A Comparison of External Data Formats

Craig Partridge
BBN Systems and Technologies Corporation/Harvard University

and

Marshall T. Rose
The Wollongong Group, Inc.

Abstract

A comparison and analysis of three external data formats: Abstract Syntax Notation One (ASN.1), Sun Microsystems' External Data Representation (XDR) language and Apollo Computer's Network Data Representation (NDR) language is presented. The performance, ease of implementation and use in specifications of the three schemes is examined.

1. Introduction

External data formats are an essential ingredient in the development of loosely-coupled distributed systems. Usually considered part of the presentation layer, external data formats allow heterogeneous systems to exchange information in a mutually intelligible manner. As networks have become more common, interest in external data formats has correspondingly increased.

Many people believe that external data formats are a recent invention. In fact, they are a very old idea, dating to the earliest days of networking. They were first considered in the earliest days of the ARPANET [2] and considerable work on external data formats was done during the late 1970's by the National Software Works project [14], at Bolt Beranek and Newman for the National Bureau of Standards [4,5], and most notably, at Xerox, where Courier, the first popular external data representation, was developed [23].

Despite the long history of work with external data formats, to the authors' knowledge, no detailed comparison of the different schemes and their specification languages has been presented in the open literature. (DeSchon [3] has done a survey of the range of data types supported by various encoding schemes). Indeed, one of the authors was motivated to prepare this paper after going through the painful process of trying to determine which format was appropriate for use in specifying and implementing an experimental network management system. (The other author, being well–versed in international realities, considers only ASN.1 for any projects.)

We examine three well-known data formatting schemes here: Abstract Syntax Notation One (ASN.1), Sun Microsystems' External Data Representation (XDR) language, and Apollo Computer's Network Data Representation (NDR) language. After introducing each of the schemes, we compare their performance, ease of implementation, and how easy it is to specify their use. Some of this comparison will necessarily be a matter of style, but as others in the field have already pointed out, style matters; schemes that are cumbersome or uncomfortable to employ are less suitable than schemes that are easy to understand and use.

2. Abstract Syntax Notation One (ASN.1)

ASN.1 is the ISO standard for describing and encoding data structures. There are two parts to ASN.1: a language for specifying the data types used in an exchange [7], and rules for how the data types

Mr. Partridge's work was supported by a subcontract with the University Corporation for Atmospheric Research, which operates under contract with the U.S. National Science Foundation. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the UCAR or the NSF.

expressed in that language are actually represented in binary form [8]. The standards are derived from the CCITT Recommendation X.409, which is the encoding scheme used in the 1984 X.400 Recommendations on Message Handling Systems. (The new 1988 CCITT Recommendations which correspond to the ASN.1 standard are X.208 and X.209.) X.409, in turn was heavily influenced by message format standards developed for the U.S. National Bureau of Standards [4,5] and the Courier data encoding scheme.

The encoding scheme supports both simple, or *primitive*, types, such as integers and booleans, and structured, or *constructed*, types such as sets and sequences of other types. For example, to represent an array of integers, ASN.1 encoded integers are placed within a larger ASN.1 data structure, such as a **Sequence**, that represents an array.

The encoding rules for ASN.1 use a *tagged* scheme: a certain amount of typing information is prepended to the front of each data item. In ASN.1, the prefatory information is simply an identifier followed by a length field, which indicates the size of the data which follows. (This regularity or consistency of formatting, in which all data is represented in triples of a standard form, is considered an important innovation of the NBS/X.409/ASN.1 data encoding scheme). The representation is shown in Figure #1.

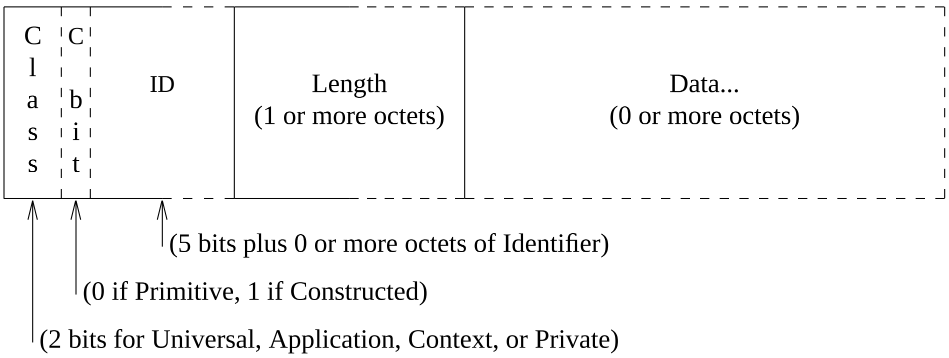

Figure #1

The data field of a constructed type may itself contain other complete ASN.1 encoded values, and often does. (E.g., the case of an array of integers mentioned above).

