Nest Weave
TLV

White Paper

Revision 4
2020-03-04

Status: Approved / Active

Copyright © 2020 Google LLC
Google LLC Public Information

Revision History

Revision	Date	Modified By	Description
1	2014-11-24	Grant Erickson	Initial revision.
2	2015-02-25	Grant Erickson	Final draft.
3	2015-06-03	Grant Erickson	Updated quantitative analysis to include Flatbuffers. Reran all quantitative analysis using GCC 4.8.2 targeted to an ARM Cortex A9 with a hard floating point ABI.
4	2020-03-04	Grant Erickson	Updated for CBOR.

Copyright © 2020 Google LLC
Google LLC Public Information

Table of Contents

Copyright © 2020 Google LLC
Google LLC Public Information

Copyright © 2020 Google LLC
Google LLC Public Information

Summary

This document describes the motivation, rationale, and competitive analysis that went into the creation and definition of the Nest Weave: Tag Length Value (TLV) data representation format. In the competitive analysis, comparisons are made against alternative representations such as JSON, Google Protocol Buffers, ASN.1, EXI, Apache Thrift, and CBOR along a number of evaluation criteria.

Copyright © 2020 Google LLC
Google LLC Public Information

Introduction

This document describes the motivation, rationale, and competitive analysis that went into the creation and definition of the Nest Weave: Tag Length Value (TLV) [5] data representation format[1].

Motivation and Rationale

Target System Resources

At its outset, Nest Weave was designed with the intent to bring rich, highly-secure, Internet-class IPv6 connectivity and applications to deeply-embedded devices within the home. The capabilities of such devices are measured in storage resources such as dynamic RAM in the 64 KiB to 128 KiB range and non-volatile flash in the 128 KiB to 512 KiB range and compute resources of their 32-bit ARM Cortex M-class processors running at between 19 to 40 MHz.

In addition to these constrained computing devices, Weave further considered constrained communications media beyond the world of Ethernet and WiFi and looked at the world of IPv6-capable low-power networks such as 6LoWPAN, an IPv6 adaptation over 802.15.4, where packet sizes are limited to 127 bytes and link data rates to 250 Kbps, both mere slivers of the comparable metrics associated with Ethernet or WiFi.

As a consequence of these constrained computing devices and communication media, size was and continues to be a dominant driving factor in the design of Weave and its constituent components.

As the core components of Weave were being defined and as the definition for the basic protocol message format was considered, a conscious decision was made to split the design space into two:

- Core Message Format
- Application Data Representation

Core Message Format

The Weave Message Layer [4], while out of scope for this document, was made hand-tuned and highly-structured and -defined to ensure a consistent, compact representation and implementation that can support not only secure transport of arbitrary Weave message exchanges over either TCP or UDP but also application protocols within those exchanges that are either equally-structured and -defined or more flexibly-represented data along with a light to modest amount of structure and definition.

[1] Technically, this is actually Type Tag Length Value (TTLV); however, Type and Tag are packed into the same field.

Copyright © 2020 Google LLC
Google LLC Public Information

Application Data Representation

With the definition of the Weave Message Format defined, the design efforts focused on the definition of an application data representation that could be used anywhere in Weave where a tight, hand-tuned, explicit data format and representation would be either too restrictive or too non-interoperable.

Before striking out to implement something anew, Nest conducted a broad market survey, evaluating existing technologies against those criteria summarized in Table 1 and described briefly below.

Requirements

Requirement	Weight
Capable of Representing Basic Machine Types	Heavy
Capable of Representing Arrays	Heavy
Capable of Representing Structures	Heavy
Capable of Forward- and Backward Compatibility	Heavy
Capable of Representing Optional Content	Heavy
Partitioned and Controlled Tag Space	Heavy
System Neutrality	Heavy
Resource Overhead	Heavy
Trivial In-place Usage of Basic Machine Types	Heavy
Over-the-wire Compactness	Moderate
Lossless Translation to JSON	Heavy
Licensing and Seat Costs	Moderate
Market Penetration	Light
Proliferation and Quality of Infrastructure and Tools	Light

Table 1. Application data representation requirements and weighting.

Capable of Representing Basic Machine Types

This requirement gauges the native ability of the representation to symbolize basic machine types typically encountered in embedded system runtime environments such as C and C++. These types include:

- Null

Copyright © 2020 Google LLC
Google LLC Public Information

- Booleans
- 8-, 16-, 32-, and 64-bit Signed and Unsigned Integers
- 32- and 64-bit IEEE 754-1985 Floating Point Numbers
- UTF-8 Character Streams
- Byte Streams

These *plain-old-data (POD)* basic types are the building blocks upon which all other aggregate data types are constructed. Runtime environments can, of course, be built to operate on abstract data representations; however, ultimately the resources required to do so is just additional system resource overhead to effect and operate on this abstraction. Consequently, representation of and efficient operation with these types is crucial.

Capable of Representing Arrays

This requirement gauges the native ability of the representation to represent a homogeneous array of another primitive or aggregate type, such as an array of UTF-8 character strings or signed integers.

Capable of Representing Structures

This requirement gauges the native ability of the representation to represent a heterogeneous collection of other primitive or aggregate types, such as might be needed to represent personal contact information consisting of a:

- first name
- last name
- title
- postal address
- e-mail address
- phone number

Capable of Forward- and Backward-Compatibility

This requirement gauges the ability of the representation to add new content and deprecate existing content in the future without invalidating the entirety of the application-level representation or protocol on the part of either the initiator or responder.

The presence of this feature allows:

- Old initiators to omit new content and have new responders cope by providing defaults.
- New initiators to emit new content and have old responders cope by ignoring it.

Capable of Representing Optional Content

This requirement gauges the ability of the representation to indicate optional content which, whether present or absent, does not represent an error on the part of either the initiator or responder. This is similar but not identical to support for backward- and forward-compatibility.

