4 THE LANGUAGES OF OPEN SYSTEMS

Introduction

Traditionally, data exchanged between applications across networks have been treated as arbitrarily long strings of "*n*-bit" bytes (typically, in TCP/IP and OSI, $n = 8$; hence, the term *octet* is used throughout). Applications did not distinguish how data were represented as an end (user) form from the way they were transferred. This was (and remains) especially true for the Internet application protocols, where the support of ASCII as the principal character set for providing human-to-terminal interfaces evolved in such a fashion that ASCII became the *de facto* programming language used for defining commands and replies (protocols) for computer-to-computer applications such as electronic mail and file transfer.

Parallel to the development of the OSI protocols, a set of nonproprietary languages has been developed and standardized for open systems to accommodate data representation ("abstract syntax notation") and a corresponding encoding for data transfer ("transfer syntax"). They provide a universal language for network application programming that enables applications to exchange values of data without losing the semantics of those data (how they are structured, what types of data are present in a complex data structure, how long the structure is). These languages are covered in some detail in this chapter, as they illuminate a number of differences in the approaches that have been applied to open systems networking by the OSI and TCP/IP architectures. Readers will also note that the languages ostensibly created for OSI have been applied to more recently developed Internet applications, most notably in the area of network management.

Equally important to the study of languages are the practices of naming and addressing; the definition and representation of names are discussed in this chapter, while the details of the overall naming and addressing strategies of OSI and TCP/IP are described in Chapter 5. A subpractice of naming—protocol identification ("protocol IDs")—is described in chapters in which this practice is relevant.

"Open" Languages—Breaking Language Barriers

A computer vendor promoting a proprietary network architecture based on its own products can define and use whatever terms it likes, name things without worrying about conflicts or ambiguity, and document its work using the descriptive notations and syntaxes that are most convenient for its technical writers and customer-training people. When you own the show, you can make the rules, and anyone who wants to participate has to learn to speak your language. The architectural "language barrier" is a powerful argument for network homogeneity—one that many computer vendors have used to effectively exclude their competitors' products from participation in networks based on their proprietary architecture. Homogeneity, of course, is precisely the prison from which the open systems networking concept promises to release the builders and users of networks. In order to do so, it must define and use terminology, naming schemes, and descriptive techniques—*languages*—that are universally understood.

It is hard to overemphasize the importance of languages in the development of open systems. During a meeting in 1988 of the ISO/IEC standards committee concerned with network-layer standards (ISO/IEC JTC 1/SC 6/WG 2), the editor for one of the OSI routing protocol standards, Dave Oran, was engaged in a discussion with another delegate about the relationship between level-1 and level-2 intradomain routing. Across the table, three Japanese delegates were following the discussion intently with the help of a Japanese-English dictionary. Repeatedly, Dave attempted to make what seemed to him an obvious point; the other delegate continued to disagree with equal resolve. After several iterations, it suddenly dawned on Dave that the other delegate's understanding of the situation was radically different from what he had assumed, at which point he sat back and said, "Well, if that's *what you mean, then we're* really *in deep sneakers!" During the brief silence that followed, one could hear the pages flipping in the dictionary as the Japanese dele-*

gates searched for the word sneakers*; then a buzz of whispered consultation among the three in Japanese and more page flipping to confirm that they had indeed found the correct definition of* sneakers*. Finally, the head delegate turned to Dave and, with an expression of intense consternation, asked, "Excuse me, Mr. Editor, but please explain* deep sneakers*."*

The work of all the major international standards organizations, including ISO and CCITT, is conducted almost exclusively in English; non–English-speaking delegates are expected to either learn to cope in "English-as-a-second-language" mode or provide their own translations of documents (a daunting prospect, considering the enormous amount of documentation that attends even the simplest standardization effort). The result is to give a significant practical advantage to native English speakers—and fluent non-native English speakers—in the formal standards-development process. In effect, ISO and CCITT have adopted English as the "standard" language for standards development, with a significant penalty for noncompliance.

