none
Internet-Draft
Intended status: Informational
Expires: 6 February 2026

C. Partridge
T. O'Leary
Colorado State University

5 August 2025

Designing an Intermediate Data Format
draft-partridge-xdr-00

Abstract

Intermediate data formats (IDFs), also known as external data formats or data-interchange formats, are used to exchange data between networked applications. Example formats are JavaScript Object Notation (JSON) and Abstract Syntax Notation One (ASN.1). IDFs are ubiquitous in networking.

Despite their importance, there's remarkably little written guidance about how to actually design or architect an IDF. This guidance gap exists despite fifty years of experience designing and deploying IDFs, going back to the days of ARPANET. As a result, even recently developed IDFs have obvious flaws that could have been avoided.

The purpose of this memo is to provide basic basic guidance about how to design an IDF. This memo is NOT the product of any IETF or IRTF working group.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 6 February 2026.

Copyright Notice

Table of Contents

1. Terminology

The terms "byte" and "octet" are use interchangeably in this document to refer to 8-bit quantities.

Big- and little-endian are used as defined in [IEN137].

2. Introduction

Intermediate Data Formats (IDF) solve a critical problem in networking, namely how to transmit data units in a mutually intelligible manner between two networked applications that may be on different hardware platforms and have been implemented in different programming languages. The first external data formats were developed in the 1970s and they have been an integral part of data networking ever since. In the old OSI network stack, they were considered the core of the presentation layer. In the modern network stack they are considered part of the application layer.

Despite their importance in the network stack, not much has been written about how to design an IDF. Indeed, the most recent paper seems to be over 30 years old [PartridgeRose]. The motivation for this memo is to provide updated, albeit unofficial, guidance. This memo is NOT the product of any IETF or IRTF working group.

To illustrate the role of IDFs, consider the following example. Suppose you wished to exchange points on a map, aka objects containing integers for the X and Y coordinates, between two applications over the Internet. The sending application is written in Python and resides on a big-endian computer and the receiving application is written in C and resides on a little-endian computer. The Python application represents the X,Y coordinates as a class that incorporates two named integer attributes, "X" and "Y", of arbitrary size (Python integers can be arbitrarily large). The receiving computer probably represents the point as a C structure of two 32-bit integers (in little-endian format). Naively, if the Python application sought to send an instance of its class, it would presumably send two key:value pairs representing attributes X and Y, where the integers were in big-endian format. The receiving application, written in C, would likely expect data in its native format (so 8 bytes of binary data representing two integers in little-endian form) and view the Python output as inscrutable data.

IDFs solve this problem by having the sending application convert the data (in this case the X, Y coordinates) into a standard representation that is transmitted over the network. The network representation is then transmitted over the network to the receiving application, which converts from the network representation to the receiving application’s local representation.

Conceptually this is all the typical programmer needs to know about IDFs. When sending, put local data into an IDF format before sending and when receiving, extract from the IDF format into the local form.

But for protocol designers and more specifically, designers who need to create a custom IDF, a number of challenging issues reside under this conceptual simplicity. Indeed, one of the motivations for this memo is that one of the authors found himself reviewing multiple documents where the authors had concluded they needed a custom IDF and then demonstrated they did not understand how to design one.

The remainder of this memo is devoted to walking through the issues in designing an IDF and concludes with guidance to the IDF designer.

3. Brief IDF Examples

Before discussing the various issues, it is useful to sketch key features of some widely used IDFs. The purpose dual is (1) to give the reader a sense of the different styles of IDFs; and (2) describe the IDFs in enough detail that they can be used as examples; while deferring actual issues to later in this memo.

3.1. XDR

The eXternal Data Representation (XDR) was developed by Sun Microsystems in the 1980s to support applications, such as the Network File System, which were built on a remote procedure call API [RFC1014][RFC1832][RFC4506] [RFC1094][RFC3530][RFC7530].

Heavily influenced by the earlier Courier [Courier] intermediate format, XDR is a binary IDF; the data is sent over the network in a binary (non-text) format. XDR supports a range of data types including application-defined structures. Novel application data types are sent as opaque binary blobs.

XDR’s philosophy (reasonable for remote procedure call) was that the receiving application knows what data types it should receive. Accordingly, XDR typically sends only the binary representation of the data, with a length field, when required (e.g. for strings), and without any typing information.