Copyright © 2020 Google LLC
Google LLC Public Information

Partitioned and Controlled Tag Space

This requirement gauges whether the representation has a tag space for marking and tagging data and whether, if present, that tag space may be partitioned and controlled in its entirety by the Weave ecosystem and its participating partners.

This is an important requirement since Nest, with Weave, explicitly endeavors to not only tightly manage a portion of the space but also to assign vendors their own tag space and to allow vendors to extend that space as they see fit.

System Neutrality

This requirement gauges the ability of the representation to allow two systems with different processors, operating systems, and runtime environments to successfully exchange and interpret one another's data.

This is a hallmark of nearly all data presentations and is one of the chief motivating factors in their creation and existence.

Resource Overhead

This requirement gauges the ability of the core implementation as well as the encoders and decoders to be efficiently represented in terms of read-only machine code as well as read-only and read/write RAM space.

An ideal implementation would require little more than *memcpy* from the C Standard Library and processor-native memory load and store operations (e.g. store and load 8-, 16-, 32-, and 64-bit quantities).

Appendix A: Size Analysis contains a detailed resource assessment of several competitive representation technologies.

Trivial In-place Access of Basic Machine Types

This requirement gauges the ability of the representation to handle encoding and decoding of the aforementioned basic machine types through the use of *memcpy* from the C Standard Library or processor-native memory load and store operations, potentially along with byte swapping operations.

Representations that excel here largely facilitate in-place operation on over-the-wire data which impacts system memory requirements.

Copyright © 2020 Google LLC
Google LLC Public Information

Over-the-wire Compactness

This requirement gauges the efficiency of the representation in encoding a given set of native machine data into the least number of bytes.

Achieving low- or high-degrees of over-the-wire compactness comes with trade-offs against both the *Resource Overhead* and *Trivial In-place Access of Basic Machine Types* requirements by increasing the amount of encoding and decoding that must take place before the represented data can be either remotely-transmitted or locally-manipulated.

Lossless Translation to JSON

This requirement gauges the ability of the representation to trivially and mechanically map to and from a JSON-based representation without data loss and without application- and content-specific knowledge.

JSON is the effective lingua franca of the RESTful APIs of the modern Internet and is the core backbone of much of the cloud server infrastructure behind those APIs. As a consequence, a representation that can trivially and losslessly map to and from JSON presents an opportunity to achieve maximum leverage of that infrastructure while still meeting the constraints of the Weave ecosystem.

Licensing and Seat Costs

This requirement gauges the capital and operational expense of acquiring and supporting a particular implementation for a representation.

Market Penetration

This requirement gauges the scope of adoption of the representation in the overall marketplace. Broad market penetration will have a tendency to imply a broad number of implementations, tools, development resources, support resources, and other resources.

Proliferation and Quality of Infrastructure and Tools

This requirement is a subset of the *Market Penetration* requirement and specifically gauges the proliferation and quality of infrastructure and tools available to support the development, testing, integration, and support of the data representation. Broad infrastructure and tools availability potentially makes an easier entry point for ecosystem developers and lowers development and support costs for ecosystem device vendors.

Competitive Analysis

Using the requirements outlined above, we analyzed the following data representations for use in Weave:

Copyright © 2020 Google LLC
Google LLC Public Information

- JSON
- memcpy
- Google Protocol Buffers
- ASN.1
- EXI
- Thrift
- CBOR

before coming up the design and implementation that is Weave TLV.

Summary

Criteria	Representation							
	JSON	**memcpy**	**Google Protocol Buffers**	**ASN.1**	**EXI**	**Thrift**	**CBOR**	**Weave TLV**
Capable of Representing Basic Machine Types	Partial	Yes	Yes	Partial	Partial	Partial	Yes	Yes
Capable of Representing Arrays	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Capable of Representing Structures	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Capable for Forward- and Backward-Compatibility	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Capable of Representing Optional Content	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Partitioned and Controlled Tag	Maybe	No	Partial	Yes	No	Yes	Partial	Yes

Copyright © 2020 Google LLC
Google LLC Public Information

Space								
System Neutrality	High	Low	High	High	High	High	High	High
Resource Overhead	High	Low	Variable[2]	High	High	High	Moderate	Moderate
Trivial In-place Access of Basic Machine Types	No	Yes	Partial	No	No	No	Partial	Yes
Over-the-wire Compactness	Low	Low	Moderate	Variable[3]	Low	Variable[4]	Moderate	Moderate
Lossless Translation to JSON	Yes	No	Mostly	No	No	Mostly	Mostly	Mostly
Licensing and Seat Costs	Low	Low	Low	High	Low	Low	Low	Low
Market Penetration	High	High	Moderate	Moderate	Low	Moderate	Low	Low
Proliferation and Quality of Infrastructure and Tools	High	High	Moderate	High	Low	Moderate	Moderate	Low

Table 2. Summary of data representations.

JSON

JSON, while the lingua franca of the RESTful modern Internet, simply has an over-the-wire representation that is too large and system resource overhead that is too great given the required encoding and decoding to move from the representation to native data types.

Consequently, while it works great for backend server infrastructure and can work well with cellphone-class embedded systems, its advantages decay when pushed into the deeply embedded world of devices Weave endeavors to address.

[2] Resource overhead varies from moderate to high depending on whether the Google reference implementation *libprotobuf* or the alternate *nanopb* is used.
[3] Supports a variety of encoding types that range from low to high in over-the-wire compactness.
[4] Supports a variety of encoding types that range from low to high in over-the-wire compactness.

Copyright © 2020 Google LLC
Google LLC Public Information

JSON, therefore, fails on the following requirements:

- Resource Overhead
- Trivial In-place Access of Basic Machine Types
- Over-the-wire Compactness

memcpy

This representation is a proxy for all bespoke, hand-tuned data representations, including those that simply agree on a byte order and then effectively byte-swap and memcpy types into and out of network buffers, potentially with some shifting and masking.

This representation has compelling attributes across the board but fails the following requirements:

- Capable of Forward- and Backward-Compatibility
- Representation of Optional Content
- Partitioned and Controlled Tag Space
- Lossless Translation to JSON

That said, this stood and still stands as a benchmark representation.