In some cases, the actual size of a data field of a constructed type may not be known until after it has been generated. In these situations, ASN.1 permits the use of an indefinite length value (a code that says the length is unknown) and the data is terminated by a special delimiter, called an End of Contents (EOC) marker. The use of the indefinite length mechanism is particularly useful when ‘‘streaming’’ ASN.1 objects, and can permit some elegant software paradigms to be employed. Unfortunately, the current OSI end–to–end protocols are all packet-, not stream-oriented, and do not easily avail themselves to this architecture. However, other end–to–end protocols, such as the Transmission Control Protocol (TCP) [1], are stream–oriented. For example, a lightweight presentation protocol for use in TCP–based internetworks has been specified [16], which serializes ASN.1 objects directly onto a TCP stream.

The identifier portion of the encoding is divided into three fields: a class field, a constructor field, and a type field. The type field is a number that is assumed to have meaning in some context. In other words, the receiver of the encoded data is expected to be able to use the type field to correctly interpret the data. The class field is a two bit value that indicates the context in which the type field is defined. The available type classes are Universal (predefined data types), Application–wide (meaningful within a given application context), context–specific (meaningful only in the context of the data structure or stream in which it is found), and private (meaningful when a bilateral agreement is in place).

Both the typing information and the data values are encoded in an extensible fashion (variable length), but in the fewest number of octets required to represent the information. For example, an integer containing a value which can be represented in one octet has its data portion represented in one octet (in contrast, other schemes might encode integers using a fixed width, e.g., 32–bits or 4 octets). The scheme used by the ASN.1 encoding rules are designed to minimize the use of network bandwidth, while not restricting the types of data that can be sent.

Beyond having a powerful encoding scheme, ASN.1 also is a very rich description language. ASN.1 is useful for defining the structures exchanged by all application–layer protocols; indeed, it is currently used in the definition of all OSI applications.

In addition to stating what data types are to be exchanged, the description language can also be used to define new types, including ennumerated types with a limited set of legal values. The language also defines a macro facility that permits specifiers to extend the language grammar should they find it useful. A complete description of the language is beyond the scope of this paper, but we present a simple example here to give the flavor of the language.

Figure #2 shows the declaration of a record that stores the x and y coordinates of a point on a graph:

```
GraphPoint ::= [APPLICATION 0] IMPLICIT SEQUENCE  -- point on a graph
  {
        xCoord     INTEGER,  -- x coordinate
        yCoord     INTEGER   -- y coordinate
  }
```

Figure #2

This declaration says that *GraphPoint* is sequence of two **Integers**, the first of which has the label *xCoord*, the second *yCoord*. The **Implicit** keyword means that the type of the value (e.g., **Sequence**) can be inferred from the tag (an application-wide tag of 0) and need not be encoded in the data field. If the **Implicit** clause had been left out, then values of this type would be doubly-encoded, with the first encoding have a tag of *GraphPoint* tag, and a data field of an ASN.1 **Sequence**, which contained the two integer values.

If we were to encode the *GraphPoint* structure for the point {x 5, y 4} we would get the following hexadecimal string (assuming we used definite lengths):

```
60 06          ... Application type 0, with length of 6
  01 01 05     ... xCoord with value 5
  01 01 04     ... yCoord with value 4
```

Figure #3

One final note before beginning the analysis of ASN.1. Some readers may wonder, given that ASN.1 is the ISO standard, whether there is any point in even thinking about using other data encoding standards. The answer is that other encoding standards are still of interest, for several reasons. First, the existence of an Abstract Syntax Notation *One* implies that at some point there might be an Abstract Syntax Notation *Two*, should there be reason to believe that an alternative notation or encoding rules are useful. (Although the ASN.1 standard claims to distinguish between the description language and the encoding rules, the inclusion of the **Implicit** clause in the description language is perhaps a potential intrusion from the encoding rules.) Furthermore, if there are reasons to believe that other specification or encoding schemes are superior to ASN.1 either in general, or for specific applications, there is some chance that they too someday become standards.

