ProtoN: A Fast Binary Protocol for JSON-like Data Transfer

ARON SZANTO, Harvard University
NICHOLAS BOUCHER, Harvard University

1 INTRODUCTION

JSON has emerged as one of the leading data transfer formats used on the Internet today. It is a ubiquitous protocol for communication in web applications and is the lingua franca for public-facing APIs across the web[1]. Indeed, programs as important as Google and Facebook's user-facing web applications depend on JSON to move data between application databases and client browsers.

Though standard, the JSON data format is heavyweight and slow, making it poorly suited for growing data loads and for modern applications that shift computation to the client side. Due to its encoding of each datum (integer or floating point digit, string character, etc.) as an at least 1 byte long Unicode character, it is a naive way to transfer data. A protocol that examines the data before encoding it to determine the optimal format would be smaller and thus faster across the wire.

One solution to this was introduced by Google as the Protocol Buffers[2] service. Protocol Buffers are a framework for binary data interchange. They require prespecifying the structure of a message before it is sent and generating code to compile each message into a lightweight format by consulting the type field in the specification for each datum in

[1]Over 50% of the APIs listed on https://www.programmableweb.com/apis/, the most comprehensive directory of its kind, use JSON.
[2]https://developers.google.com/protocol-buffers/

the message. As desired, this is much more efficient than JSON, but this reduction in size comes at the cost of flexibility: any message that is sent has to have a type schematic that is predefined. Unfortunately, these schematics are brittle and developers are prone to make mistakes in them, potentially resulting in loss of forward- or backward-compatibility. Though efficient, Protocol Buffers thus leave to be desired a binary encoding scheme that is both flexible and small.

In this work, we introduce Protocol: Nimble (ProtoN), a lightweight, adaptive, and fast binary format as a drop-in replacement for JSON, demonstrating both analytically and experimentally that it requires less space to encode the same information as *any* JSON object. We find that ProtoN is on average 31% smaller than standard JSON in a wide variety of both hand-crafted and automatically-generated test cases. This bodes well for the supplanting of JSON with a binary format that maintains the functionality of JSON but encodes data much less wastefully.

The remainder of the paper will proceed as follows: Section 2 details the JSON and ProtoN data formats and defines the specific binary encoding inherent to ProtoN. Section 3 compares JSON and ProtoN analytically, showing that under virtually all use-cases ProtoN is provably smaller than JSON. Section 4 describes implementation details and introduces the experimentation suite and the results from both random and handcrafted testing. Section 5 discusses the implications of this project and concludes.

2 DATA FORMATS

2.1 JSON

The principal design goal of ProtoN is to encode JSON-like data in a small binary format. To this end, we begin with an overview of the JSON data format.

JSON[3] is built on two structures: the object and the list. Objects are a collection of name-value pairs, while lists are ordered arrays of values. Values are strings, numbers, booleans, or nulls, each with specific allowed formatting. JSON is flexible and recursive, able to encode a large variety of data. Under the hood, however, JSON falters by representing each bit of data in its string format. This means that the value `1` requires one byte, while the string value `"1"` requires three and the value `null` requires four. Figure 1 defines the formal JSON semantics; this is the data format the ProtoN's protocol supports as well. As

[3]as specified in http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

such, ProtoN is *complete*: any data that could be represented by JSON may be encoded in ProtoN as well.

JSON's disadvantage is that every bracket, quote, colon, comma, and other supporting character associated with the format is vital to its encoding. Because whitespace is ignored by JSON, there is no way to do away with these extraneous characters without introducing ambiguity into the data. ProtoN seeks to do better.

2.2 ProtoN

ProtoN defines an abstract binary protocol whose encoding space is a superset of the JSON data space. The insight of ProtoN is that it operates at the bit level, inferring the smallest possible number of bits it needs to encode each segment of data unambiguously. There are three layers of ProtoN entities: binary encodings, primitives, and containers.

2.2.1 Binary Encodings. The smallest unit in ProtoN is a binary encoding, a bit-level specification of data. binary encodings include 3-bit typecodes (denoting the type of data that follows), raw numeric or character data, such as 32-bit integers or unicode strings, and other metadata. One key feature of ProtoN binary encodings is that there may be several flavors of encodings for each abstract datatype. For example, the protocol supports 8-, 16-, 32-, and 64-bit signed integers. This allows the encoder to select the smallest bitwise (rather than bytewise) encoding that captures the data. Figure 2 defines the seven binary encodings resident in the ProtoN specification. Note that the `len` encoding flexibly encodes an unsigned integer by selecting the smallest representation possible. The `len` is fundamental to building complex data types such as strings and lists.

