serialization speed
- Strings/ Byte Arrays use custom field options to specify size
- Precompile flags to tailor fit (ex. Field-tag size)
- For dynamic sized fields there are callbacks
- IDL processing: Google PB compiler + python plugin
- Complete documentation

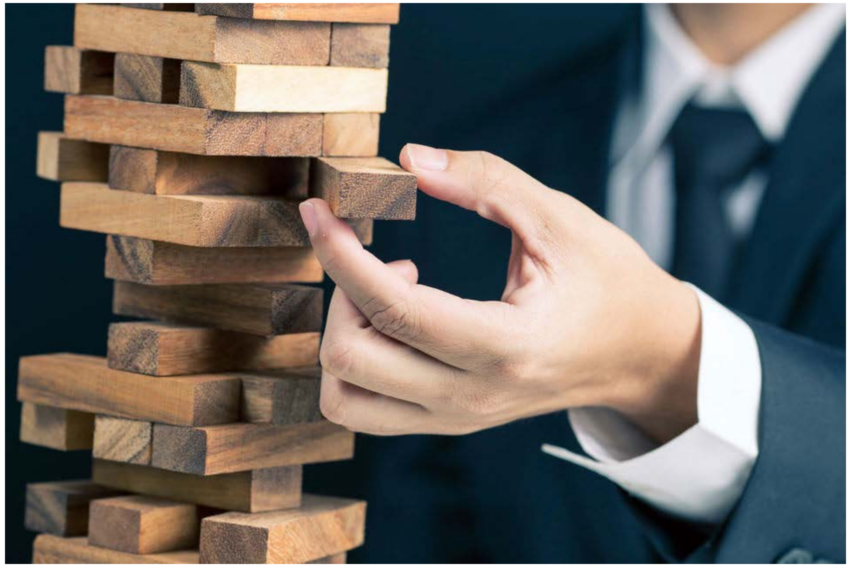


Unit tests included!

May sound crazy…

You still have a spec!

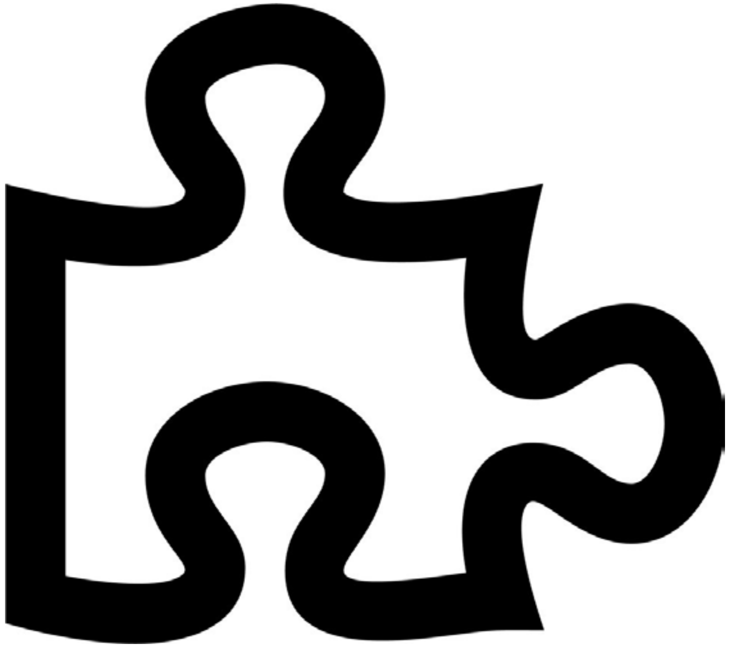
- Optimize for your use case
- Example: [Custom PB Serializer]
- But still…



Thanks for the overview but….

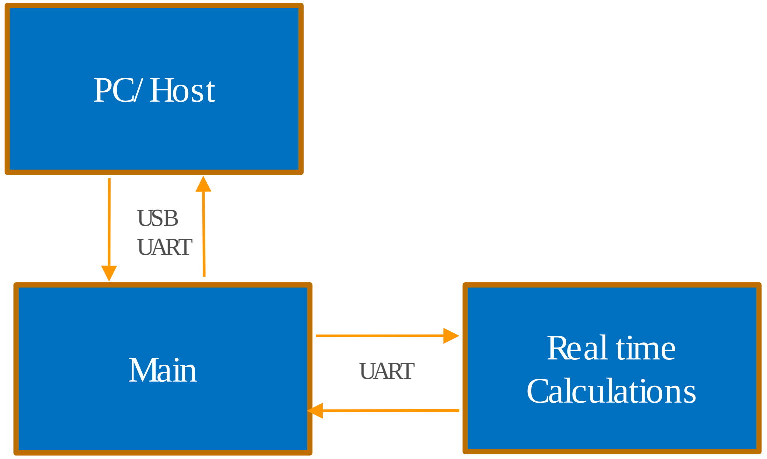
Why NanoPB was our choice?