Data Representation

Five thousand years ago, the Sumerians, Babylonians, and Egyptians encountered a problem conducting commerce in ancient Mesopotamia: although they generally agreed on the representation of simple numbers (*1*, *2*, and *3*) as recognizably similar stroke or cuneiform numerals, they did not agree on the representation of larger numbers and did not (originally) agree on syntactic markers for place value or the way in which graphic symbols for basic units (such as *10* and *60*) should be composed to represent a numerical sum (such as the value "80" represented by the composition of symbols for *60 + 10 + 10*). In the years since then, the problem of unambiguously representing numerical and non-numerical data has grown steadily worse, as the number of things demanding unambiguous representation has kept pace with the increasing complexity of social and economic intercourse. The absolute literal-mindedness of the computer turned this problem into a genuine nightmare, but it took networking to turn the problem into a nightmare of global proportions.

Although we have introduced a formal language and grammar—mathematics—the general issue of dealing with numbers has changed little since the time of ancient Mesopotamia. Even those who have had little formal education in mathematics understand the notion of a whole number or an integer—no fractional parts, positives and negatives, remember? Someone says "integer," and we think {. . . , –2, –1, 0, 1, 2, . . .}:

517 is an integer, *67,190* is an integer, and Avogadro's number[1] is an integer. We can pretty much grasp this without advanced degrees. The notion of integer in this context is *abstract*.

Generally speaking, programming languages such as C and Pascal have a model of "integer" that is less abstract than the pure number-theory concept of "integer" but more abstract than the way in which a particular operating system stores things that are labeled "integers" into containers (memory locations). Consider that when a mathematician says "integer," she is dealing with the entire unconstrained concept of integer (abstract); but when a programmer says "integer" to the UNIX 4.2bsd C compiler by declaring a variable to be of type `short`, she gets a signed 15-bit container capable of expressing "integer" values in the range –32,678 through 32,767, and if she declares the variable to be of type `unsigned long`, she gets a 32-bit container capable of expressing "integer" values in the range 0 through $2^{32} - 1$. These represent a more *concrete syntax* of integer—i.e., one that is semantically bound to the machine and operating system architecture on which the C program is compiled. And finally, when the C compiler generates machine-language instructions for a particular computer, the concept of "integer" is not even present—there is only the concept of binary numbers (a concrete representation) constrained to fit into containers of a certain size.

As a rule, when people discuss numbers, they don't think of "containers" for integers. Computers, and folks who program them, do (a preoccupation inherited in part from times when memory was a scarce and precious resource, and by today's standards, outrageously expensive). Now, containers have built-in limitations on ranges of numbers; for example, encoded in hexadecimal, the value "517" requires a 2-octet container; the value "67,190" requires 3 octets; and the value of Avogadro's number requires many. By and large, folks who program computers don't expect ranges of values as broad as Avogadro's number when they define data constructs in programming languages like C and Pascal as part of the process of writing applications. Typically, they define containers that appear to satisfy near- and modestly long-term needs. This is true for protocols as well (protocol headers are, after all, merely another form of data structure). A version field of a protocol doesn't appear to require more than an octet; heaven forbid we ever reach 255 versions of any single protocol and require yet another octet to encode version 256! On the

1. The number of atoms in a mole—the amount of any substance whose weight in grams is numerically equal to its atomic or molecular weight. Avogadro's number is 6.02×10^{23}.

other hand, a 16-bit container for the window-size field of TCP, which allowed for a 65,535-octet window, and the 32-bit sequence-number field seemingly satisfied long-term needs in the 1970s; with multimegabit, fiber "data pipes," these containers are now generally regarded as too small.[2]

There is another complication when dealing with the representation of numbers in computers and programming languages: even within the context of the same programming language, different machine architectures may interpret a data structure in an entirely different manner. Take the `int` example. Compiling the same C program containing `int` declarations on two different machines may produce two different results: on a DEC PDP-11, an `int` without qualification is 16 bits, whereas on a VAX 11/780, an `int` without qualification is 32 bits, effectively begging the question of whether either interpretation matches the programmer's intent.