Google Protocol Buffers

Google Protocol Buffers (protobufs) was originally developed as a Google-internal representation and remote procedure call (RPC) mechanism emphasizing simplicity and performance, attempting to best XML and one other representation considered here, ASN.1.

Through its efforts to open source the implementation, protobufs have received broader marketplace adoption outside of Google.

However, protobufs optimized for a different design center on a number of different fronts. Chief among them was in the area of low resource overhead.

The Google reference implementation includes two C/C++ codecs, *libprotobuf* and *libprotobuf-lite*. However, the recommended application space for even libprotobuf-lite, "*more appropriate for resource-constrained systems such as mobile phones*" [3], is dramatically out of scope relative to what Weave considers to be a resource-constrained device.

In addition, Google has carved out a chunk of the usable tag space for Protocol Buffers, making adopting into the Weave ecosystem difficult.

Ultimately, Google Protocol Buffers did not meet the following requirements:

- Partitioned and Controlled Tag Space
- Resource Overhead

Copyright © 2020 Google LLC
Google LLC Public Information

- Trivial In-place Access of Basic Machine Types

Trivial In-place Access of Basic Machine Types

With respect to trivial in-place access of basic machine types, protobufs, like thrift, focuses on arm's length programmer interaction and high-compression, yet high-overhead, "zig-zag" encoding for integer data, which results in and represents an effective double-buffer that increases stack or global data overhead, depending on reentrancy needs of the application. This is shown in Figure 1 below.

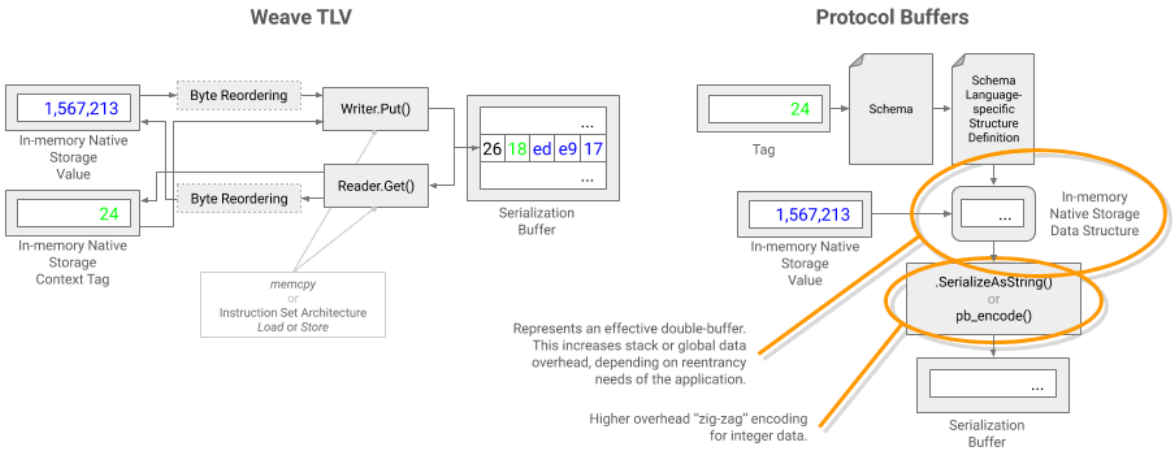

Figure 1. Illustration of Weave TLV versus Google Protocol Buffers approach to encoding.

ASN.1

Borne out of computing and telecommunications standardization efforts in the mid-1980s, ASN.1 defines a notation for data representation with a suite of defined encoding rules, encoding the notation into representations such as XML (XER) or tightly-packed binary (PER).

Due to its early standardization, ASN.1 has been broadly adopted; however, primarily in the *legacy Internet* in protocols such as LDAP or SNMP where representations such as JSON or XML have become ascendant.

ASN.1 has nearly infinite flexibility; however, this comes with trade-offs that made it unsuitable for Weave.

Ultimately, ASN.1 failed to meet the following requirements:

- Capable of Representing Basic Machine Types
- Resource Overhead
- Trivial In-place Access of Basic Machine Types
- Over-the-wire Compactness
- Lossless Translation to JSON
- Licensing and Seat Costs

Copyright © 2020 Google LLC
Google LLC Public Information

In terms of *Licensing and Seat Costs*, while ASN.1 is a fairly open standard, in our implementation survey, we were unable to find any high-performance, well-supported, open-source implementations.

EXI

The *Extensible XML Interchange* (EXI) representation was borne out of the many efforts to make the storage and transmission of the Extensible Markup Language (XML) representation more space-efficient.

It has traditionally and most-commonly been paired with the *Constrained Application Protocol* (CoAP) as well as with HTTPS in the context of the ZigBee *Smart Energy Profile* (SEP) 2.0.

Unfortunately, because of the nearly-infinite inherent expressiveness of XML and the highly-compact binary encoding, EXI failed on the following fronts:

- Resource Overhead
- Trivial In-place Access of Basic Machine Types
- Partitioned and Controlled Tag Space
- Over-the-wire Compactness
- Lossless Translation to JSON
- Market Penetration
- Proliferation and Quality of Infrastructure and Tools

That the CoAP effort has all but abandoned EXI (as the preferred CoAP application layer encoding) in favor of CBOR is an implicit confirmation of the lack of suitability of EXI for the Weave ecosystem.

Other

The following technologies were not initially surveyed when the initial analysis was done around the creation of Weave TLV, either due to similarity with a technology already under evaluation or a creation date that came after Weave TLV.

Thrift

Thrift has many similarities to *Google Protocol Buffers* and has achieved, through its adoption at Facebook, similar if not broader levels of marketplace penetration.