3.2. ASN.1

Abstract Syntax Notation One (ASN.1) was developed by a team working on OSI email standards. Unlike XDR, they had to assume the receiving email application could receive previously unknown data types (e.g. in attachments from new applications) and had to be able to at least parse, and perhaps display, the new data types. (Similar design needs led OSI and Internet network management protocols to adopt ASN.1). ASN.1 also placed a premium on keeping the encodings small, out of concerns that a large encoding would harm transmission times (this was a time when email was often sent over 4Kbps links) and

affect email storage (again, a time when disks were small, with capacities measured in megabytes) [ASN1-design][ASN1-std].

ASN.1 is a binary IDF in which all information is sent as <type, length, value> triples. There are globally-defined type values and application-defined type values. Triples may be nested (e.g. for data structures), so the value field may itself contain triples. Type, length, and value are each encoded to use the minimum number of bytes.

3.3. JSON

JavaScript Object Notation (JSON) was developed to support communication with web-browsers. It is a text-based protocol that subsets the JavaScript language [RFC4627][RFC8259]

JSON supports a small set of data types along with two structured types: objects of key:value pairs, and lists. Novel application types typically get mapped to objects.

JSON transmits no explicit type or length information. Both are implicit in the text representation. So, for instance, "IETF" is a string of length 4.

JSON was designed to transmit finite objects and did not contemplate sending streams of objects (i.e. a TCP socket on which objects are sent as needed), with the result that there are multiple JSON streaming standards [RFC7464].

3.4. protobufs

Protocol Buffers (protobufs) were developed by Google as an internal IDF for language-agnostic serialization. They were later released open source in version 2 alongside other open source Google projects that depended on them.

Protobufs were designed to be similar to JSON or XML in that they support hierarchical structures for organizing data, essentially key-value maps like JSON objects. Also similar to JSON, new datatypes not included with protobufs are mapped to objects (messages). The major difference is that protobufs use a binary format on-the-wire that is certainly not human-readable, but is often more efficient.

Protobufs do not transmit type information. However, types are explicit in its custom language for defining the structure of the data. It can be viewed as a more rigorous version of XDR. Once the structure is defined, a compiler generates code in the user’s desired language for serialization and deserialization. This process allows

data definitions to be defined in one place even in multilingual projects. The generated code provides a convenient interface that, with the definition language, seeks to mitigate the readability problem of the binary format for the programmer. The serialized data, however, is definitely not human readable. Another common drawback is protobufs lack of support for multi-dimensional arrays.

Protobufs support both finite and streaming transmission.

4. Design Goals

The central challenge for the designer of an IDF is that there are multiple possible design goals and those goals are, in varying degree, in conflict. In this section, we step through the typical list of goals and some of the issues within each goal. We discuss the tensions between the goals in Section 7.

4.1. Processing costs

Programmers, quite reasonably, want their clients and servers to focus their computations on delivering the application service they implement, and NOT spending inordinate amounts of time converting into and out of intermediate formats. As a result, IDF designers often seek to make the computational cost of converting into and out of their intermediate form low.

In the 1990s, there was a substantial debate about whether binary or text representations were inherently computationally less demanding. While no definitive resolution was reached, most participants appear to have concluded that either representation can be architected to be computationally efficient. (Likely contributing to this conclusion was the realization that it was possible to build fast lexical analyzers for well-defined text formats).

That said, there continues to be a naive belief that a binary format should be more computationally efficient. There are two counter arguments. First, lexical analyzers can be extremely fast (see [Flex]). Second, Abstract Syntax Notation One (ASN.1), a binary format [ASN1-std], is generally viewed as the most computationally demanding IDF in widespread use.

4.2. Representation size

Particularly when transmitting large data objects, such as an array of a million integers, the time to transfer the data depends, in large part, on how big the intermediate data representation is. This observation leads to pressure to make the intermediate representation concise. Unfortunately, it is often unclear what the most concise representation is.

Consider the problems of transmitting integers.

When transmitting integers, it may seem obvious that a binary representation is more efficient. For instance, transmitting a 32-bit integer in binary requires only 4 octets, but may require up to 10 UTF-8 characters (or 13 characters if you include commas, e.g. 4,294,967,295). More generally, transmitting text representations of integers uses only 10 possible values in each octet, while a binary value uses every bit in each octet.