Of course, such problems extend beyond the world of whole numbers; only some of the data we represent in computers are integers. The variation among machine architectures in their representation of more complicated data types—real numbers, complex numbers, characters, graphic strings—is even worse. This introduces yet another issue: how to preserve the semantics of information as well as the value when it is exchanged between two computers.

All of these factors make for pretty messy networking; clearly, there can be no networking of open systems without a standardized, machine-independent language in which it is possible to represent basic information elements in such a way that (1) the information can be interpreted unambiguously in any context; (2) the values of data structures can be, in principle, unbounded; and (3) the information can be conveyed between computer systems without loss of semantics.

OSI captures the meaning (semantics) of data exchanged between open systems (the *abstract syntax*) independently from the specification and internal representation of that data in a computer (the *concrete syntax*) and the bit patterns used to transmit the data structure from one computer to another (the *transfer syntax*). The separation of abstract from

2. In RFC 1323, *TCP Extensions for High Performance*, Jacobson, Braden, and Borman discuss the problem of dealing with "long delay paths"—links with high bandwidth × delay products; over such links (e.g., SONET OC-3C [155 Mbps] and future gigabit transcontinental U.S. fiber links), the 32-bit sequence number can "wrap" dangerously close to or faster than the 2-minute maximum segment limit assumed by TCP, and the 16-bit window size, which limits the effective bandwidth to $2^{16/RTT}$ (round-trip time) is insufficient to "fill the pipe." (Here, the U.S. transcontinental delay, approximately 60 milliseconds, represents a hard-and-fast lower bound on RTT, which cannot be defied.)

concrete and transfer syntax is significant in the sense that data can be represented without concern for the container size for the data or the manner in which they are recorded in any given computer.

Abstract Syntax Notation

Abstract Syntax Notation One (ASN.1 ISO/IEC 8824: 1987, which is equivalent to CCITT Recommendation X.208) is arguably the most widely accepted language for the representation of data for open systems networking. ASN.1 was originally developed as part of the work on OSI upper-layer standards to serve as a uniform canonical representation of any data type (both predefined, or standardized, data types and user-defined data types), so that *objects* in the OSI environment—protocol headers, electronic-mail message headers and bodies, directory entries, management information, and virtual filestores—might be conveyed from one system to another in a form that could be understood without reference (adherence) to any specific machine or operating system architecture. ASN.1 has since been adopted as the specification language of choice by people working in areas other than OSI, including TCP/IP, despite the fact that it is frequently criticized as expensive, in terms of processor cycles, to perform the transformation between a system's native (hardware- and operating system–specific) representation of data and ASN.1.

Readers who are familiar with the terminology of database management will recognize ASN.1 as a *data-description language*. ISO/IEC 8824 defines the language itself and the many predefined data types that it standardizes. A companion standard, ISO/IEC 8825: 1987 (CCITT Recommendation X.209), defines *basic encoding rules* (BER) for encoding the abstract data types of ASN.1 as actual bit streams that are exchanged by open systems. Although the standards do not require the exclusive use of the ISO 8825 basic encoding rules to encode ASN.1, in practice almost every encoding of ASN.1 is specified according to the basic encoding rules.

As a general-purpose data-description language, ASN.1 can be used in many different ways. Its original use, especially in its earliest incarnation as CCITT X.409-1984, was to specify the contents of the headers of the OSI Message Handling System and upper-layer protocols; ASN.1 descriptions are used to specify protocol data units, to define objects that can be used to manage network resources, and to define objects and object attributes that can be registered and entered into a global database or directory. ASN.1 offers the equivalent of programming tools of grammar. Using ASN.1, one can construct arbitrarily complex data structures

and retain the semantics of these data structures across many and diverse computer (operating) systems. One can think of protocol data units encoded in ASN.1 as verbs (they request actions such as get, set, modify, read, open, close, search, and initialize) and their objects as nouns—one "gets" the value of a management object, for example, or "reads" an attribute from a directory entry. There is even a means of assigning proper names—object identifiers.