2.1. Implementation Approaches

The authors are aware of at least three approaches to writing programs that encode and decode ASN.1 data objects. We call these three approaches *compiling*, *partially decomposing*, and *hard coding*.

2.1.1. Compiling (the ISODE approach)

The ISO Development Environment (ISODE) is a software package useful for developing OSI–based applications [17]. As part of its infrastructure, the ISODE contains a set of ASN.1 tools and libraries. The approach taken by the ISODE is to optimize flexibility in the manipulation of ASN.1 objects, at the expense of space considerations. With respect to the encoding and decoding of ASN.1 data objects, the ISODE approach is straight–forward: ASN.1 objects are represented using an ‘‘internal form’’ called a *presentation element*. These elements are very similar in nature to the stream of lexemes produced by the front–end of a language compiler (i.e., a ‘‘parse-tree’’ of objects). At the lowest level, library routines are provided which convert a presentation element into machine–dependent data types such as integers and strings in the *C* programming language.

Above this, an ASN.1 compiler–compiler called *pepy* is employed. The *pepy* program reads an ASN.1 specification and produces: an encoder, which takes machine–dependent data types and produces a collection of presentation elements; a decoder, which performs the inverse operation; and, a pretty–printer, which can be used to display presentation elements. The *pepy* program produces these compilers by reading an augmented ASN.1 specification. This augmented specification contains the directives necessary to tell pepy where and how presentation elements should be mapped into/from various data objects.

However, providing these augmentations by hand is tedious and error–prone; so, at the highest–level, another compiler, the *posy* program is used as a front–end to *pepy*. The *posy* program reads an unaugmented ASN.1 specification and produces two sets of definitions: first, a set of *C* structures definitions which correspond to the ASN.1 types in the specification; and, second, the augmented ASN.1 specification for direct use with *pepy*.

This approach has the advantage that applications programmers need only concern themselves with the *C* language structures. All of the aspects and details of ASN.1 are completely hidden from the user. For example, if the ASN.1 fragment in Figure #2 where defined in the module ‘‘GRAPH’’, *posy* would define the structure shown in Figure #4 for the user:

```
struct type_GRAPH_GraphPoint {
        int xCoord;
        int yCoord;
};
```

Figure #4

In addition, *posy* would generate the augmented ASN.1 specification as input to *pepy*. Space limitations prevent us from illustrating this here. For a more detailed explanation of the toolkit provided by the ISODE consult [9,18].

The disadvantages of the ISODE approach are two-fold: first, although presentation elements are flexible by design, all flavors of them are currently core resident — that is, the entire top–level ASN.1 object must be in memory when an encoder, decoder, or pretty–printer is used; and, second, the compilers produced by *pepy* are recursive–descent and not table–driven. This yields rather large executable programs. Neither of these two problems are inherent to the design of the ISODE’s ASN.1 facilities however, both could be fixed (by the use of *virtual* presentation elements for the former case, and by using a table–driven approach for the latter case).

An implementation approach similar to that of ISODE was taken in the NBS ASN.1 implementation [13].

2.1.2. Partially Decomposing

Partially decomposing ASN.1 objects was an idea developed in an implementation of the High-Level Entity Management System (HEMS) [11,12]. HEMS is a research system designed to support network management on very large internetworks. To request management information HEMS uses a simple postfix database query language. This language is binary encoded in ASN.1 and operates on an ASN.1 encoded database (called a management information base).

Because so much of HEMS works with ASN.1 data structures, it can be convenient to implement HEMS using ASN.1-like data structures internally. This observation led to the idea of partial decomposition.

An application using partial decomposition chooses an internal format for its data items that makes it easy to convert them into and out of ASN.1. In the HEMS implementation, all data items that may be

ASN.1 encoded are stored in tree-shaped structures. Each node in a tree contains ASN.1 tag information and either a data value, or pointers to children that store additional ASN.1 values. So, for example, a structured type with four fields is represented as a parent node with four tagged children (one for each field). Generating ASN.1 from this format is easy; a simple pre-order tree traversal on the data structure can be used to generate the ASN.1 encoding. Converting ASN.1 into the tree format is equally straightforward.

The appeal of this approach is that it suggests a way to implement a set of application-independent library routines to manipulate ASN.1 structures. The obvious disadvantage is that the intermediate data structures are less memory efficient than using the host's native formats.

