Presentation Layer

The presentation layer is concerned with preserving the *meaning* of information sent across a network. The presentation layer may *represent* (encode) the data in various ways (e.g., data compression, or encryption), but the receiving peer will convert the encoding back into its original meaning. The presentation layer concerns itself with the following issues:

1. *Data Format.* Converting the complex data structures used by an application — strings, integers, structures, etc. — into a byte stream transmitted across the network.

   Representing information in such a way that communicating peers agree to the *format* of the data being exchanged. E.g., How many bits does an integer contain?, ASCII or EBCDIC character set?

2. *Compressing data* to reduce the amount of transmitted data (e.g., to save money).

3. Security and Privacy issues:

   Encryption : Scrambling the data so that only authorized participants can unscramble the messages of a conversation. Recall, that it's easy to "wiretap" transmission media such as Ethernets.

   Authentication : Verifying that the remote party really is the party they claim to be rather than an impostor.

   Note: *Encryption* is the solution to these problems. Where should encryption be done? Data link or presentation layer? It is not exactly clear. For instance, it is easy to add encryption at the data link layer, encrypting every transmitted frame. However, if you are concerned about security, would you trust someone else to perform encryption for you? On the other hand, having the presentation layer perform encryption leaves the packet headers of the lower layers unencrypted, allowing intruders to perform *traffic analysis*.

Abstract Syntax Notation

Abstract Syntax Notation (ASN.1) is an ISO standard that addresses the issue of representing, encoding, transmitting, and decoding data structures. It consists of two parts:

1. An *abstract syntax* that describes data structures in an unambiguous way. The syntax allows programmers to talk about "integers", "character strings", and "structures" rather than bits and bytes.
2. A *transfer syntax* that describes the bit stream encoding of ASN.1 data objects.

Send: data and additional fields to describe the type of data.

Upon receipt of data, the reverse operation takes place, with the receiver converting from ASN.1 format to the internal representation of the local machine.

Alternative approaches to the data representation problem:

1. Have the sender convert data into the format expected by the receiver, so that the receiver doesn't have to perform any decoding. The disadvantage to this approach is that every sender needs to know how to encode data for *every* possible target machine.
2. ASN.1 takes the approach of converting everything into a common form, much like the "network standard representation" of TCP/IP. However, this approach has the disadvantage that communication between two identical machines results in (needless) conversions.

ASN.1's abstract syntax is similar in form to that of any high level programming language. For instance, consider the following C structure:

```
  struct Student {
    char name[50];   /* ``Foo Bar'' */
    int grad;        /* Grad student? (yes/no) */
    float gpa;       /* 1.1 */
    int id;          /* 1234567890 */
    char bday[8];   /* mm/dd/yy */
  }
```

Its ASN.1 counterpart is:

```
  Student ::= SEQUENCE {
    name        OCTET STRING, -- 50 characters
    grad        BOOLEAN,      -- comments preceded
    gpa         REAL,         -- by ``--''
    id          INTEGER,
    bday OCTET STRING  -- birthday
  }
```

ASN.1 consists of primitive types, and complex types built from primitive types. The names of primitive types are written (by convention) in upper case, and the following primitive types are defined:

INTEGER : An integer (of arbitrary length).

BOOLEAN : TRUE or FALSE.

BIT STRING : Sequence of zero or more bits. Bit string values are written as '01001101'B (for binary) or '4D'H (in hex).

OCTET STRING : List of zero or more bytes. Used to represent strings, they have no maximum length.

ANY : A union of all types (e.g. one of several different possible values).

REAL : A real number.

NULL : No type at all, corresponds to "NIL" in C.

OBJECT IDENTIFIER : The name of an object (e.g. library). When a session is established, both sides negotiate about which ASN.1 objects they will be using.

Primitive types can be combined into more complex types. ASN.1 provides the following constructors:

SEQUENCE : Ordered list of various types (like a C structure). Types can be either primitive or complex types.

SEQUENCE OF : Ordered list of a single type (e.g., an array).

SET : Unordered collection of various types.

SET OF : Unordered collection of a single type.

Note: SET and SET OF are similar to SEQUENCE and SEQUENCE OF, except that the order of components is not guaranteed to be preserved at the receiver. Using SET or SET OF may reduce the amount of copying relative to SEQUENCE or SEQUENCE OF.