ASN.1 Data Types and Tags

Like many programming languages, ASN.1 provides the means to identify the type of data structure. The data types that are predefined by ISO/IEC 8824 cover most of those types that are required for the specification of protocols (see Table 4.1).

In ASN.1, data are typed as either *simple* or *structured*. Simple data types are rather intuitive; they are data types that are defined by the set of values that may be specified for that type, for example:

- A BOOLEAN has two distinguished values ("true" and "false").
- An INTEGER may be assigned any of the set of positive and negative whole numbers.

TABLE 4.1 ASN.1 Data Types

Number (Tag)	*Type*	*Number (Tag)*	*Type*
1	BOOLEAN	17	SET, SET OF
2	INTEGER	18	NumericString
3	BITSTRING	19	PrintableString
4	OCTETSTRING	20	TeletexString (T61String)
5	NULL	21	VideotexString
6	OBJECT IDENTIFIER	22	IA5String
7	ObjectDescriptor	23	UTCTime
8	EXTERNAL	24	GeneralizedTime
9	REAL	25	GraphicsString
10	ENUMERATED	26	VisibleString
11	CHOICE	27	GeneralString
12–15	Reserved for Addenda		
16	SEQUENCE, SEQUENCE OF	28	CharacterString

- A REAL may be assigned any of the members of the set of real numbers—i.e., a number that can be stored in floating-point processors and represented by the general formula $mantissa \times base^{exponent}$.
- BITSTRING and OCTETSTRING may be composed of an ordered sequence of 0 or more bits and octets, respectively (and similarly for other character-string types—e.g., NumericString, Printable-String).
- The ever-interesting NULL may be assigned the singular value "null."
- An ENUMERATED type is one for which a list of values is identified as part of the type notation.

Structured data types are constructed from the simple data types. The availability of so many simple types alone provides for a wealth of combinatorial possibilities. Commonly used structured types are:

- SET, a fixed, unordered list of distinct types, some of which may be optional.
- SET OF, an unordered list of zero or more of the same type.
- SEQUENCE, a fixed, ordered list of distinct types.
- SEQUENCE OF, an ordered list of zero or more of the same type.
- CHOICE, like SET, a fixed, unordered list of distinct types, but in which any instance of a CHOICE takes the value of one of the component types.
- The nefarious ANY, a CHOICE type bounded not by a list of distinct types but by anything that can be defined using ASN.1, which is often used to indicate "I dunno" or "for further study".

Figure 4.1 provides examples of simple and structured data types.

The *OBJECT IDENTIFIER* data type serves as the basis for a general-purpose naming scheme that can be used to identify anything that can usefully be represented in an open systems architecture as an "object." As one might expect, there are a great many "things" in a network that can usefully be modeled as objects (borrowing at least some of the information-theoretical "object" model that has been used in, for example, object-oriented programming languages). *Object identifiers* (OIDs) serve as a uniform way to refer unambiguously to any of these "things" (this is examined more closely in Chapter 5).