2.1.3. Hard Coding

A third approach to processing ASN.1 is to hand craft routines to manipulate only those ASN.1 structures that your application will use. For example, given an application that only requests and sends small strings, one can implement very efficient routines that can only generate and send short ASN.1 strings. (The efficiencies come from optimizations such as using templates for tags, making assumptions about size of length fields, etc.)

The goal of hard coding is to achieve greater efficiency (both in code space and execution time) in exchange for sacrificing the greater generality and easier maintenance of compilation or partial decomposition. Informal reports [19] suggest that for applications using a modest number of ASN.1 types, hard coding improves performance by a factor of two to four and reduces code size by an equal amount. From the authors' perspective that is a useful but not dramatic improvement. It is about the same performance improvement that one would get by using assembly language instead of a high-level language. That suggests that hard coding may only be useful in those cases analogous to instances where people recommend use of assembly language: typically critical paths in key subsystems.

2.2. Performance

The speed and compiled code size advantages of the hard coding approach leads us nicely into the issue of performance.

Some critics of ASN.1 have argued that tagging data is an unnecessary expense. At the risk of paraphrasing several different arguments into one, the objection is that the receiving application already knows the type of the next data item (presumably because the protocol specifies what values are sent, and in what order) and thus it is unnecessary overhead on the network and the sender to require the inclusion of a tag that the receiver does not need to examine. This line of reasoning is often extended to contend that the networking community should be developing time-efficient external representations of data that do not require tag and length fields.

There is some justification to these arguments. Tags are unnecessary in many remote procedure call (RPC) schemes where stub routines at both ends of a connection ‘‘know’’ exactly what data types will be exchanged. Furthermore, ASN.1's emphasis on minimum encoding lengths can be a nuisance. It is often easier, and sometimes yields more efficient code, to implement routines that convert to and from a fixed-size external representation, instead of routines that convert to and from a variable-size representation.

But speaking in ASN.1's defense, it is also true that tags can be useful. Common data structures like unions, optional values, and variant records all require some sort of explicit or implicit tagging to be intelligible to the receiver. It is also perfectly possible for applications that ‘‘know’’ what data is coming to simply ignore the one-octet tag field. In addition, experience with encoding and decoding ASN.1 suggests that the emphasis on minimum-sized items, while a nuisance, has only a small effect on performance (enough that one might notice, but little enough that one usually doesn't care). Debugging miscommunications between ASN.1 peers is also easier than with untagged schemes, because it is possible to interpret the ASN.1 data stream by reading the tags. Finally, systems that do not use tagging usually do so at the expense of flexibility in the encoding scheme. (See, for example, the discussion of the limits on the size of XDR encoded integers in section 3.2).

2.3. Specification Language

ASN.1 is usually recognized as the most expressive of the data specification languages. That is a mixed blessing.

Most data structures can be specified in ASN.1, and the language gives the specifier several choices about how a structure should be defined.

Unfortunately, the richness of the language also makes it easy to make errors in or misunderstand specifications that use it. Experience suggests that a large number of specifications contain passages of incorrect ASN.1. ASN.1 compilers make it reasonably easy to detect such errors (although the macro facility makes most compilers less effective) but not all specifiers currently have access to a compiler. Furthermore, a large number of readers of ASN.1 apparently find it confusing to use.

However there is also a small group of people who think ASN.1 is not expressive enough. Their concern is that many readers of ASN.1 mistake the partial semantics that ASN.1 data description provides for the full semantics of a specification. An example may help here. Suppose we have a protocol for retrieving input packet counts from different machines on a network. In ASN.1 we can easily define an **InputPacketCounter** type to contain this value. It is important to recognize that this does not define what an **InputPacketCounter** really is. In particular, we have not defined what types of packets we are counting, nor where we count them in the system (in the interface or the operating system?). The problem is that readers can (and do) mistake the listing of ASN.1 types as a complete specification, instead of a data definition.

Furthermore, there are a few types you cannot specify in ASN.1. For example, arrays of a fixed length cannot be defined. Examples of when this might be useful include defining fields to store the two-letter ISO country abbreviations, or a nine-digit U.S. zip code.

3. External Data Representation (XDR)

The XDR [21,22] was developed around 1984 by Sun Microsystems, Inc., to support their distributed file system, NFS.[1] Since then, the XDR has become a popular data encoding scheme and has been used in several networked systems. The design of the XDR was heavily influenced by Xerox's Courier system [23] and the comments here about the XDR generally apply to Courier as well.