2.2.2 Primitives. The middle unit is composed of a collection of primitive datatypes: Null, String, Integer, Float, and Boolean. These are the fundamental types for all data transfer (and indeed are those primitives specified by JSON). It is at this layer that ProtoN's efficiency becomes obvious. All primitives begin with a 3-bit typecode.

To encode a primitive integer, the ProtoN encoder determines the smallest number of bits (choosing between 8, 16, 32, and 64) that the data can be represented by. It then encodes its selection as a two bit number `sz`. The full integer encoding is then the three bit typecode for an integer followed by the two bit `sz` value followed by a $2^{(\mathtt{sz}+3)}$-bit representation of the number.

```
object
    {}
    {members}
members
    pair
    pair,members
pair
    string:value
array
    []
    [elements]
elements
    value
    value,elements
value
    string
    number
    object
    array
    true
    false
    null

string
    ""
    "chars"
chars
    char
    char,chars
char
    any Unicode character  except:
        " or \ or control character
    \"
    \\
    \/
    \b
    \f
    \n
    \r
    \t
    \u four-hex-digits
number
    int
    int frac
    int exp
    int frac exp
int
    digit
    digit digits
    -digit
    -digit digits
frac
    .digits
exp
    e digits
digits
    digit
    digit digits
e
    e
    e+
    e-
    E
    E+
    E-
```

Fig. 1. Abstract JSON semantics

```
Numerical Representation
- **0b...** denotes binary digit *(1 bit)*
- **0q...** denotes quaternary digit *(2 bits)*
- **0o...** denotes octal digit *(3 bits)*

Byte Order
    Network Byte Order (Big-Endian)

Encodings
- int*n*: *n*-bit signed integer
- uint*n*: *n*-bit unsigned integer
- float64: IEEE 754 64-bit signed floating point number
- bool: 1-bit encoding of boolean values {0b0: `False`; 0b1: `True`}
- len: <uint2, uint8|uint16|uint32|uint64>
- String: <len, UTF-8 encoded string>
- ShortStr: <uint3, UTF-8 encoded string>


Typecodes:

Primitives (Prim)

- **PrimNull**: *0o0*
- **PrimString**: *0o1* <String\>
- **PrimInt**: *0o2* <uint2, int8|int16|int32|int64\>
- **PrimFloat**: *0o3* <bool, ShortStr|float64\>
- **PrimBool**: *0o4* <bool\>

Pairs (Pair)
- **ConPair**: *0o5* <String, Prim|Con>

Containers (Con)
- **ConList**: *0o6* <len, {Prim|Con}\*>
- **ConObject**: *0o7* <len, {Key}\*>
```

Fig. 2. ProtoN Binary Format

Strings are encoded as a `len` type followed by a UTF-8 encoding of the string.

Floats are encoded either as a 64-bit IEEE 754 floating-point number or as a string, depending on which is smaller (i.e., the decision is made depending on whether the string representation of the number is smaller than 8 characters long). Because JSON floats are interpreted as 64-bit floats, this maintains full precision.

2.2.3 Containers. The largest unit of encoding in ProtoN is a *message*, akin to a JSON object. At its top level, a message is a container – a list or an object, defined as in JSON. A message is specified by the protocol version number followed by exactly one container.

A list is encoded as a list typecode, then a `len` type, then an arbitrary number of primitives or containers.

A ProtoN object is a collection of `Pair` types, which (like their JSON counterparts), are tuples of String keys and arbitrary values. Like lists, objects are headed by their lengths.

3 COMPARISON

To demonstrate that ProtoN encodings are smaller than JSON, we consider every data type that JSON supports.

Integers in JSON are encoded as digitwise unicode strings. Therefore, a JSON integer x has (in the best case, where the x is positive and thus does not require a minus sign) $8 * (\lfloor \log_{10} x \rfloor + 1 + 1)$ bits, where second 1 corresponds to the colon, comma, or brace preceding the integer, depending on whether it is part of an object or (at the front of a) list. ProtoN requires 3 bits for the integer typecode and 2 bits to encode the size of the integer. We handle the four sizes separately:

(1) The smallest (by space) decimal number that an 8-bit signed integer encodes is 0, which requires two bytes (counting the leading character) in JSON, or 16 bits. ProtoN requires the 5 bits of overhead plus 8 bits for the integer, or 13 bits total.
(2) The smallest decimal number that a 16-bit signed integer encodes (that an 8-bit integer cannot) is $2^7 = 64$, which requires three bytes (counting the leading character) in JSON, or 24 bits. ProtoN requires the 5 bits of overhead plus 16 bits for the integer, or 21 bits total.
(3) The smallest decimal number that a 32-bit signed integer encodes (that a 16-bit integer cannot) is $2^{15} = 32768$, which requires six bytes (counting the leading

character) in JSON, or 48 bits. ProtoN requires the 5 bits of overhead plus 32 bits for the integer, or 37 bits total.