Except... In a large number of cases, the typical integer will be small. Zero is often the most common value transmitted and requires just a single octet of text. An array of a million zeros, with a space between each zero character, is 2MB. The same array sent as binary integers is twice as big. So binary versus text for concise representation of integers depends very much upon the data being sent.

The issue of representation size has become less important as network bandwidths have increased. On a 10Mbps Ethernet, which was the network for which many early IDFs were designed, sending an extra 10MB of data took an extra 9 seconds. On today’s gigabit networks, that 10MB is roughly an extra 80 milliseconds, which in many cases (though not all) is negligible for the application.

4.3. Generality and Extensibility

These are two tightly intertwined goals. The essential issues are (1) how broad a range of data objects does the the IDF’s core intermediate representation handle? and (2) how easy or painful is it for a programmer (or protocol designer) to extend the core representation to support additional data types?

Supporting a small set of data objects in the IDF’s core representation tends to simplify the IDF’s implementation, so there is some instinct to keep the list small. For instance, JSON [RFC7159] supports only floating point numbers, arrays, objects, strings and a small set of constants. XDR [RFC1014], which is more typical in the range of data objects, supports 32-bit integers,

unsigned 32-bit integers, 64-bit integers, unsigned 64-bit integers, floating point numbers, double precision floating point numbers, fixed and variable length opaque data, strings, fixed and variable length arrays, structures, unions, and voids.

Any data type or value not in the core representation must be transmitted as an application-defined extension. Extensions require programmers to develop their own standards for encoding and decoding the newly defined types. Different IDFs make supporting extensions easier or harder. Using JSON and XDR as the examples illustrates some issues.

JSON makes adding extensions easy: they are sent as JSON objects, which are simply collections of key:value pairs, where the value itself may be another JSON object. Converting into and out of key:value pairs is typically straightforward in any computing language. Thus, in general, JSON can support a small set of core types because it is easy to extend.

In contrast, XDR makes extensions painful. They are sent as opaque binary blobs of data, which forces the programmer to develop custom binary conversion routines, which is painful in some programming languages. As a result, XDR supports a wider range of core types including structured data types, in the hopes custom extensions will be rare.

The JSON example highlights that, done right, a small set of core data types can work well. It turns out JSON also illustrates a hazard of going small, namely leaving out an essential core value. JSON does not support the floating point constants Not-A-Number (NaN) and Infinity (Inf), which means if either value is possible, one needs to define a custom object. That’s a glaring omission, especially as NaN is often used in sensing applications to indicate a missing value.

4.4. Readability

One of the driving concerns for those who argued for text IDFs in the 1990s was the desire to make it easy to debug their protocols. Reading text data as it is sent in packets over the network is much easier than reading binary data.

Over time, we have come to realize the issue of readability has two parts. The first is whether the data stream be easily parsed by a human reader. The second is whether the semantics of the data stream can be inferred. Put more bluntly, can you (1) tell that the value 10 was transmitted? and (2) do you know what a 10 in specific position in the data stream means?

Parsing is the easier problem of the two. Both text and binary representations can indicate the type of the data. As discussed in Section 6, in text it is usually implicit, while in binary representations it may be explicit (ASN.1) or implicit (protobufs, XDR).

Semantics are tougher. Type fields or key:value representations can help, but work better in text than binary. For instance, "voltage:10", is easy to understand, while <application type=47><integer 10> (which is what an ASN.1 parser might display) leaves the reader wondering what application type 47 encodes. We visit this subject a bit more in Section Section 6 below.

5. Fundamental Design Rule: the MxN Rule

There is one fundamental design rule for IDFs that no designer should flout.

The MxN ("M by N") rule was first articulated by Padlipsky [Padlipsky]. Padlipsky sought to explain why the Network Virtual Terminal (NVT), developed in the early 1970s [RFC0377] and arguably the first IDF, was so successful.

The problem that the NVT solved was of connecting M different terminal types, via the Telnet protocol, to N hosts running different operating systems. Padlipsky observed that one wanted to avoid a protocol design in which adding a new, an M+1, terminal type forced all N existing hosts to be upgraded to support. The NVT solved this problem by providing a standard external terminal representation. Adding the new terminal type only required that software be written, solely on the new terminal’s host, to map the new terminal type to the NVT representation. Once the terminal type to NVT mapping was created, the terminal could connect to any of the existing N hosts.