However, in addition, Thrift adds built-in mechanisms for both performing and managing RPC, something explicitly out of scope for Weave. Consequently, Thrift was not analyzed in depth, failing the following requirements:

- Capable of Representing Basic Machine Types
- Partitioned and Controlled Tag Space
- Resource Overhead
- Trivial In-place Access of Basic Machine Types

Copyright © 2020 Google LLC
Google LLC Public Information

CBOR

The *Concise Binary Object Representation* (CBOR) IETF draft was submitted the same month that the Weave TLV implementation was completed and, as a consequence, was unavailable for competitive evaluation at that time.

CBOR is intended to go hand-in-hand with the *Constrained Application Protocol* (CoAP) and to displace EXI as the preferred data representation for CoAP, addressing EXI's many shortcomings.

Relative to Weave TLV, CBOR has many of the same design motivations and requirements and, in its realized design, many of the same attributes. However, CBOR is at the same time, far looser by allowing any other CBOR type—including an array or structure—to act as a tag or key, thereby eliminating in its full implementation *Lossless Translation to JSON* and greatly increasing the complexity of its implementation at full specification.

From a market adoption perspective, CBOR offers a wealth of liberally-licensed open source implementations in C and C++, among other languages, including those listed in Table 3 below.

Project	Language	Build	License	Maintainer	Active Project?
tinycbor	C	make	MIT	Intel	Y
QCBOR	C	make	BSD 3-Clause	Laurence Lundblade	Y
jsoncons	C++	cmake	Boost Software License v1	Daniel Parker	Y
cbor-lite	C++	cmake	Public Domain	Kurt Zeilenga	Y
libcbor	C	cmake	MIT	Pavel Kalvoda	Y
cborg	C++	-	Apache v2	ARM	N
cppbor	C++	cmake	BSD 2-Clause	David Preece	N
cbor-cpp	C++	cmake	Apache v2	Stanislav Ovsiannikov	N
cn-cbor	C	cmake	MIT	Carsten Bormann	N
scsp	C/C++	make	Apache v2 MIT	Vitaly Shukela	N

Table 3. Survey of open source CBOR C or C++ implementations.

Relative to Weave TLV, the downsides of CBOR are:

- Partitioned and Controlled Tag Space
- Trivial In-place Access of Basic Machine Types

Copyright © 2020 Google LLC
Google LLC Public Information

Partitioned and Controlled Tag Space

As documented in "*Concise Binary Object Representation (CBOR) Tags*" [6], the tag space for CBOR is partitioned into 8-bits of either *standards action* or *specification required* reserved space and the remaining 56-bits as first-come, first-served. This effectively makes tags in CBOR ecosystem-specific and means that across ecosystems, data in flight or at rest encoded with CBOR may collide, at worst, or be ambiguously tagged, at best.

To avoid or overcome either of these situations, CBOR encoded data would need to be further qualified or wrapped, increasing the complexity and size of the encoded data.

Trivial In-place Access of Basic Machine Types

CBOR falls somewhere between encoding and serialization formats such as Protocol Buffers or Thrift and formats such as Weave TLV. Where some types, and sizes of those types, can support trivial in-place access and others use more complicated encoding schemes that preclude such access. For data patterns that bias towards the latter, those applications will see an increase in code size due to the implementation complexity necessary to encode and decode such data.

Flatbuffers

Flatbuffers is a cross-platform serialization library developed by Google, as an alternative to protobufs, optimized for game development and other performance-sensitive applications. Emphasis is placed on the following criteria, many of which are very similar to Weave TLV:

- Trivial In-place Access of Basic Machine Types
- Capable for Forward- and Backward-Compatibility
- Capable of Representing Optional Content
- Low Resource Overhead

Flatbuffers, across most quantitative comparison criteria, is close to Weave. However, it loses out because of the overhead of the Flatbuffer *vtable* which, while offering fast access to arbitrary elements of the buffer, increases over-the-wire size.

In addition, Flatbuffers has dependencies on both C++ STL and the C++11 standard, which increase the platform requirements relative to Weave.

Conclusion

In the rich world of historical and current competitive technologies in the realm of platform-independent data presentation and exchange, Weave TLV provides a solid balance of attributes. These attributes make it well-suited to provide a long-lasting, evolving foundation for deeply-embedded, Internet-connected devices from multiple parties. At the same time, easy and seamless translation to JSON make it an ideal data representation vehicle to match the needs of Internet-enabled services today.

Copyright © 2020 Google LLC
Google LLC Public Information

Appendix A: Size Analysis

To exercise and quantify not only the feature capabilities of each representation but also the efficiency of various data representation implementations, we decided to create a test case.

The test case is based on a C/C++ native struct, shown in Listings 1 through 7 below. From there, a data representation was made in each technology and the over-the-wire size measured as well as the size of the resulting code and data necessary to marshal the native struct into the representation. The summary of these sizes is shown in Table 4 below. In the table, *Core* represents the size contributions, if any, of the core library supporting the representation, common to any application use case. *Application* represents the size contributions for encoding the particular use cases exemplified in this document.

```
enum CapType {
      Proximity = 1,
      Temperature = 2,
      SixLoWPAN = 3,
      WiFi = 4,
      Gateway = 5,
      Heat = 6,
      Cool = 7,
      Light = 8
};
```

Listing 1. Native C language test definition for a capability type enumeration.

```
struct KVPair {
      char kvKey[11];
      char kvValue[33];
};
```

Listing 2. Native C language test definition for a key/value pair.

```
struct Capability {
      CapType capType;
      bool enabled;
      KVPair kvpair;
};
```

Listing 3. Native C language test definition for a capability.

```
struct Device {
      uint16_t vendorId;
      uint16_t deviceType;
      uint8_t serialNumberLength;
      char serialNumber[32];
      uint8_t publicKey[16];
      uint32_t lastSeenTime;
      Capability capabilities[32];
```


Copyright © 2020 Google LLC
Google LLC Public Information

```
};
```

Listing 4. Native C language test definition for a network device.

```
struct Fabric {
	char id[41];
	uint8_t groupSecret[16];
	char password[2][17];
};
```

Listing 5. Native C language test definition for a network fabric.

```
struct Directory {
	Fabric fabric;
	Device device[4];
};
```