CHOICE : Any one type taken from a given list.

ASN.1 allows fields to be declared OPTIONAL or DEFAULT. The idea is to allow the sender to omit parts of a data structure and let the omitted fields take on default values. For data structure containing many components, this may lead to substantially less actual data transferred.

However, the use of the OPTIONAL or DEFAULT types leads to potential problems. Suppose that a SEQUENCE has ten fields, all of them of type INTEGER and all OPTIONAL. If only three fields were transmitted, how would the receiver know which three they are?

ASN.1 uses *tagging* to solve this problem, allowing any data type or field to have a tag that identifies it. Tags are written in square brackets, and the following four types have been defined:

UNIVERSAL : Tags that are universally defined and globally unique. Such tags are defined in the ASN.1 standards.

APPLICATION : Tags that are unique within a given ASN.1 module. In any particular ASN.1 module, only one data type may have a given tag.

PRIVATE : The tag must be unique within a given enterprise, as provided by bilateral agreement. Used between principles (e.g. organizations) that have agreed to a set of tags that can be used when communicating with each other (e.g., a library of tags).

CONTEXT SPECIFIC : Tags that are unique within a given constructor type, such as SEQUENCE.

For example, [APPLICATION 4] refers to an APPLICATION tag; [PRIVATE 12] refers to a PRIVATE tag; and [44] refers to a PRIVATE tag (the default if no type is specified).

Now, if only three items of a 10-item sequence are transmitted, the receiver can use the tags to determine which fields have been transmitted.

Note: Whenever we send an item, we send its type, length, and value. If it is tagged, we also transmit its tag. Thus, we've added redundancy to our representation. If we include a tag, the tag *implicitly* specifies the type. To suppress the transmission of redundant information, ASN.1 also allows an IMPLICIT keyword to be appended after tag. When specified, the IMPLICIT keyword surpresses transmission of the type information.

Back to our original example, all tags are of type CONTEXT SPECIFIC:

```
Student ::= SEQUENCE {
   name    [0] IMPLICIT OCTET STRING OPTIONAL,
   grad    [1] IMPLICIT BOOLEAN OPTIONAL DEFAULT FALSE,
   gpa     [2] IMPLICIT REAL OPTIONAL,
   id      [3] IMPLICIT INTEGER,
   bday    [4] IMPLICIT OCTET STRING OPTIONAL
}
```

In addition to defining types, the ASN.1 syntax includes rules for defining values. Sample values for our example are:

{ "Wiz Kid", TRUE, { 4, 10, 0}, 123456789, "11/11/65" }

{ "Slow Learner", FALSE, {12, 10, -1}, 123456780, "12/24/88" }

Note: REAL values contain 3 components, a *mantissa*, *base*, and *exponent*. Thus, 4, 10, 0 means 4×10^0.

When transmitting a Student value, the following might be possible:

{ *"Wiz Kid" [1], 123454321*} : Here, we omit the *grad, gpa,* and *bday* fields. *Grad* defaults to FALSE, while *gpa* and *bday* are unspecified.

{ *"Slow Learner" [1], TRUE, "11/5/60" [4]*} : An error; we must send an id.

{ *"11/5/60" [4], 123454321*} : The tag [4] distinguishes *bday* from *name*.

ASN.1 is currently used in the Internet as part of the *Simple Network Monitoring Protocol* (SNMP), a protocol used to query gateways. SNMP is used to ask a gateway about its routing tables, the status of its interfaces, etc.

Sun's XDR

Sun Microsystem's External Data Representation (XDR) is an alternative to ASN.1. XDR is much simpler than ASN.1, but less powerful. For instance:

1. XDR uses *implicit typing.* Communicating peers must know the type of any exchanged data. In contrast, ASN.1 uses *explicit typing*; it includes type information as part of the transfer syntax.
2. In XDR, all data is transferred in units of 4 bytes. Numbers are transferred in network order, most significant byte first.
3. Strings consist of a 4 byte length, followed by the data (and perhaps padding in the last byte). Contrast this with ASN.1.
4. Defined types include: integer, enumeration, boolean, floating point, fixed length array, structures, plus others.

One advantage that XDR has over ASN.1 is that current implementations of ASN.1 execute significantly slower than XDR.

Data Compression

Why do data compression? What is it? Trying to reduce the amount of data sent. What is the tradeoff? Higher overhead on each end.