One can see the MxN rule as a cautionary insight that a poorly designed protocol can undo Metcalfe’s law. Metcalfe’s Law observes that adding a new resource to the network makes all existing hosts on the network more valuable, because they benefit from the new resource. (Indeed, Metcalfe’s Law states the value of the network grows as the square of the number of users or resources on the network). Padlipsky’s MxN rules points out that if an existing hosts must be upgraded before the benefit of the new resource is realized, the benefit is sharply reduced.

The MxN rule has important implications for IDF design.

* First, the concept that "the receiver makes it right" is busted. The idea of "receiver makes it right" appeared early in the development of IDFs, most notably in the Network Data Representation [NDR], though elements of the concept linger in protobufs. The concept was that the sender would send (in binary) using its hardware's native data representation along with tags describing the data format and the receiver would then translate from the sender's format into the receiver's format. As observed in [PartridgeRose] it requires N sets of decoding rules in the receivers, and when a new hardware configuration arises (as it inevitably does), all receivers will need to implement an N+1 set of rules. ([PartridgeRose] notes that enabling a negotiation between two systems to see if their binary data formats are identical would work, but this adds a round-trip time to any data exchange, which may be undesirable.)

* Second, the IDF has to get the core set of data types right the first time. once an IDF is deployed, it is extremely hard to add new core data types as this requires updating all deployed systems. This makes IDFs one of the notable exceptions to Brooks' rule to "plan to throw one away" [Brooks]. It also tends to cause IDFs to age poorly. A particular issue is floating point representations. Over time, floating point numbers have tended to get larger. If an IDF selects a specific floating point format as its core floating point type (and we will suggest in Section 8.2 that IDFs should no longer choose a particular floating point format), it is inevitable that some years in the future, that core type will be too small.

6. Tags and Types

Every object in an IDF has a type and a meaning (semantics). The two properties are often confused and there is no consensus about how to handle them.

The problem is most clearly understood from the receiver's perspective. A stream of data is arriving and the receiver needs to interpret the data. At minimum, the receiver needs to know the intended data type, so it can correctly represent the data on its machine.

Text-based IDFs typically use implicit typing. Thus, 10 is a decimal integer, "10" is a string, and 0x10 is a hexidecimal integer. Explicit typing only becomes an issue for structured data, where you may wish some indication that the object containing two integers is a point on a map and not the velocity and mass of a wheeled vehicle.

In contrast binary IDFs need explicit tags to indicate types at least some of the time. If the application knows the structure of the binary stream, e.g. it is a sequence of 32-bit integers, then no type tag is required. If, on the other hand, the application may not know what is coming, or you want the ability to extend the stream to use either 32-bit or 64-bit integers, then some sort of tagging is required to interpret what is otherwise a undifferentiated sequence of bytes.

Whether to have type tags all the time or just when needed depends on whether the binary IDF intends to support the receipt of unexpected data. Some binary IDFs, notably XDR, explicitly state they do not support the receipt of unexpected data. In contrast, for generality, ASN.1 tags everything.

Semantic tags are a tougher problem. To illustrate the issue, consider an application that is receiving temperature data in the form of floating point numbers from a suite of sensors. Suppose the original set of sensors all returned data in Fahrenheit, but a new set of sensors returns the data in Celsius. Converting to support the new sensors requires either negotiating the temperature type when the sensor is first attached (e.g. via a TCP connection) to the application, or reworking the IDF to insert a temperature scale indicator such as "F", "C" (and since you’re doing it "K") into the the data stream.

None of the major IDFs except for ASN.1 support semantic tagging. And ASN.1’s approach is imperfect: it uses an application specific tag. As a result, if our temperature sensing system uses ASN.1, a receiving application will recognize that the temperature data is arriving with a new tag (which is great -- there is no chance of temperatures being misunderstood), but, until new code is written, the application will be unable to use the data.

For an IDF designer, the key message is that tagging is not a special case. Rather the decision of how to tag represents a decision about how you see the IDF being used and its ability to extend to new uses in the future.

7. IDF Design Tradeoffs

There are a few tradeoffs that any IDF designer should consider. In this section, we walk through those tradeoffs.

7.1. Optimizing representation harms the other goals

The biggest design challenge is trying to optimize the size of the IDF data objects (Section 4.2). The harder you push to make the representation as small as possible, the greater the impact on processing costs, the harder the representation typically is to read, and there's a tendency to avoid type values (which consume space) even when it causes generality to suffer.