Each data type is assigned a unique numeric *tag* for unambiguous identification within the domain of types. These are intended mainly for machine use, and they provide the data-stream identification of a data type. There are four classes of tags. The UNIVERSAL tags are defined in ISO/IEC 8824 and shown in Table 4.1. Tags that are assigned in other OSI standards—e.g., tags that identify protocol data units of applications

```
brainDamaged       ::= BOOLEAN

numberOfEmployees  ::= INTEGER  -- one typically hires integral numbers of employees

avogadrosNumber    ::= REAL { 602, 10, 23 }

                           -- value of Avogadro's number is 6.02 × 10^23

digitizedVoice     ::= BITSTRING

G3NonBasicParams   ::= BITSTRING {
                           twoDimensional(8),
                           fineResolution(9),
                           unlimitedLength(20),
                           b4Length(21),
                           a3Width(22),
                           b4Width(23),
                           uncompressed(30) } -— from CCITT X.411-1984

UMPDU              ::= OCTETSTRING    -- from CCITT X.411-1984
                                      -- no enumerated values, can be any length

sevenDeadlySins    ::= ENUMERATED { pride(1), envy(2), gluttony(3), avarice(4),
                           lust(5), sloth(6), wrath(7) }
                   -- corresponds to seven layers of OSI


messageBody        ::= SEQUENCE OF BodyPart
                   -- every element of "Body" is of type messageBodyPart, defined as
follows:
messageBodyPart    ::= CHOICE {
                           [0] IMPLICIT asciiText, -- an IA5STRING
                           [1] IMPLICIT telex,     -- an OCTETSTRING
                           [2] IMPLICIT voice,      -- a BITSTRING
                           [3] IMPLICIT G3Facsimile, -- a SEQUENCE
                           [4] IMPLICIT teletex,   -- an OCTETSTRING
                           [5] IMPLICIT graphicalImage } -- a BITSTRING
-- the use of the keyword IMPLICIT indicates that the tag
-- of the tagged types in the CHOICE need not be
-- encoded when data type is transferred; results in minimum
-- transfer of octets without loss of semantics

TeletexNonBasicParams  ::= SET {
                           graphicCharacterSets [0] IMPLICIT T61String OPTIONAL,
                           controlCharacterSets [1] IMPLICIT T61String OPTIONAL,
                           pageFormats [2] IMPLICIT OCTETSTRING OPTIONAL,
                           misTerminalCapabilities [3] IMPLICIT T61String
                                 OPTIONAL,
                           privateUse [4] IMPLICIT OCTETSTRING OPTIONAL }
-- OPTIONAL indicates that there is no constraint on the presence or absence of the ele-
ment type
```

FIGURE 4.1 Examples of ASN.1 Data Type Definitions

such as the X.500 Directory or the X.400 Message Handling System—are assigned APPLICATION-specific tags. Tags that have context within a

type (that is, within an already tagged type)—e.g., members of a set, elements of "fields" of a protocol data unit—are as-signed CONTEXT-SPECIFIC tags. Finally, ASN.1 has provisions for organizations and countries to define additional data types; types of this origin are distinguished from others of like origin by PRIVATE tags.

Modules

A set or collection of ASN.1-related descriptions is called a module. A module of ASN.1 statements can be compared to a source file from a library of files that are included in a C or Pascal program. Rather than labor through the fairly intuitive syntax of a module, a fragment of an ASN.1 module is illustrated in Figure 4.2 (the keywords and relevant elements of the module definition are in boldface type).

Note that like other programming languages, ASN.1 permits the import and export of modules of ASN.1 (absent in this example). Thus, if another application service would find it useful to import the FTAM defi-

```
ISO8571-FTAM DEFINITIONS ::=
BEGIN
PDU ::= CHOICE                { InitializePDU, FilePDU, BulkdataPDU }
InitializePDU ::= CHOICE      { [APPLICATION 0] IMPLICIT FINITIALIZErequest,
                              [1] IMPLICIT FINITIALIZEresponse,
                              [2] IMPLICIT FTERMINATErequest,
                              [3] IMPLICIT FTERMINATEresponse,
                              [4] IMPLICIT FUABORTrequest,
                              [5] IMPLICIT FPABORTresponse }
FINITIALIZErequest ::=        SEQUENCE {
                              protocolId[0] INTEGER { isoFTAM(0) },
  versionNumber[1] IMPLICIT   SEQUENCE { major INTEGER,
                              minor INTEGER },
                              serviceType[2] INTEGER { reliable (0), user correctable
                               (1) }
                              serviceClass[3] INTEGER { transfer (0), access (1),
                               management (2) }
                              functionalUnits[4] BITSTRING { read (0), write (1),
                               fileAccess (2),
-- definitions continue . . .
FINITIALIZEresponse ::= SEQUENCE . . .
-- definitions continue . . .
END
```

Source: ISO 8571, "File Transfer, Access, and Management" (1988).