Listing 6. Native C language test definition for a network directory.

```
static const struct Directory sDirectory = {
	// Fabric
	{
		"Fabric #1",
		{ 'a', 's', 'd', 'f', 'q', 'w', 'e', 'r', 't', 'y' },
		{
			"password1",
			"password2"
		},
	},
	// Devices
	{
		// Device 0
		{
			0x235A,
			0x0002,
			16,
			{
				'0', '2', 'A', 'A', '0', '1', 'A', 'B',
				'3', '2', '1', '2', '0', '0', '0', '0'
			},
			{
				0xb9, 0xad, 0x7f, 0x03,
				0x9f, 0x0f, 0xf1, 0x67,
				0x79, 0xb7, 0x39, 0xdd,
				0x93, 0x88, 0xae, 0xea
			},
			0x0,
			{
				{ Temperature, true, { "key1", "val1" } },
				{ }
			}
		},
		// Device 1
```


Copyright © 2020 Google LLC
Google LLC Public Information

```
                {
                      0x235A,
                      0x0002,
                      16,
                      {
                            '0', '2', 'A', 'A', '0', '1', 'A', 'B',
                            '3', '2', '1', '2', '0', '0', '0', '1'
                      },
                      {
                            0xb0, 0x3d, 0x7a, 0x07,
                            0xf2, 0x89, 0xe5, 0x34,
                            0x23, 0x55, 0xd8, 0x4e,
                            0xb7, 0xda, 0xec, 0x71
                      },
                      0x0,
                      {
                            { Proximity, true, { "key2", "val2" } },
                            { }
                      }
                },
                // Device 2
                {
                      0x235A,
                      0x0002,
                      16,
                      {
                            '0', '2', 'A', 'A', '0', '1', 'A', 'B',
                            '3', '2', '1', '2', '0', '0', '0', '2'
                      },
                      {
                            0x88, 0x6c, 0x74, 0x27,
                            0x7b, 0x65, 0x8e, 0xf5,
                            0x1d, 0xc7, 0xd2, 0xb0,
                            0x4f, 0x80, 0x9a, 0xff
                      },
                      0x0,
                      {
                            { Light, true, { "key3", "val3" } },
                            { }
                      }
                },
                // Device 3
                {
                      0x235A,
                      0x0002,
                      16,
                      {
                            '0', '2', 'A', 'A', '0', '1', 'A', 'B',
                            '3', '2', '1', '2', '0', '0', '0', '3'
                      },
                      {
                            0xbd, 0x14, 0x06, 0xaf,
                            0x9d, 0xeb, 0xe4, 0xc0,
```


Copyright © 2020 Google LLC
Google LLC Public Information

```
                                0x41, 0xbc, 0x0e, 0xf8,
                                0x96, 0xfb, 0x69, 0x1e
                        },
                        0x0,
                        {
                                { Temperature, true, { "key4", "val4" } },
                                { }
                        }
                },
        }
};
```

Listing 7. Native C language test declaration for a sample network directory data.

Representation	Product	Over-the-wire Size / B	Application			Core	
			Serialization Code Size / B	Serialization Data Size / B	Stack Data Size / B	Code Size / B	Data Size / B
JSON	mjson-1.3	965	4,162[1]	0	28	22,620[2]	0
memcpy	-	6,591[3]	0	0	0	0	0
Google Protocol Buffers	protobuf-2.6.0 (libprotobuf size-optimized)	313	25,448	112	60	632,040	857
	protobuf-2.6.0 (libprotobuf speed-optimized)		25,452	112			
	protobuf-2.6.0 (libprotobuf-lite)		24,886	106		68,904	156
	nanopb-0.3.1		996	0	7,104	8,200	0
Thrift	thrift-0.9.2 (binary)	398	41,762	308	212	35,404[4]	44[4]
	thrift-0.9.2 (compact)	216	44,990		196		
	thrift-0.9.2 (debug)	2133	41,879			59,244[5]	52[5]
	thrift-0.9.2 (dense)	181	43,931		228	65,684[6]	52[6]
	thrift-0.9.2 (JSON)	776	34,577		204	85,632[7]	190[7]
Flatbuffers	flatbuffers-v1.1.0-70-g932b22f	728	7,430	4	132	0	0
Weave TLV	Nest Labs[8]	422	1,252	0	124	8,000	0

Copyright © 2020 Google LLC
Google LLC Public Information

CBOR	QCBOR-06350ea (global buffer)	321	1,195	384	124	9,320	0
	QCBOR-06350ea (heap buffer)		1,207	0	132		
	QCBOR-06350ea (stack buffer)		1,191	0	508		
	tinycbor-ef28900 (global buffer)	305	2,618	384	204	8,043[9]	0
	tinycbor-ef28900 (heap buffer)		2,638	0	204		
	tinycbor-ef28900 (stack buffer)		2,614	0	588		

Table 4. Comparison of over-the-wire, data, and code sizes for various encoding representations and products for sample test data.

[1] The baseline *mjson* library does not make it programmatically convenient to generate JSON nor does it include a Base64 encoder for byte streams. This size includes 1,555 bytes for a JSON builder class, 223 bytes for read-only JSON key strings, 292 bytes for a Base64 encoder, and 2,092 bytes for the actual marshalling code.

[2] Excludes *json_helper.o*.

[3] Packed size. The non-packed size is 6,988 bytes.

[4] Sized for: *Thrift.o, Util.o, TMultiplexedProtocol.o, TTransportException.o, TTransportUtils.o,* and *TBufferTransports.o*.

[5] Sized for: *Thrift.o, Util.o, TDebugProtocol.o, TMultiplexedProtocol.o, TTransportException.o, TTransportUtils.o,* and *TBufferTransports.o*.

[6] Sized for: *Thrift.o, Util.o, TDenseProtocol.o, TMultiplexedProtocol.o, TTransportException.o, TTransportUtils.o,* and *TBufferTransports.o*.