ASN.1 is an excellent example of the challenges. The ASN.1 designers sought to both create a fully general IDF, in which every object has both a type and length field, AND minimize the amount of space to represent any object ([ASN1-design]). For instance, integers are compressed to the minimum number of bytes to represent the value. Type fields are minimized; when type values exceed a byte, there is an extension bit that says the field extends to the next byte. The result is an IDF that is elegant in its design, but a computational nightmare.

7.2. Core data types vs. generality vs. computation

Historically, there has been a temptation to define a relatively large set of core data types (XDR has 18) because it reduces the pressure to support user-defined types. Indeed, with a large set of core types, the extension mechanisms for user-defined types can be made simpler (e.g. XDR simply allows for binary blobs).

But the larger set of core types often increases the computational costs for handling the IDF. (Note: the increased cost is typically in the form of the code footprint and thus cache misses. The actual code is often some form of switch or match dispatch based on a type field.)

Arguably the greatest contribution of JSON has been to solve this trilemma. JSON has a small set of core types combined with a powerful and simple extension mechanism. Extensions are simply sets (objects in JSON terminology) of key:value pairs. Sets of key:value pairs are easy to parse and easy to map to internal data representations (sets, structures, objects, etc.). Their only downside is that key:value pairs tend have a larger representation footprint, and minimizing representation size by using short key names such as "xy9" hurts readability.

Note that JSON, by using easy to read text and key:value pairs, also goes a long way towards solving the semantics problems without a type field (per Section 6).

8. Lessons Learned

8.1. The IDF representation is authoritative

There is a tendency when creating a new networked application to view the host data structure as authoritative. For instance, one might say "we chose to represent the C data structure this way in JSON."

That’s the WRONG way to think about IDF data formats. Rather, in a networked application, the IDF data format is the authoritative data structure and the host/language representations are interpretations of the authoritative definition. Indeed, for this reason, many IDFs have carefully defined data definition languages.

Protobufs gets this lesson right, with a mechanism to compile the authoritative IDF format into encoders and decoders for a range of languages.

8.2. Floating point is hard

Handling floating point numbers have historically been a hard problem. There are two issues:

* Precision. How many bits of floating point precision are represented.

* What to do about NaN (Not-A-Number) and Infinity. Floating point numbers may have non-numeric values and they are useful in a number of cases such as as an indication that an expected value is missing.

At the risk of getting in front of current consensus, we suggest the following approach going forward:

* Future IDFs should not specify the floating point precision (indeed, JSON waffles towards this approach now). The existence of high-quality arbitrary-precision floating point libraries for most computing languages means there is no reason to pick a particular precision.

* NaN and Inf must be supported.

8.3. Integer types tend to proliferate

In binary IDFs, there is a tendency to create lots of integer types. Think of the cross-product of signed and unsigned and a range of widths from 8 bits to 128 bits.

ASN.1 is the exception to this insight, having defined an arbitrarily large integer type, in which the actual integer value is transmitted in the minimum number of bytes. But this flexibility comes with increased processing costs.

Text-based IDFs neatly skirt this problem. The integer, of whatever size, is transmitted as a readable string. The only concern is that the receiving application is unable to handle very large integers, an increasingly unlikely issue given the proliferation of arbitrary precision integer math packages.

8.4. Define the streaming format

IDFs are routinely used to implement streams of multiple data objects. This means an IDF needs a rule that defines when one object ends and the next starts.

In many IDFs this is simple. The new object starts with the byte after the end of the prior object. This is the approach, for instance, in ASN.1 and XDR. In fact, this approach is so simple that a designer may fail to realize that rule for delimiting objects is required.

A failure to define how objects are delimited can lead to compatibility issues. JSON failed to define an object delimiter, with the result that there are multiple, incompatible, ways to stream JSON. Furthermore, each streaming approach has limitations [RFC7464].

Any new IDF should define its streaming rules.

9. Security Considerations

IDFs sit at a critical security boundary. The IDF code at the receiver is parsing data received from the network. This parsing process has frequently been a source of over-the-network attacks, including stack overflow attacks and SQL injection attacks.

That concern being acknowledged, as of the time this memo was written, there are no accepted best practices for designing IDFs to be safer from these styles of attacks.