The XDR is usually employed with Sun Microsystems' RPC (remote procedure call) protocol to implement lightweight distributed applications (of which NFS is an example). It is beyond the charter of this paper to compare Sun/RPC to the OSI Remote Operations Service (ROS).

Except when absolutely necessary, the XDR does not use tag or length fields. Most data is sent in a standard format without any qualifying prefix. To avoid alignment problems during data conversion at both ends, all data items, and most prefixes and subfields with data items are padded to four-octet boundaries. The XDR refers to these four-octet units as *blocks*.

The XDR supports fifteen basic types including both simple types such as integers and booleans and constructed types such as arrays and structures. There is also an *opaque* type that permits consenting applications to exchange special purpose data types. The XDR does not support bit strings, although this is noted as an obvious extension.

The XDR language is designed to make data type definitions look very similar to *C* data declarations. So, for example, in the XDR, the *GraphPoint* example used in Figure #2 might be expressed in this form:

```
typedef struct {     /* point on a graph */
        integer xCoord;  /* x coordinate */
        integer yCoord;  /* y coordinate */
} GraphPoint;
```

Figure #5

[1] NFS is a trademark of Sun Microsystems Inc.

The key advantage of this format is that it expresses the data in a form that programmers are familiar with. It also makes it very easy to define RPC interfaces. For example, one might define a routine to send a *GraphPoint* to another system as:

```
SEND_GRAPH_PT (gp) returns (success)
   GraphPoint gp; /* the point to be sent */
   boolean success;        /* whether the point was sent */
```

Figure #6

Indeed, SUN has defined a remote-procedure call definition language, an extension of the XDR language, that can be used to define these sorts of procedure stubs.

If we encode the *GraphPoint* structure using the XDR, we get a very different data encoding from that of ASN.1 (although they both turn out to be 8 octets long). In the XDR there is no tag information, just the two, four-octet, integer values:

```
00 00 00 05        ... xCoord with value 5
00 00 00 04        ... yCoord with value 4
```

Figure #7

Notice that the receiver must know that it is receiving a *GraphPoint* to be able to correctly interpret the data.

One should not interpret the XDR view that the receiver knows what is coming as suggesting that the XDR is functionally less versatile than ASN.1. In fact, from the encoding perspective, XDR can be shown to be functionally equivalent to ASN.1. Consider:

```
typedef struct {     /* XDR type to hold ASN.1 types */
        integer tag;         /* ASN.1 tag */
        integer length;      /* ASN.1 length */
        opaque data;         /* ASN.1 data */
}
```

Figure #8

which can represent any ASN.1 encoding.

3.1. Implementation Approaches

Most people implement the XDR using *stub compilers*, programs designed to generate data conversion routines for use with remote procedure calls. Stub compilers typically read a data definition language and generate both data declarations in the host language (e.g., include files for *C*) and routines that convert the host data structures to and from XDR formats. An example of such a compiler is SUN's *rpcgen* [20].

Another approach to the stub compiler idea, developed in the *rpcc* program [15], is to implement a pre-processor for a particular programming language (typically *C*) that generates stub routines from specially annotated code. The idea is that one implements and tests a program on one's own machine, and then runs it through the preprocessor to generate code that uses network RPC instead of local subroutines.

3.2. Performance

The designers of the XDR tried to minimize conversion costs by assuming that the receiver knew what was being sent to it, which eliminates the need for most tags, and by using a combination of fixed data sizes and a blocking factor (all data must be padded to a block boundary) to make the conversion easier in both directions. The two optimizations are related but distinct, and we look at each issue separately.

Assuming that the receiver knows what data is being sent to it is a reasonable strategy for many applications. In most data exchanges, the receiver knows full well what is being transferred. And by making this assumption, the XDR is able to get rid of the type tag on each data item. ASN.1 usually requires only one octet to contain the type tag, so getting rid of the type tag is not a huge performance gain but it makes some impact. It is one octet less to generate, move and process per datum, and if the number of data exchanged is large (for example, a long list of integers) this may give a performance gain.

Using fixed-size types and a blocking factor results in larger performance gains: first, using fixed–size types, allows us to get rid of the length fields for most types. This achieves a gain in performance because it both reduces the size required to represent a datum and makes it easier to optimize the data conversions. Second, using a blocking factor results in word–aligned encodings which will almost always make data conversion at both the sending and receiving ends easier.