FIGURE 4.2 Example of an ASN.1 Module

nitions, it could do so rather than defining these same ASN.1 types again.

Note that in addition to a module reference or name, the DESCRIPTION statement may include an *object identifier* to provide a globally unique identifier for this module. (Object identifiers are rather unique ASN.1 simple types that provide a universal identification mechanism and are covered under the general rubric of naming and addressing; see Chapter 5.)

Transfer Syntax—Basic Encoding Rules (BER) for ASN.1

Constructing the bit-stream representation of the number *255* is a relatively simple task—eight binary 1s, and you don't even have to worry about the order of bit transmission. It is quite another issue to indicate along with this bit-stream representation whether the originator of these eight binary 1s intended that they be interpreted as the integer *255* or 1 octet of a much larger integer or the "true" value of a BOOLEAN data type or the mantissa of a real number. To preserve semantics of a data type, to accommodate values of indefinite length, and to accommodate the transmission of complex data structures in which component data types may be present or absent (OPTIONAL), encoding and transmitting the value are simply not enough.

The bit patterns used to transmit ASN.1-encoded data types from one computer to another—the *transfer syntax*—are defined in ISO/IEC 8825: 1987, *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).* There are three components to an encoding: the identifier or *tag*, the *length*, and the contents or *value*; the term *TLV encoding* is derived from the names of the fields of encodings (Rose 1990).

To say that the basic encoding rules are arcane is an understatement. Like so many aspects of OSI, the efficiency of BER has been compromised by the perceived need for backward compatibility with its ancestor, CCITT X.409-1984. The identifier octets, which convey the type class and number (tag), are encoded in one of two manners. Bits 8 and 7 of the initial identifier octet indicate the *class*—UNIVERSAL (00), APPLICATION (01), CONTEXT-SPECIFIC (10), or PRIVATE (11). Bit 6 identifies whether the data type is primitive (0) or constructed (1). The remaining bits of this octet contain the tag number if the number is in the range $0 <$ tag number < 31:

Bits 8 and 7	Bit 6	Bits 5, 4, 3, 2, and 1
Class	Primitive/constructed	Tag number

For example, the identifier octet of the BOOLEAN "brainDamaged" from Figure 4.1 should convey the following information: UNIVERSAL, primitive, tag number 1. It is thus encoded in the following manner:

Bits 8 and 7	Bit 6	Bits 5, 4, 3, 2, and 1
00	0	00001

If the tag number exceeds 30, then the tag number field in the initial identifier octet is populated with the value "31," indicating "There are more identifier octets." Bit 8 of every subsequent octet is reserved as a flag, where the value "1" is used to indicate "More octets follow," and the value "0" indicates "This is the last octet of the tag number, I promise!" Thus, if an application had a protocol data unit with a tag number of 32, the identifier octets would look like this:

Initial Identifier Octet			Subsequent Octet	
Bits 8 and 7	Bit 6	Bits 5, 4, 3, 2, and 1	Bit 8	Bits 7–1
01	1	11111	0	0100000

There are two methods for encoding the length octets: using the *definite form*, 1 or more length octets indicate the length in octets of the value field. More bit cleverness is used to extend the length octets: here, bit 8 of the initial octet is reserved as a continuation/termination flag. A single length octet is used to indicate contents field lengths up to 127 octets (in such cases, bit 8 of this octet is 0); for contents fields with lengths greater than 127 octets, bit 8 of the initial octet is set to 1, and bits 7 through 1 indicate the number of subsequent octets in the length octets field. Thus, if the contents field of a data type is 202, the length octets are encoded as follows:

Initial Length Octet		Subsequent Octet
Bit 8	Bits 7–1	Bits 8–1
1	0000001	11001010

For the *indefinite form*, the initial length octet serves as the initial delimiter of the contents field: bit 8 of this octet is set to 1, bits 7 through 1 to 0—i.e., represented in hexadecimal as 01111111. The contents field follows this octet and is variable in length. An *end-of-contents* field follows the contents field: it is encoded as 2 octets containing binary 0s (treated as a basic encoding of a UNIVERSAL tag value zero of zero length).