[7] Sized for: *Thrift.o, Util.o, TJSONProtocol.o, TBase64Utils.o, TMultiplexedProtocol.o, TTransportException.o, TTransportUtils.o, and TBufferTransports.o*.

[8] Version 2.0.

[9] Sized for: *cborencoder.o*, *cborencoder_close_container_checked.o*, *cborparser.o*, *cborparser_dup_string.o*.

All code was compiled using the following compiler:

- *Freescale "Poky" 1.6 (arm-poky-linux-gnueabi-gcc 4.8.2)*

with the following options:

- -march=armv7-a -mtune=cortex-a9 -mfpu=neon -mfloat-abi=hard -ftree-vectorize -fno-forward-propagate -Os -g -Wall -Wchar-subscripts -Wformat -Wparentheses -Wreturn-type -Wsequence-point -Wframe-larger-than=9472 -Wshadow -Wuninitialized -Wunused -Wno-psabi -Werror -Wimplicit -Wmissing-prototypes -Wstrict-prototypes

Copyright © 2020 Google LLC
Google LLC Public Information

JSON

This comparison used *msjon-1.3* with the keys shown in Listing 8 below to generate the JSON representation shown in Listing 9 below, derived from the native C/C++ data representation from Listing 7 above. Byte streams were accommodated using Base64 encoded strings of the data, as is typical for JSON representations.

```
const char * const kCapabilitiesKey = "capabilities";
const char * const kCapabilityTypeKey = "capType";
const char * const kDevicesKey = "devices";
const char * const kDeviceTypeKey = "deviceType";
const char * const kDirectoryKey = "directory";
const char * const kEnabledKey = "enabled";
const char * const kFabricKey = "fabric";
const char * const kGroupSecretKey = "groupSecret";
const char * const kIdentifierKey = "id";
const char * const kPropertyKeyKey = "kvKey";
const char * const kPropertyPairKey = "kvpair";
const char * const kPropertyValueKey = "kvValue";
const char * const kLastSeenTimeKey = "lastSeenTime";
const char * const kPasswordsKey = "password";
const char * const kPublicKeyKey = "publicKey";
const char * const kSerialNumberKey = "serialNumber";
const char * const kVendorIdentifierKey = "vendorId";
```

Listing 8. The JSON key strings used for the JSON representation of the test declaration for a sample network directory data.

```
“directory” : {
      “fabric” : {
            “id” : “Fabric #1”,
            “groupSecret” : “asdfqwerty”,
            “password” : [
                  “password1”,
                  “password2”
            ]
      },
      “devices” : [
            {
                  “vendorId” : 9050,
                  “deviceType” : 2,
                  “serialNumber” : “02AA01AB32120000”,
                  “publicKey” : “ua1/A58P8Wd5tzndk4iu6g==”,
                  “lastSeenTime” : 0,
                  “capabilities” : [
                        {
                              “capType” : 2,
                              “enabled” : true,
                              “kvpair” : {
                                    “kvKey” : “key1”,
```


Copyright © 2020 Google LLC
Google LLC Public Information

```
                                    "kvValue" : "val1"
                              }
                        }
                  ]
            },
            {
                  "vendorId" : 9050,
                  "deviceType" : 2,
                  "serialNumber" : "02AA01AB32120001",
                  "publicKey" : "sD16B/KJ5TQjVdhOt9rscQ==",
                  "lastSeenTime" : 0,
                  "capabilities" : [
                        {
                              "capType" : 1,
                              "enabled" : true,
                              "kvpair" : {
                                    "kvKey" : "key2",
                                    "kvValue" : "val2"
                              }
                        }
                  ]
            },
            {
                  "vendorId" : 9050,
                  "deviceType" : 2,
                  "serialNumber" : "02AA01AB32120002",
                  "publicKey" : "iGx0J3tljvUdx9KwT4Ca/w==",
                  "lastSeenTime" : 0,
                  "capabilities" : [
                        {
                              "capType" : 8,
                              "enabled" : true,
                              "kvpair" : {
                                    "kvKey" : "key3",
                                    "kvValue" : "val3"
                              }
                        }
                  ]
            },
            {
                  "vendorId" : 9050,
                  "deviceType" : 2,
                  "serialNumber" : "02AA01AB32120003",
                  "publicKey" : "vRQGr53r5MBBvA74lvtpHg==",
                  "lastSeenTime" : 0,
                  "capabilities" : [
                        {
                              "capType" : 2,
                              "enabled" : true,
                              "kvpair" : {
                                    "kvKey" : "key4",
                                    "kvValue" : "val4"
                              }
```


Copyright © 2020 Google LLC
Google LLC Public Information

```
                                }
                            ]
                        }
                ]
        }
```

Listing 9. An example JSON representation of the test declaration for a sample network directory data.

memcpy

This comparison simply evaluates the size of the native data representation show in Listing 7 above for the over-the-wire size because no translation is required to an over-the-wire format. Consequently, the system resource costs to marshal and unmarshal the data, above and beyond the `memcpy` function from the C Standard Library, are zero.

However, while this encoding offers the best format transformation costs, the over-the-wire overhead is high when optional, repeating but otherwise empty fields are considered such as the `Capabilities` array.

Google Protocol Buffers

The comparison of Google Protocol Buffers examines both the reference Google *libprotobuf* implementation as well as the highly-tuned *nanopb* implementation.

libprotobuf

For the Google reference *libprotobuf* implementation, three code generation optimization options were evaluated:

- size
- speed
- lite

using the description language definition shown in Listing 10.

```
enum CapType {
    Proximity = 1;
    Temperature = 2;
    SixLoWPAN = 3;
    WiFi = 4;
    Gateway = 5;
    Heat = 6;
    Cool = 7;
    Light = 8;
}

message KVPair {
    required string kvKey = 1;
```


Copyright © 2020 Google LLC
Google LLC Public Information

```
    optional string kvValue = 2;
}

message Capability {
    required CapType capType = 1;
    required bool enabled = 2;
    required KVPair kvpair = 3;
}

message Device {
    required uint32 vendorId = 1;
    required uint32 deviceType = 2;
    required string serialNumber = 3;
    optional bytes publicKey = 4;
    optional uint32 lastSeenTime = 5;
    repeated Capability capabilities = 6;
}

message Fabric {
    required string id = 1;
    required bytes groupSecret = 2;
    repeated string password = 3;
}

message Directory {
    required Fabric fabric = 1;
    repeated Device device = 2;
}
```

Listing 10. An example Google Protocol Buffers representation of the test declaration for a sample network directory data.