- **No** custom serialization spec, no XML/JSON
- Portability: C#, C/C++, and Java
- Size: Must run on mid-range microcontrollers
- License: Sell it on the customer



Scenario and Constraints

- Remote procedures:
  - Device operation through Host App(s) in C# and Java
  - Services on Master controller <-> Services on hard real time controller
- Main controller:
  - ❑ Plenty of ROM (> 1MB)
  - ❑ RAM 256KB
  - ❑ Middleware Embedded OS
- Real time controller:
  - ❑ 256 KB RAM + ROM
  - ❑ Tight real time requirements
  - ❑ No OS



Diesel and Gaudi provide the glue

- No RPC in Protocol Buffers but easy to add!
- Diesel DSL approach with ruby, rake, anGaudi
- Our DSL target:
  - System - Shared definitions (types, enums, defines)
  - Services – remote API for on device functional modules

Example

```
%% Provides an interface to control LEDs on the device.
service LED=15 {
  namespace = LED
  version = 2.0.0
  platform = main_controller
```

```
%% Defines the types of LEDs.
 enum LEDType {
  prefix = eLDT

  mapping {
   Red = 0
   Green = 1
  }
 }
```

```
%% Turns on and off LEDs on the controller.
 command SetSteadyState=1 {
  encodable_from = host

  fields {
   %% The type of LED to update
   LEDType led_id = 1
   %%  Desired status of the LED
   bool enabled = 2
  }
 }
```

Generate whatever you can!

NanoPB boilerplate:

- DSL → Proto IDL → NanoPB structs+

Diesel

PB compiler

C++ boilerplate classes per service:

- Decoders: byte streams → NanoPB C-structs → service API calls
- Encoders: API Calls → NanoPB C-structs → PB streams

Packet structure

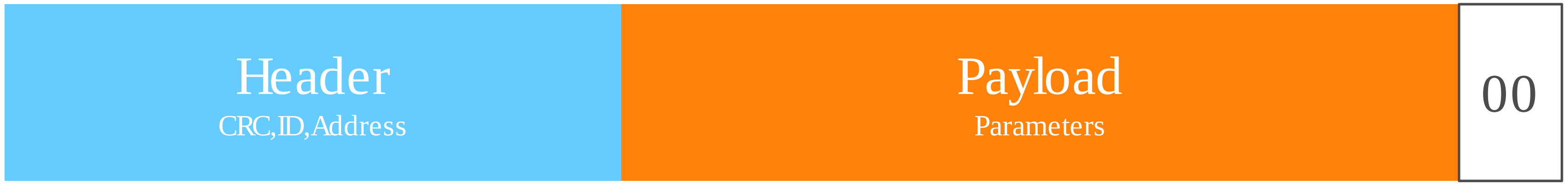
Custom meta format

- Two PB based packets back to back for each RPC.

1) Shared Header: Manually serialized using library.

2)Payload: Full PB message serialized using generated code.

No separator!



Header

CRC,ID,Address

Payload

Parameters

00

Packet streams

- Zero byte terminators
- Bytes reencoded with Consistent Overhead Byte Stuffing
- No “reserved” zero bytes in payload

1	11 22 00 33	03 **11 22** 02 **33** 00
2	11 22 33 44	05 **11 22 33 44** 00
3	11 00 00 00	02 **11** 01 01 01 00

✓ Packets can be defined of unlimited size

✓ Overhead minimum of 1 byte and maximum of n/ 254

x Errors require full resend

Packet integrity

Did we get the right message

1. First field is CRC
2. CRC uses Fixed32 type
3. Initialize CRC to zero
4. Encode header and payload
5. Calculate CRC over encoded stream
6. Encode CRC value in zero’d bytes
7. Reverse the steps to verify

Stream integrity

A fork in the road

Custom ACK/NACK packet

- Just CRC header and sequence number payload
- Can be distinguished easily by field tags
- Lower bandwidth and faster performance

Lessons learned

- ✓ Reliability from mature predefined protocol saves a lot of time
- ✓ NanoPB implementation solid and efficient
- ✓ Custom light weight RPC more than enough
- ✓ Zlib license is low stress
- ✓ Docs from Google and Nanopb comprehensive
- ✓ Rarely needed to peek in the box
- ✓ … and when we did it wasn’t scary
- – More flexibility and features than needed
- – Efficient but not optimal (4 byte minimum fixed size)