Do I Really Have to Deal with All This?

Just as programmers are no longer expected to deal directly with low-level machine languages, protocol implementers are not expected to deal directly with ASN.1 and certainly not with BER unless they wish to. Public-domain and commercial ASN.1 compilers are available that accept a formal ASN.1 specification as input and produce reasonably machine-independent high-level language code (most commonly C code) for the programming language data structures that result from applying the basic encoding rules (or in some cases, any other encoding rules specified by the user) to the ASN.1 input stream. This permits implementers to deal with familiar elements of their favorite programming language to process protocol headers and other data items originally specified using ASN.1.

Running the protocol header specified formally by the ASN.1 statements shown in Figure 4.3 through an ASN.1 compiler might produce the C data structure shown in Figure 4.4.

This code would typically be incorporated into a program that would use it to construct outgoing protocol headers and to parse incoming ones. The actual bits transmitted and received would depend on the values given to the individual elements of the data structure in each instance.

It is possible to dig much deeper into the world of ASN.1; readers who are interested in doing so should consult Steedman (1990), which is entirely concerned with ASN.1. Other fertile but predictably harshly critical sources for acquiring more knowledge about ASN.1 are Rose (1990, 1991).

```
simpleDatagram DEFINITIONS ::=
BEGIN
PDU        ::= SEQUENCE {
                          protocolIdentifier [0]   INTEGER,
                          versionNumber [1]     INTEGER,
                          sourceAddress [2]     INTEGER,
                          destinationAddress [3]   INTEGER,
                          userData [4]          OCTETSTRING }
END
```

FIGURE 4.3 simpleDatagram Protocol Header in ASN.1

```
# define maxUserDataLength 255
struct _pduStruct_t{
                    unsigned long        protocolIdentifier;
                    unsigned long        versionNumber;
                    unsigned long        sourceAddress;
                    unsigned long        destinationAddress;
                    userData             char[maxUserDataLength]
} pduStruct-t
```

FIGURE 4.4 simpleDatagram Protocol Header in C

Languages and the TCP/IP Community

Most of the applications written for TCP/IP are written in a programming language and compiled to an executable format suitable to the machine and operating system on which the application will reside. Application protocol data units are often plain-text ASCII with specific guidelines for the interpretation of keywords, "white space," special characters, and escape character sequences (see, for example, Internet mail, discussed in Chapter 8). Although this may not at first seem particularly elegant, it has the very desirable characteristics of (1) being nearly universally understood and (2) getting the job done.

Some TCP/IP applications make use of machine-independent data-definition languages. The popular *Network File System* (RFC 1094; Sandberg 1988) uses *External Data Representation* (RFC 1014), which, although admittedly highly optimized for translation to and from UNIX/ C data representations, is easily ported to operating systems such as MS-DOS. ASN.1 is used in the definition of the Simple Network Management Protocol (SNMP; see Chapter 9) and in the definition of managed objects for the SNMP; and of course, the OSI upper-layers implementations that run on top of TCP/IP are encoded in ASN.1.