(4) The smallest decimal number that a 64-bit signed integer encodes (that a 32-bit integer cannot) is $2^{31} = 2147483648$, which requires eleven bytes (counting the leading character) in JSON, or 88 bits. ProtoN requires the 5 bits of overhead plus 64 bits for the integer, or 69 bits total.

Therefore, every integer value up to the max value of a 64-bit signed integer is encoded smaller in ProtoN than in JSON. (For integers larger than 64-bits, the user should represent the integer as a string).

Floats in ProtoN are encoded either as strings or as 64-bit floating point numbers. In JSON, floats are encoded characterwise. ProtoN chooses the optimal encoding– if the string representation of the value is smaller than 8 characters, then ProtoN encodes it identically to JSON. Otherwise, it is encoded as a 64-bit floating point value, which will be smaller than the JSON string representation (but maintains precision since even string-floats are interpreted as 64-bit floating point values on the receiving side). Considering the leading character as above (along with the decimal point), floats are strictly smaller in ProtoN than in JSON.

Null values are encoded simply as the 3-bit typecode in ProtoN, but require four bytes to spell the word `null` in JSON.

Booleans are encoded as the 3-bit typecode followed by a 1-bit true or false flag. JSON requires the full word `true` or `false` (4 or 5 bytes, respectively).

Strings of length n in JSON require $n + 3$ bytes because of the leading character and the requirement that they be bookended by double quotes. In ProtoN, strings of length n are encoded as a `len` type followed by the string. JSON is smaller than ProtoN *if and only if* $n > 2^{16} - 1 = 65525$ because the three bytes of overhead would be dominated by ProtoN's 2-bit + 32-bit overhead. However, we posit that the use cases in which JSON is employed to transmit many strings of length greater than 65525 are quite sparse. This is the only counterexample to the otherwise strict size difference between ProtoN and JSON. By redefining `len` to encode 32-bit lengths, we can push the boundary even further, to over 4 million. As such, ProtoN's string encoding remains smaller than JSON's in virtually all cases.

Lists of size n require $n - 1$ commas and 1 closing bracket for a total of n bytes of overhead in JSON. In Proton they require 3 bits for a typecode and a `len` encoding, which will be smaller than n bytes by definition.

Objects of size n require $n - 1$ commas, n colons, and 2 braces for a total of $2n + 1$ bytes of overhead in JSON. In Proton they require 3 bits for a typecode and a `len` encoding, which will be smaller than n bytes by definition.

Because of the bitwise encoding inherent to ProtoN, its data format is much smaller than JSON's in nearly all cases.

4 IMPLEMENTATION AND EXPERIMENTS

Our proof of concept fully implements the ProtoN binary format in both Python and JavaScript. These two languages represent two of the most prominent frameworks used for server programming and handle a large portion of worldwide JSON traffic. We implement a unit testing suite and an experimental efficiency testing suite, both composed of handcrafted test cases as well as automatically generated large and small JSON files to measure correctness and size differential.

4.1 ProtoN Implementation

We implemented the ProtoN binary format in both Python and JavaScript. This choice of languages both allows us to empirically test the theoretical implementation of ProtoN and also provides a working implementation that can be used in Python and Node.js webservers together with JavaScript browser clients.

The primary difficulty in implementing ProtoN came from the fact that most programming languages, including Python and JavaScript, prefer to think of bytes as the smallest unit of data. This means that, in addition to implementing a type-inferring encoder and a corresponding decoder, a plethora of bitwise operations were required to achieve the correct results (which were at the level of bits).

The implementation of and encoder and decoder in both Python and JavaScript lent itself nicely to recursive functions. The encoders, for example, both recursively traverse the object being encoded and recursively write every subcomponent individually. Both language versions of the encoder and decoder implement the same binary format protocol specification, just in different languages.

The Python implementation requires Python 3.5+ and the JavaScript implementation requires ECMA6, meaning that the code will execute on any modern Python3 installation and any modern browser independent of platform.

Opportunities for future work include implementing the ProtoN specification in additional languages.