Open source…

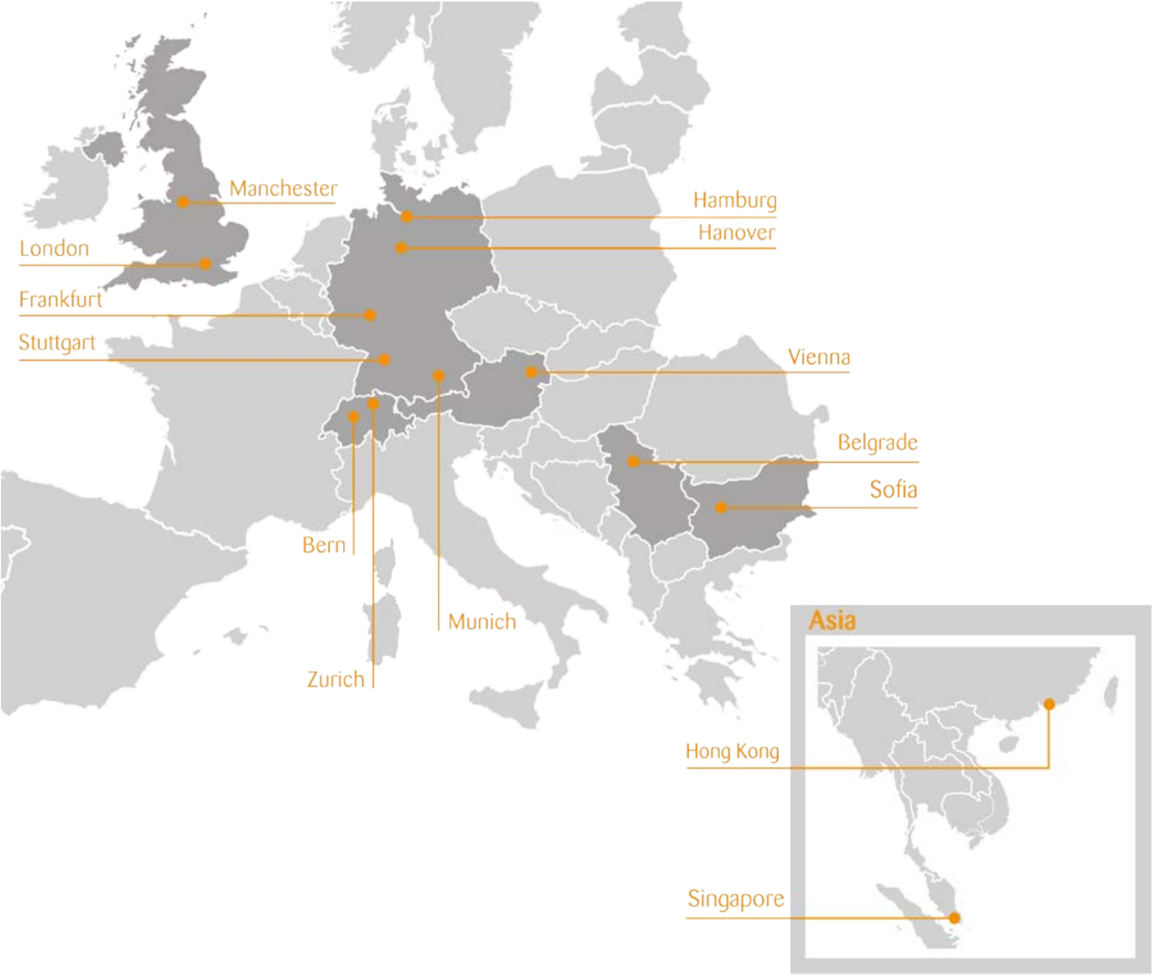
It's perfect right?

- Advantages are clear when the project is popular!
- But free is not always appropriate:
  - Consider the licensing model … Is what's free today free tomorrow?
  - Verification doesn't come for free, is the choice temporary or will it go the distance?
- And for the maintainers….
  - https://blog.marcgravell.com/2018/04/having-serious-conversation-about-open.html

About Zühlke

Facts and figures

We are hiring!



- Founded 1968
- Owned by partners
- Teams in Germany, United Kingdom, Austria, Bulgaria, Serbia, Singapore, Hong Kong and Switzerland
- Over 10,000 projects implemented
- 1,000 employees and a turnover of CHF 154 million (2017)
- Certifications: ISO 9001 and 13485

Morgan Kita
Expert Software Engineer
+49 174 302 9332
morgan.kita@zuehlke.com
https://de.linkedin.com/in/morgan-kita-6513a343
https://www.zuehlke.com

That’s all folks!