Only a subset of ASN.1 is used to define SNMP. The full comple-

```
IfEntry ::=
    SEQUENCE {
      ifIndex
        INTEGER,
      ifDescr
        DisplayString,
      ifType
        INTEGER,
      ifMtu
        INTEGER,
      ifSpeed          --a gauge is an application-specific ASN.1 data structure in SNMP
        Gauge,         --it is a 32-bit INTEGER that can increase or decrease but will not
“wrap”
      ifPhysAddress
        OCTETSTRING,
      ifAdminStatus
        INTEGER,
      ifOperStatus
        INTEGER,
      ifLastChange     --timeticks is an application-specific ASN.1 data structure in
SNMP
        Timeticks,     --it is an INTEGER; each increment represents .01 second of time
      ifInOctets       -- a counter is an application-specific ASN.1 data structure in
SNMP
        Counter,       --it is a 32-bit INTEGER that monotonically increases and “wraps”
      ifInUcastPkts
        Counter,
      ifInNUcast Pkts
        Counter,
      ifInDiscards
        Counter,
      ifInErrors
        Counter,
      ifInUnknownProtos
        Counter,
      ifOutOctets
        Counter,
      ifOutUcastPkts
        Counter
      ifOutNUcastPkts
        Counter,
      ifOutDiscards
        Counter,
      ifOutErrors
        Counter,
      ifOutQLen
        Gauge,
```

FIGURE 4.5 SNMP Table `ifTable` in ASN.1

ment of ASN.1 data types was considered to be burdensome to impose on computer systems that had roles in life other than performing management operations. The predominant sentiment in the Internet community is that like many OSI efforts, ASN.1 tried to be all things for all people, for all time, and the result was something certainly general but too complex. SNMP uses the simple types INTEGER, OCTETSTRING, OBJECT IDENTIFIER, and NULL, as well as the structured types SEQUENCE and SEQUENCE OF (basically, everything you could construct using the standard C programming-language data structures, `int` and `char`). The other data types that are needed are emulated using this reduced set. A BOOLEAN, for example, is represented as an INTEGER with two values, true and false; relative time is represented as an INTEGER as well, with each INTEGER value representing a hundredth of a second, or a "timetick." The rest of the ASN.1 types, it is argued, are inessential (and expensive) luxuries.

An example of ASN.1 use in the SNMP is shown in Figures 4.5 and 4.6. The SNMP table `ifTable` was compiled using an ASN.1 compiler; the compiler creates an "include" file, which contains the C data structure shown in Figure 4.6.

```
typedef
struct _ifEntry_t {
    long                ifIndex;
    OctetString         *ifDescr;
    long                ifType;
    long                ifMtu;
    unsigned long       ifSpeed;
    OctetString         *ifPhysAddress;
    long                ifAdminStatus;
    long                ifOperStatus;
    unsigned long       ifLastChange;
    unsigned long       ifInOctets;
    unsigned long       ifInUcastPkts;
    unsigned long       ifInNUcastPkts;
    unsigned long       ifInDiscards;
    unsigned long       ifInErrors;
    unsigned long       ifInUnknownProtos;
    unsigned long       ifOutOctets;
    unsigned long       ifOutUcastPkts;
    unsigned long       ifOutNUcastPkts;
    unsigned long       ifOutDiscards;
    unsigned long       ifOutErrors;
    unsigned long       ifOutQLen;
    OID                 *ifSpecific;
} ifEntry_t;
```

FIGURE 4.6 `ifTable` in C, as Generated by an ASN.1 Compiler

Conclusion

This chapter has described the formal way in which data are represented in OSI applications and the means by which not only the values of data but the semantics associated with them are transferred between distributed applications. The authors have described only enough of ASN.1, the data-definition language used in the development of OSI application services, to enable readers to understand some of the protocol examples given in the text. Readers interested in the details and intricacies of ASN.1 are encouraged to read Steedman (1990). This chapter has also described the way data are represented and manipulated by TCP/IP applications and how ASN.1 is applied in the Internet as well as the OSI community, but with an economy of data-type definitions (see also RFCs 1155, 1157, and 1213). And finally, the chapter has introduced the notion of object identification; in the following chapters, readers will appreciate the important role that object identifiers play in the open systems "name game."