Size

```
option optimize_for = CODE_SIZE;

import "tlv-white-paper-protobuf-data.proto";
```

Listing 11. An example Google Protocol Buffers representation optimizing for code size.

Speed

```
option optimize_for = SPEED;

import "tlv-white-paper-protobuf-data.proto";
```

Listing 12. An example Google Protocol Buffers representation optimizing for code speed.

libprotobuf-lite

```
option optimize_for = LITE_RUNTIME;
```


Copyright © 2020 Google LLC
Google LLC Public Information

```
import "tlv-white-paper-protobuf-data.proto";
```

Listing 13. An example Google Protocol Buffers representation optimizing for code size and the *lite* runtime.

nanopb

The *nanopb* implementation of Google Protocol Buffers is 100% compatible with the over-the-wire format of the reference implementation, but has code-generation that is optimized far above and beyond even the reference *libprotobuf-lite* optimization. It uses the same description language, but Listing 14 leverages the *nanopb* `max_count` and `max_size` properties to reduce code size further by eliminating run time callbacks that would otherwise compute the maximum length of arrays, strings, and byte streams.

```
import "nanopb.proto";

enum CapType {
    Proximity = 1;
    Temperature = 2;
    SixLoWPAN = 3;
    WiFi = 4;
    Gateway = 5;
    Heat = 6;
    Cool = 7;
    Light = 8;
}

message KVPair {
    required string kvKey = 1 [(nanopb).max_size = 11];
    optional string kvValue = 2 [(nanopb).max_size = 33];
}

message Capability {
    required CapType capType = 1;
    required bool enabled = 2;
    required KVPair kvpair = 3;
}

message Device {
    required uint32 vendorId = 1;
    required uint32 deviceType = 2;
    required string serialNumber = 3 [(nanopb).max_size = 32];
    optional bytes publicKey = 4 [(nanopb).max_size = 16];
    optional uint32 lastSeenTime = 5;
    repeated Capability capabilities = 6 [(nanopb).max_count = 32];
}

message Fabric {
    required string id = 1 [(nanopb).max_size = 41];
    required bytes groupSecret = 2 [(nanopb).max_size = 16];
```


Copyright © 2020 Google LLC
Google LLC Public Information

```
      repeated   string   password   =   3   [(nanopb).max_count   =   2,
(nanopb).max_size = 17];
}

message Directory {
    required Fabric fabric = 1;
    repeated Device device = 2 [(nanopb).max_count = 4];
}
```

Listing 14. An example Google Protocol Buffers representation for the *nanopb* implementation and code generation plugin.

Thrift

The comparison of Apache Thrift examines the reference implementation and within that, five different over-the-wire encoding formats:

- binary
- compact
- debug
- dense
- JSON

using the description language definition shown in Listing 15 below.

```
enum CapType {
    Proximity = 1;
    Temperature = 2;
    SixLoWPAN = 3;
    WiFi = 4;
    Gateway = 5;
    Heat = 6;
    Cool = 7;
    Light = 8;
}

struct KVPair {
    1: required string kvKey;
    2: optional string kvValue;
}

struct Capability {
    1: required CapType capType;
    2: required bool enabled;
    3: KVPair kvpair;
}

struct Device {
    1: required i32 vendorId;
    2: required i32 deviceType;
    3: required string serialNumber;
```


Copyright © 2020 Google LLC
Google LLC Public Information

```
    4: optional binary publicKey;
    5: optional i32 lastSeenTime;
    6: list <Capability> capabilities;
}

struct Fabric {
    1: required string id;
    2: required binary groupSecret;
    3: list<string> password;
}

struct Directory {
    1: required Fabric fabric;
    2: list<Device> device;
}
```

Listing 15. An example Apache Thrift representation of the test declaration for a sample network directory data.

Flatbuffers

The comparison with Flatbuffers uses the description language definition shown in Listing 16 below.

```
enum CapType : byte {
    Unknown = 0,
    Proximity = 1,
    Temperature = 2,
    SixLoWPAN = 3,
    WiFi = 4,
    Gateway = 5,
    Heat = 6,
    Cool = 7,
    Light = 8,
}

table KVPair {
    kvKey:string (required);
    kvValue:string;
}

table Capability {
    capType:CapType;
    enabled:bool;
    kvpair:KVPair (required);
}

table Device {
    vendorId:uint;
    deviceType:uint;
    serialNumber:string (required);
    publicKey:[ubyte];
```


Copyright © 2020 Google LLC
Google LLC Public Information

```
    lastSeenTime:uint;
    capabilities:[Capability];
}

table Fabric {
    id:string (required);
    groupSecret:[ubyte] (required);
    password:[string] (required);
}

table Directory {
    fabric:Fabric (required);
    device:[Device];
}
```

Listing 15. An example Flatbuffers representation of the test declaration for a sample network directory data.

Weave

The comparison with Weave uses the profile identifier and context tags specified shown in Listing 17 below.

```
#define kTLVWhitePaper_Profile                  0xFFFD0001

#define kTLVWhitePaper_Directory                 1
#define kTLVWhitePaper_Fabric                    2
#define kTLVWhitePaper_Fabric_Identifier         3
#define kTLVWhitePaper_Fabric_GroupSecret        4
#define kTLVWhitePaper_Fabric_Passwords          5
#define kTLVWhitePaper_Fabric_Password           6
#define kTLVWhitePaper_Devices                   7
#define kTLVWhitePaper_Device                    8
#define kTLVWhitePaper_Device_VendorID           9
#define kTLVWhitePaper_Device_DeviceType        10
#define kTLVWhitePaper_Device_SerialNumber      11
#define kTLVWhitePaper_Device_PublicKey         12
#define kTLVWhitePaper_Device_LastSeenTime      13
#define kTLVWhitePaper_Device_Capabilities      14
#define kTLVWhitePaper_Capability               15
#define kTLVWhitePaper_Capability_Type          16
#define kTLVWhitePaper_Capability_Enabled       17
#define kTLVWhitePaper_Capability_KVPair        18
#define kTLVWhitePaper_KVPair_Key               19
#define kTLVWhitePaper_KVPair_Value             20
```

Listing 17. The profile identifier and context tags used for a Weave representation of the test declaration for a sample network directory data.