10. Concluding Guidance

If you were asked tomorrow to design a next generation IDF, intended for general use, we offer the following guidance:

* Use a text-based format. The size difference between text and binary representations is sufficiently application and data dependent, that it is not a good principle for making a choice. As a result, the benefit of being able to read the data in transit is the greater benefit (per Section 4.2 and Section 4.4). Text formats also do a better job of conveying semantics (Section 6).

* Minimize the number of types. Define integers as of arbitrary size, to avoid the trap of many integer types (per Section 8.3).

* Make floating point simple. Allow arbitrarily large floats and make sure to support NaN and Inf (per Section 8.2).

* Have a simple extension mechanism such as JSON’s key:value object.

* Define the streaming format (per Section 8.4).

This guidance still leaves considerable room for innovation. For instance, one might push to make the text representation concise and do things such as (1) transmit all numbers (floats and integers) in hexidecimal (or a larger radix) and (2) have a way to minimize key names, without substantially compromising semantics. Furthermore, designing a security-focused IDF is a wide-open problem.

11. Acknowledgements

12. IANA Consideration

This document has no IANA actions.

13. Informative References

[ASN1-design] Deutsch, D., Resnick, R., and J. Vittal, "Specification of a Draft Message Format Standard (BBN Report 4486)", September 1980.

[ASN1-std] ITU-T, "ITU-T X.690 (02/201) - Information technology - ASN.1 encoding rules- Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", 1 February 2021.

[Brooks] Brooks, F., "The Mythical Man-Month", ISBN 978-0201835953, 1975.

[Courier] Corporation, X., "Courier - The Remote Procedure Call Protocol (XSIS 038112)", December 1981.

[Flex] Levine, J. R., "flex and bison - Text Processing Tools", ISBN 978-0596155971, 22 September 2009.

[IEN137] Cohen, D., "IEN137 - On Holy Wars and a Plea for Peace", 1 April 1980.

[NDR] Dineen, T. H., Leach, P. J., Mishkin, N. W., Pato, J. N., and G. L. Wyant, "The Network Computing Architecture and System - An Environment for Developing Distributed Applications", USENIX Conference Proceedings , 1987.

[Padlipsky] Padlipsky, M. A., "A Perspective on the ARPANET Reference Model", Proceedings of IEEE INFOCOM , 1983.

[PartridgeRose] Partridge, C. and M. T. Rose, "A Comparison of External Data Formats", Proceedings IFIP Intl. Conf. on Message Handling Systems and Distributed Applications , 1988.

[RFC0377] Braden, R., "Using TSO via ARPA Network Virtual Terminal", RFC 377, DOI 10.17487/RFC0377, August 1972, <https://www.rfc-editor.org/info/rfc377>.

[RFC1014] Sun Microsystems, "XDR: External Data Representation standard", RFC 1014, DOI 10.17487/RFC1014, June 1987, <https://www.rfc-editor.org/info/rfc1014>.

[RFC1094] Nowicki, B., "NFS: Network File System Protocol specification", RFC 1094, DOI 10.17487/RFC1094, March 1989, <https://www.rfc-editor.org/info/rfc1094>.

[RFC1832] Srinivasan, R., "XDR: External Data Representation Standard", RFC 1832, DOI 10.17487/RFC1832, August 1995, <https://www.rfc-editor.org/info/rfc1832>.

[RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and D. Noveck, "Network File System (NFS) version 4 Protocol", RFC 3530, DOI 10.17487/RFC3530, April 2003, <https://www.rfc-editor.org/info/rfc3530>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <https://www.rfc-editor.org/info/rfc4506>.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, DOI 10.17487/RFC4627, July 2006, <https://www.rfc-editor.org/info/rfc4627>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <https://www.rfc-editor.org/info/rfc7159>.

[RFC7464] Williams, N., "JavaScript Object Notation (JSON) Text Sequences", RFC 7464, DOI 10.17487/RFC7464, February 2015, <https://www.rfc-editor.org/info/rfc7464>.

[RFC7530] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", RFC 7530, DOI 10.17487/RFC7530, March 2015, <https://www.rfc-editor.org/info/rfc7530>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

Authors' Addresses

Craig Partridge
Colorado State University
Department of Computer Science
Fort Collins, Colorado, 80528
United States of America
Email: craig.partridge@colostate.edu

Tyson O'Leary
Colorado State University
Department of Computer Science
Fort Collins, Colorado, 80528
United States of America
Email: tyson.oleary@colostate.edu