Suppose that we wanted to transfer a 20 Mb file to another machine. Would we really need to send 20 Mb of data? If the file consisted entirely of the letter "A", we could send the letter "A", followed by a count of the number of times it appears in the file, and have the receiver regenerate the file.

There are three general approaches to data compression. Each approach assumes that the data stream can be transformed into a more compact representation, which the receiver reconstructs back into the original data.

Approach 1: Finite Set of Symbols

Consider a library with many branch offices in which the previous days transactions are sent to every other branch after closing. Transactions consist of checked out and returned books. We could exchange information in the following ways:

1. We could send the name of the book, its author, the copy number, etc. together with the type of transaction.
2. Alternatively, the library could maintain a sitewide table assigning a unique ID number to every book in every branch. Transactions could then refer to the book's ID number, rather than its title. Because book IDs are small (e.g. a few bytes), less data will be transmitted.

Note: The above technique is used throughout programming. We frequently exchange pointers and array subscripts to avoid the cost of transferring large amounts of data between subroutines.

The previous approach assumes that all objects occur with equal frequency, and that the set of objects (e.g., books) is finite. If we examine text, however, we immediately notice that some words appear more often than others. We could reduce the number of bits needed to represent a document by using a coding scheme that employs small code words to represent common words and longer code words to represent words the appear infrequently.

Approach 2: Huffman Encoding

Huffman encoding is a technique used to encode symbols according to the frequency of their use. The algorithm is as follows:

1. Create a set of nodes, one node per symbol, with a node's value given by the probability of its occurrence in the data.
2. Find the two nodes having the smallest value, remove them from the set, and create a new node having the two removed nodes as children, and assign the new node a value that is the sum of its children's values. Add the new node back to the set of nodes.
3. Repeat step 2 until only one node remains. We now have a tree, whose probability value is one.
4. The encoding for each symbol is the path from the root to the symbol. Using a code of 0 for a left child, 1 for a right child, the length of each symbol's encoding is proportional to the relative probability of its occurrence.

One drawback with Huffman encoding, however, is that symbols have differing lengths, making it relatively expensive (computationally) to decode. Also, A single-bit error can wipe out the entire message.

Approach 3: Context Dependent Encoding

The last technique, context dependent encoding, recognizes that the probability of a particular symbol occurring next depends on the previous symbol. For instance, the probability that a "T" directly follows a "Q" is about 4 times less than the probability of a "U" following a "Q".

The main disadvantage of conditional probability methods is the increase in table space. Each symbol has its own table that gives the codes for those symbols immediately following it. For K symbols, the tables will contain K^2 entries. All K symbols have K entries for the symbols that follow them.

One variation to this technique is as follows:

1. Use 5-bit code to represent symbols.
2. Have four different "modes", where the current mode determines the meaning of symbols. Four symbols are reserved to denote a switch to another mode. For instance, different modes could represent upper-case characters, lower-case characters, numeric and special characters, and control characters.
3. The underlying assumption is that lower-case letters are likely to follow lower-case letters, numbers are likely to occur in groups, etc.
4. The occurrence of a mode symbol signifies a mode change.
5. Our 5-bit code now represents $4 \times 28 = 112$ different values.

One advantage that the above technique has over Huffman encoding is that symbols are all fixed length, making encoding and decoding using table lookups very efficient. In addition, it is more immune to transmission errors.

Run Length Encoding

Another alternative, *run length encoding*, is used to encode data containing repetitive symbols. Consider binary strings of 0s and 1s. One way to handle long runs of 0 is to use a k-bit symbol that tells how many 0 bits occurred between consecutive 1s. A code word of all 1's means that the true distance is $2^K - 1$ plus the value of the following symbol(s). For example:

```
00010000101001100000000000000000000000100000001  (47 bits)

consists of runs  of length 3, 4, 1, 2, 0, 23, and 7.

Using 4-bit symbols, it would be encoded as:
0011 0100 0001 0010 0000 1111 0100 0111, for 32 bits and
a savings of 15/47 = 30%.

Using 3-bit symbols, it would be encoded as:
011 100 001 010 000 111 111 111 010 111 000 for 33 bits.
```

Yet another variation is to encode the difference between the current value and the previous value, as was used to reduce the number of bits needed in pulse coded modulation (PCM).