Copyright © 2020 Google LLC
Google LLC Public Information

CBOR

The comparison with CBOR uses the tags specified shown in Listing 18 below.

```
#define kTLVWhitePaper_Directory                1
#define kTLVWhitePaper_Fabric                   2
#define kTLVWhitePaper_Fabric_Identifier        3
#define kTLVWhitePaper_Fabric_GroupSecret       4
#define kTLVWhitePaper_Fabric_Passwords         5
#define kTLVWhitePaper_Fabric_Password          6
#define kTLVWhitePaper_Devices                  7
#define kTLVWhitePaper_Device                   8
#define kTLVWhitePaper_Device_VendorID          9
#define kTLVWhitePaper_Device_DeviceType       10
#define kTLVWhitePaper_Device_SerialNumber     11
#define kTLVWhitePaper_Device_PublicKey        12
#define kTLVWhitePaper_Device_LastSeenTime     13
#define kTLVWhitePaper_Device_Capabilities     14
#define kTLVWhitePaper_Capability              15
#define kTLVWhitePaper_Capability_Type         16
#define kTLVWhitePaper_Capability_Enabled      17
#define kTLVWhitePaper_Capability_KVPair       18
#define kTLVWhitePaper_KVPair_Key              19
#define kTLVWhitePaper_KVPair_Value            20
```

Listing 18. The tags used for a CBOR representation of the test declaration for a sample network directory data.

Note that the CBOR implementation may be less than ideal relative to its JSON equivalent due to the use of tags only for maps rather than tags plus integral keys (there is a fine distinction there in CBOR, since tags are regarded as optional).

Consequently, the *Over-the-wire Size* size shown in Table 4 above for CBOR may be slightly smaller than if a strict JSON-equivalent approach had been taken using integral keys. Examples of this approach are shown in Listing 19 and Listing 20 for *QCBOR* and *tinycbor*, respectively.

```
// Group Secret

QCBOREncode_AddBytesToMapN(&lEncodeContext, kTLVWhitePaper_Fabric_GroupSecret,
(UsefulBufC){ &nl::Native::gDirectory.fabric.groupSecret[0], 16 });

// Passwords

{
    QCBOREncode_AddTag(&lEncodeContext, kTLVWhitePaper_Fabric_Passwords);

    QCBOREncode_OpenArray(&lEncodeContext);

    {
        QCBOREncode_AddSZString(&lEncodeContext,
&nl::Native::gDirectory.fabric.password[0][0]);
```


Copyright © 2020 Google LLC
Google LLC Public Information

```
        QCBOREncode_AddSZString(&lEncodeContext,
&nl::Native::gDirectory.fabric.password[1][0]);
    }

    QCBOREncode_CloseArray(&lEncodeContext);
}
```

Listing 19. QCBOR CBOR encoding approach used for group secret and passwords sample data.

```
// Group Secret

lError = cbor_encode_tag(&lFabricEncodeContext,
kTLVWhitePaper_Fabric_GroupSecret);
TEST_ASSERT(lError == CborNoError);

lError = cbor_encode_byte_string(&lFabricEncodeContext,
&nl::Native::gDirectory.fabric.groupSecret[0], 16);
TEST_ASSERT(lError == CborNoError);

// Passwords

{
    CborEncoder         lPasswordsEncodeContext;

    lError = cbor_encode_tag(&lFabricEncodeContext,
kTLVWhitePaper_Fabric_Passwords);
    TEST_ASSERT(lError == CborNoError);

    lError = cbor_encoder_create_array(&lFabricEncodeContext,
&lPasswordsEncodeContext, CborIndefiniteLength);
    TEST_ASSERT(lError == CborNoError);

    {
        lError = cbor_encode_text_stringz(&lPasswordsEncodeContext,
&nl::Native::gDirectory.fabric.password[0][0]);
        TEST_ASSERT(lError == CborNoError);

        lError = cbor_encode_text_stringz(&lPasswordsEncodeContext,
&nl::Native::gDirectory.fabric.password[1][0]);
        TEST_ASSERT(lError == CborNoError);
    }

    lError = cbor_encoder_close_container(&lFabricEncodeContext,
&lPasswordsEncodeContext);
    TEST_ASSERT(lError == CborNoError);
}
```

Listing 20. Tinycbor CBOR encoding approach used for group secret and passwords sample data.

Copyright © 2020 Google LLC
Google LLC Public Information

References

1. Aimonen, Petteri. *nanopb - protocol buffers with small code size*. http://koti.kapsi.fi/jpa/nanopb/.
2. Github. flatbuffers. https://google.github.io/flatbuffers/.
3. Google LLC. *Protocol Buffers / API Reference / C++ Generated Code*. 2014-09-03.
4. Google LLC. *Weave Message Layer: Protocol Specification*. Version 1.1.1. 2017-09-03.
5. Google LLC. *Weave TLV Format*. Revision 4. 2013-05-20.
6. IANA. *Concise Binary Object Representation (CBOR) Tags*. 2020-03-02.
7. Sourceforge. *exip*. http://exip.sourceforge.net/.
8. Sourceforge. *mjson*. http://mjson.sourceforge.net/.
9. Walkin, Lev. *ASN.1 Exposed*. http://lionet.info/asn1c/.

Copyright © 2020 Google LLC
Google LLC Public Information