An example helps to illustrate these points. Imagine we are sending an integer using ASN.1 and the XDR. In the XDR, this integer is always sent as an aligned 4-octet quantity in ‘‘big-endian’’ byte-order. Most machines can convert integers into and out of this format in one to five instructions. In ASN.1, the size of the encoded integer will range from 3 to 6 octets of which 1 to 4 octets are the integer value (for the moment assume that we only want to send 32-bit integers). The ASN.1 receiver must have code that knows how to interpret the tag and length fields and also how to convert unaligned 1-octet, 2-octet, 3-octet and 4-octet integers into and out of the host’s internal integer representation.

The major problem with fixed sizes is that they limit the size of the values you can send. Taking our example above, if we wanted to send 72-bit integers, the XDR does not have an integer data type that can hold them. We would have to use the opaque type to send the integer. In ASN.1, there is no restriction on the integer size. Since the notion of the appropriate data size tends to change quickly (the XDR prefers 32-bit quantities, whereas Courier, which is only a few years older, prefers 16-bit quantities), one has to consider fixed sizes a serious limitation on the long-term general utility of the data format.

3.3. Specification Language

The XDR data description language is designed to look very similar to the data definitions of the *C* programming language. In fact, they look so similar that it has been reported that readers sometimes get them confused, and mistake the XDR language for *C*.

The primary advantage of the XDR language is the similarity to existing programming languages. Programmers reading XDR definitions are likely to find the constructs familiar, and thus easy to understand.

The major disadvantage is that it is not designed to be a complete specification language. Except for type definitions, the XDR has none of the higher-level specification constructs such as module definitions, inclusion mechanisms and macro facilities that are available in ASN.1. (Some XDR compilers, most notably SUN’s, actually do supports macro and a form of inclusion but the XDR standard does not require these features to be supported). The point to be made here is that the XDR language was designed primarily to describe data exchanged through remote procedure calls, and not for the more complex specifications such as X.400 that ASN.1 was designed to handle.

4. Network Data Representation (NDR)

The NDR [6] has been recently developed by Apollo Computer, Inc., as part of their Network Computing Architecture (NCA), an object–oriented framework for developing distributed applications. As with the XDR, the NDR is part of a larger picture: used in conjunction with Apollo Computer’s remote procedure call protocol (NCA/RPC), the NDR can be used to build distributed applications. Again, it is beyond the charter of this paper to compare NCA/RPC to OSI’s Remote Operations Service (ROS) or SUN RPC.

However, the NDR is only a set of rules for encoding data. The actual definition of the types which the NDR encodes are specified using Apollo Computer’s Network Interface Definition Language (NIDL). Unlike the XDR, which is both an implicit specification language and a set of encoding rules, one may view the NIDL as corresponding roughly to the ASN.1 specification language, while the NDR as corresponding to the ASN.1 encoding rules.

The XDR and the NIDL/NDR are conceptually very similar, and really differ only in the details (sigh). There are however a few key differences which warrant our attention.

The NIDL supports around 18 basic types, which, like the XDR, include both simple types such as integers and booleans, and constructed types, such as arrays. The NIDL also supports a *transmissible* type, similar in concept to the XDR's opaque type.

The NIDL is designed to make data type definitions look very similar to data declarations in the user's programming language. So, for example, in the NIDL, the *GraphPoint* example used in Figure 3 might be expressed in this form:

```
...
typedef struct {     /* point on a graph */
        int xCoord;       /* x coordinate */
        int yCoord;       /* y coordinate */
} GRAPH$GraphPoint;
...
```

Figure #9

NIDL data definitions occur inside *interface* definitions for distributed systems. In the example in Figure 8, we can imagine the interface being named *GRAPH*.

If we encode the *GRAPH$GraphPoint* structure using the NDR, we get yet a different encoding from either ASN.1 or the XDR. Although it is true that like the XDR, the NDR does not supply tagging information, under the NDR, the encoding varies depending on the machine which encodes the data! The reason for this is that the NDR encodings are based on the ‘‘receiver makes it right’’ principle. When data is encoded, two items are produced: a *format label* and a *byte stream*. The label identifies the machine type which has encoded the data. It consists of four octets which specify the conventions used for representing scalar values, e.g., floating point format, byte-ordering for integers, and so on. The stream consists of the data values which have been encoded using those conventions. The obvious advantage of this approach is that when similar machines communicate, the necessary conversion routines are quite minimal. Of course, it also requires that each machine have resident several sets of decoding routines (one for each set of conventions, which we term a *configuration*).