4.2 Benchmarking & Testing

We begin by generating a large set of JSON files which we use for both unit testing and space-efficiency benchmarking. These JSON files are of a randomly chosen length (up to some customizable, reasonable maximum) and contain random valid JSON data (i.e. random strings, integers, floats, booleans, null values, lists, and objects). In addition to these randomly generated files, we also gathered unit tests from open source JSON implementations[4].

These JSON files are then used as unit tests to check the equality of the decoded ProtoN encodings with the parsed JSON. We also test equality of implementation between the Python and JavaScript implementations using Node.js to run the JavaScript outside of a browser.

More interestingly, however, these JSON files are also used to benchmark the space-efficiency performance of ProtoN against JSON. For each JSON file, we compare the size of the resulting ProtoN binary encoding to the size of the equivalent UTF-8 encoded, whitespace-minified JSON representation. When running this benchmarking utility over 100 randomly generated JSON files and 58 example JSON files taken from open source JSON implementations, we see that ProtoN is 30.8% smaller than JSON, on average. That is, across all of the test files, ProtoN used 30.8% fewer bytes to encode all files.

We make this benchmarking utility available with the ProtoN source code, together with a random JSON generation utility, for any further desired testing.

We were further interested in how ProtoN benchmarked against JSON for specific primitive types. That is, we wondered what differences were found in the benchmarks when we restricted the test cases to a specific data type, such as floats. The results are shown in Figure 3.

[4]Tests were discovered within https://json.org/JSON˙checker/ and https://github.com/Julian/jsonschema

```
List(Null) ProtoN/JSON Size:  0.07558488302339532
Object(Null) ProtoN/JSON Size:  0.5644471594379963
List(String) ProtoN/JSON Size:  0.9498099137300775
Object(String) ProtoN/JSON Size:  0.9251850370074015
List(Int) ProtoN/JSON Size:  0.506427187885191
Object(Int) ProtoN/JSON Size:  0.6220235676554247
List(Float) ProtoN/JSON Size:  0.4552658349842052
Object(Float) ProtoN/JSON Size:  0.578967837280753
List(Bool) ProtoN/JSON Size:  0.09142130134496547
Object(Bool) ProtoN/JSON Size:  0.5526296432511953
List(List) ProtoN/JSON Size:  0.542485838053982
Object(List) ProtoN/JSON Size:  0.7659479686386315
List(Object) ProtoN/JSON Size:  0.542485838053982
Object(Object) ProtoN/JSON Size:  0.7664901732029077
List(Smallfloat) ProtoN/JSON Size:  0.9499133949191686
Object(Smallfloat) ProtoN/JSON Size:  0.8886060687252998
```

Fig. 3. Typed ProtoN/JSON Benchmarks

For each primitive, we constructed two tests: one which encoded 1,000 random values of that primitive in a list and one with 1,000 random values of that primitive paired with random string keys in an object. The results followed nicely from the theoretical results outlined previously. We found that the least efficient encoding (relative to JSON) were values which are encoded as strings (that is, Strings themselves and "small" floating point numbers which we represent as strings). These ProtoN values consumed 92-95% as much space in their JSON equivalents. We further found that the most efficient encoding (relative to JSON) were lists f `nulls` and boolean values, which consumed only 7-9% as much space against their JSON equivalents.

Our experimental results thus validate our theoretical basis that ProtoN is smaller than JSON in nearly every case. The results also allowed us to find an experimental average of 30.8% more efficient and find the optimal encoding types to be `null` and boolean values.

5 DISCUSSION AND CONCLUSION

In this work we have presented ProtoN, a new binary protocol for JSON-like data encoding. Unlike widely-used binary protocols such as Protocol Buffers, ProtoN does not require the prespecification of type and structural information, instead inferring it adaptively and on the fly. We demonstrated mathematically that in nearly all cases it is provably smaller than JSON, and in a comprehensive sequence of tests showed that it is 31% smaller than JSON on average.

One fascinating area of future work is message structure inference. One could imagine a protocol in which a message code is passed from sender A to receiver B along with the typecodes in the message (e.g., list of length 10 of int, float, etc.), as ProtoN currently does. Then on subsequent messages with that same message code the typecodes could be omitted and instead inferred by consulting a mapping from message code to typecode set by the receiver. This would even further reduce the overhead of a binary protocol, though would restrict subsequent messages to the same basic types.

JSON is one of those technologies whose ubiquity is matched only by its inefficiency. It is our hope that further work along these lines will result in a widely-adopted successor to JSON that is lighter and faster.

The source code for ProtoN is published at https://github.com/nickboucher/ProtoN.