4.1. Implementation Approach

The NIDL is implemented using a language compiler. (There are actually two compilers, which support the *C* and *Pascal* variants of data structures.) The NIDL compiler produces both the routines to encode and decode data types (which is called *marshalling* under NCA) and stub routines for remote procedure calls.

As expected, the basic encoding routines are tailored for the user's hardware platform and are usually quite simple to implement. The only exception to this would be if none of the configurations which can be defined using the format label exactly matched the user's platform. In this case, the user would need to select the supported configuration which most closely matched the underlying hardware (perhaps a subjective choice) and then implement encoding routines to map from the native format to the supported format.

4.2. Performance

Note that, between dissimilar hardware platforms, the NDR and the XDR are essentially equivalent with respect to their performance characteristics: neither scheme sends tagging information, and both schemes exploit the notion of fixed data sizes and blocking factors (the latter is called *natural alignment* under NCA). Given this, our arguments above regarding performance with respect to the XDR are also applicable here. Hence, it is only interesting to consider the case where similar hardware platforms are communicating. In this case, if the hardware platforms do not use the same byte–ordering for scalars as defined by the XDR, then using the NDR will result in a performance boost.

The disadvantage of the NDR approach is not in terms of performance, for which it is clearly optimized. The problem is that it is an ‘‘m×n approach’’ (see [10]). Under both ASN.1 and the XDR, for a given hardware platform, two sets of routines need be implemented: one set to encode native data types into a canonical form and another set to decode the canonical form into native data types. The closer the canonical form is to the native representation, the easier it is to implement these routines, and presumably the more efficient the routines as well. Under the NDR approach, if a hardware platform can be described exactly using the format label (as of this writing, a likely event), then only one set of simple encoding routines need be written, and then *n* sets of decoding routines need also to be written. Further, all of these sets of routines need be present with every program which uses the NDR since, in theory, any potential peer might be using any of the possible configurations in the NDR format label. In favor of NDR, one might argue that if the possible number of configurations is small (e.g., about a half−dozen or so), then this isn’t a great burden. However, this argument is only valid in the short−term: it is almost certainly true that new hardware platforms will require new configurations in the format label. However, since the NDR is currently deployed, adding new configurations will make interoperability with older systems impossible. If interoperability is to be retained, then these new systems will have to choose the existing configuration which is closest to their native format (as described above). At this point, the advantages of the NDR’s ‘‘receiver makes it right’’ principle are entirely lost −− whilst all the disadvantages remain!

This scenario is not as far−fetched as one might think, and it does well to point out one of the founding tenets of OSI: to promote interoperability between machines of different technologies. For example, because ASN.1 supports arbitrarily wide integers, as hardware platforms change and new machines emerge, ASN.1 will continue to work without any change.

This is not to say that one cannot develop an external data representation scheme that permits homogeneous machines to exchange data in their internal form, only that NDR uses an approach that researchers have long known is deficient. Examples of schemes that do work are negotiation, where two peers agree between themselves about whether to use a standard format or their internal format, or prior knowledge, where a server knows from previous contact that the client uses the same data format. (The authors believe the Network Computing Architecture could easily be extended to provide this information).

4.3. Specification Language

The Network Interface Definition Language is designed to describe interfaces for distributed applications. As such, in addition to permitting the definitions of data types, it also is used to define the interfaces themselves.

If we consider only the *C* variant of the language, then our arguments above regarding the XDR specification language are also applicable here. Further, given the closeness between the *C* programming language and *Pascal*, similar arguments may also be made for the *Pascal* variant of the language.

5. Conclusions

Three well−known data formatting approaches have been presented and their merits examined. Of these, ASN.1 has been shown to be the most general and extensible, both in regards to its specification language and encoding rules. It may be argued that the ASN.1 macro facility is *too* general to permit the widespread use of automated tools for arbitrary ASN.1 descriptions. However, this argument may best be resolved by encouraging practitioners to ‘‘try their hand’’ at this problem.

In defense of the other schemes however, there remains a class of distributed systems−−those typified by the use of remote procedure calls over a connectionless−mode association−−which require superior performance. In these circumstances, a less general approach than ASN.1, one which uses data alignment with a minimum of tagging information, may prove optimal for use. This result is due to the observation that performance is typically limited by the protocol processing time at the sender and receiver in these systems. Given a fixed maximum on the size of machine data structures, and relatively simple data types to be exchanged, then both the XDR and the NDR prove more effective. However, this argument can not be truly resolved until connection−less mode mappings are provided for the OSI Remote Operation Service, and then empirical evidence can be gathered as to the actual differences in performance. (Hopefully such mappings will occur before the inevitable next generation of hardware renders schemes like the XDR and the NDR inoperable.)

Finally, if one were to employ one of the less general schemes, then the authors favor use of the XDR because it properly handles the ‘‘m×n’’ problem (and thus has a longer practical lifetime) and because it is most likely simpler to implement.

References

[1] *Transmission Control Protocol*, U.S. MILSTD 1778, (September, 1981).

[2] R. Anderson *et al*, *Status Report on Proposed Data Reconfiguration Service* (RFC 138), in Internet Working Group Requests for Comments, no. 138. Network Information Center, SRI International, Menlo Park, California, (April, 1971).

[3] A.L. DeSchon, *A Survey of Data Representation Standards*, (RFC 971), in Internet Working Group Requests for Comments, no. 971. Network Information Center, SRI International, Menlo Park, California, (January, 1986).

[4] D. Deutsch, R. Resnick, and J. Vittal, *Specification of a Draft Message Format Standard*, Bolt Beranek and Newman Report, No. 4486, (September, 1980).

[5] D. Deutsch, R. Resnick, and J. Vittal, *Specification for Message Format for Computer Based Message Systems (Revised)*, Bolt Beranek and Newman Report, No. 4765R, (April, 1982).

[6] T.H. Dineen et. al., *The Network Computer Architecture and System: An Environment for Developing Distributed Applications* in Proceedings of the 1987 Summer USENIX Conference. Phoenix, Arizona.

[7] *Information processing systems — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1)*, International Organization for Standardization. Draft International Standard 8824, (May 1987).

[8] *Information processing systems — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Notation One (ASN.1)*, International Organization for Standardization. Draft International Standard 8825, (May, 1987).

[9] J.P. Onions and M.T. Rose, *The Applications Cookbook* in Proceedings, 1988 IFIP WG 6.5 International Working Conference on Message Handling and Distributed Application Systems, (October, 1988).

[10] M.A. Padlipsky, *A Perspective on the ARPANET Reference Model* in Proceedings of INFOCOM ’83. San Diego, CA.

[11] C. Partridge, *A UNIX Implementation of HEMS* in Proceedings of the 1988 Winter USENIX Conference. Dallas, Texas, (February, 1988).

[12] C. Partridge and G. Trewitt, *The High-Level Entity Management System* (RFCs 1021-1024), in Internet Working Group Requests for Comments, no. 1021-1024. Network Information Center, SRI International, Menlo Park, California, (October, 1987).

[13] A. Pope, *Encoding CCITT X.409 Presentation Transfer Syntax*, in ACM SIGCOMM Computer Communication Review, (October, 1984).

[14] J. Postel, *Internet Multimedia Document Format*, USC Information Sciences Institute MMM-12 (RFC 767-Revised). (March, 1982).

[15] I. Reid. *RPCC - A Stub Compiler for SUN RPC* in Proceedings of the 1987 Summer USENIX Conference. Phoenix, Arizona. (June, 1987).

[16] M.T. Rose, *ISO Presentation Services on top of TCP/IP–based internets*, Internet Engineering Task Force IDEA #17 (March, 1988).

[17] M.T. Rose, *ISODE: Horizontal Integration in Networking*, ConneXions, Volume 1, Number 1, (May, 1987).

[18] M.T. Rose, *Building Distributed Applications in an OSI Framework*, ConneXions, Volume 2, Number 3, (March, 1988)

[19] M.L. Schoffstall and W. Yeong, *SGMP/SNMP Implementation*, NYSERNet, Inc.

[20] Sun Microsystems, Inc., *Networking on the Sun Workstation*, Part number 800−1324−03, 1986.

[21] Sun Microsystems, Inc., *RPC: Remote Procedure Call Protocol Specification* (RFC 1050), in Internet Working Group Requests for Comments, no. 1050 Network Information Center, SRI International, Menlo Park, California, (April, 1988).

[22] Sun Microsystems, Inc., *XDR: External Data Representation Standard*, (RFC 1014), in Internet Working Group Requests for Comments, no. 1014. Network Information Center, SRI International, Menlo Park, California, (June, 1987).

[23] Xerox Corporation. *Xerox System Integration Bulletin*, OPD B018112, 1981.