Chapter 11
The Presentation Layer

The job of the presentation layer is to ensure that the data at one end of a connection is interpreted in the same way when it reaches the other end of the connection

Chapter 11
The Presentation Layer

For example, how do we encode the letter 'A'?

One popular way is to use a 7 bit number, namely 65

Chapter 11
The Presentation Layer

The *American Standard Code for Information Interchange* (ASCII) is one standard for encoding letter, digits and various punctuation marks

However, it is not the only standard and that is precisely the problem

Presentation

Character Encodings

- When the Internet began IBM's *Extended Binary-Coded Decimal Interchange Code* (EBCDIC) was still strong
- The purpose of EBCDIC is the same as ASCII: encoding characters as numbers

Presentation

Character Encodings

- The problem is that a file containing the bytes

  108, 97, 110

would read as “lan” on an ASCII system, but “%/>” on an EBCDIC system

- In ASCII, the value 108 means the character 'l'
- In EBCDIC, the value 108 means the character '%'

Presentation

Character Encodings

- The *presentation problem* is to ensure that this file reads the same on *any* system

Presentation

Philosophy

- The *bits* are the same, but our *interpretation* changes
- So to make our interpretation consistent we have to *change the bits*
- But not only *how* to change them, but *when*

Presentation

Philosophy

- If the file 108, 97, 110 is a text file, we must change the values to ensure consistent interpretation
- If this is a list of the IQs of three people, we must *not* change the values

Presentation

Philosophy

- Everything depends on the final *interpretation* of the data: this is a subtle point and is why presentation issues are often ignored or incorrectly implemented

Presentation

Character Encodings

- These days most people have more-or-less settled on ASCII as the encoding to use for simple Roman letters and digits
- So presentation issues are minimal for these kinds of data

Presentation

Character Encodings

- On the other hand, other character sets (Chinese, Russian, Klingon, etc) are still somewhat in flux, with the *Universal Character Set* (UCS) plus *Unicode* looking to be the winning solution

Presentation

UCS/Unicode

- UCS is a character encoding that uses 31 bits instead of just 7
- This gives ample room for all the characters in all the written languages in the world
- Unicode takes UCS and adds details like direction of writing (left-to-right or right-to-left or bidirectional), defining alphabetic orders, and so on

Presentation

UCS/Unicode

- Using 4 bytes per character would not be appreciated by many programmers since it would
  - break the “one character is one byte” assumption many programs make
  - make data files four times as large when the original data are encoded in ASCII, and
  - the zero byte is conventionally used to mean “end of string” so a value such as (hex) 12340078 is open to misinterpretation

Presentation

UCS/Unicode

- So some intermediate systems are defined
- Some are backwardly compatible with ASCII in the sense that values 00 to 7f are the same as their ASCII equivalents

Presentation

UCS Encodings

- The simplest method, UCS-4, translates ASCII to UCS by merely adding three 0 bytes before every ASCII byte
- This has the expansion and zero problems

Presentation

UCS Encodings

- Less inflationary is UCS-2, that inserts a single 0 byte
- This only doubles the size of an ASCII file
- Still has the zero problem
- Can't represent all possible UCS values

Presentation

UCS Encodings

- The *UCS Transformation Format*, (UTF-8) represents all ASCII (7 bit) values as themselves while still being able to represent all UCS values

Presentation

UCS Encodings: UTF-8

- UCS values 00000000 to 0000007f are transformed into bytes 00 to 7f. Thus an ASCII file is a valid UTF-8 file
- UCS values 00000080 to 000007ff become two bytes 110xxxxx 10xxxxxx. The bits from the UCS values are copied across

Presentation

UCS Encodings: UTF-8

- So '£', UCS 000000A3, binary

  00000000 00000000 00000000 10100011

  becomes 11000010 10100011 (C2A3), since

  00010/100011 → 110/0010 10/100011

Presentation

UCS Encodings: UTF-8

- Generally:

UCS Range (hex)	UTF-8 (binary)					
00000000-0000007F	0xxxxxxx					
00000080-000007FF	110xxxxx	10xxxxxx				
00000800-0000FFFF	1110xxxx	10xxxxxx	10xxxxxx			
00010000-001FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
00200000-03FFFFFF	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
04000000-7FFFFFFF	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

Presentation

UCS Encodings: UTF-8

- Some values require up to 6 bytes
- Most common values only require three or fewer
- ASCII values only require one byte
- An ASCII file already a UTF-8 file and there is no expansion of data when regarding it as UCS

Presentation

UCS Encodings: UTF-8

- The convention of using 0 as end of string still works
- When dipping at random into a UTF-8 encoded file it is easy to find the start of the next character: just search until you find a byte starting with 0 or 11
- The length of each non-ASCII character is given by the number of leading 1 bits

Presentation

UCS Encodings: UTF-8

- All UCS values can be encoded
- The comparison order of UCS is preserved

Presentation

Character Encodings

- Presentation for characters is not a solved problem: at the very least we need to get people to actually *use* the standard

Presentation

Number and Other Encodings

- Another big presentation problem is the byte order used for representing numbers
- An integer is typically represented using four bytes: but how those bytes are used varies

Presentation

Integer Encodings

- Some machines use *big endian* format: this stores the most significant byte of an integer (the *big end*) at the lowest machine address, less significant bytes at increasing addresses
- Others use *little endian* format: the least significant byte (*little end*) is stored at the lowest machine address, more significant bytes at increasing addresses

Presentation

Integer Encodings

... 99 100 101 102 103 104 ...

00 00 00 2A

... 99 100 101 102 103 104 ...

2A 00 00 00

Other arrangements are possible, too

Presentation

Integer Encodings

- If the machine receives four bytes

00 00 00 2A

does this mean the integer 42 (hex 0000002A) or the integer 704643072 (hex 2A000000)?

- A typical solution is to pick a single order (the *network byte order*) and always transmit bytes in that order

Presentation

Integer Encodings

- When a machine wants to send a value, it converts it to network byte order
- When a machine receives a value it converts it to its native order
- The *de facto* order used on many networks is big endian

Presentation

Integer Encodings

- A big endian machine has nothing to do
- A little endian machine must reverse the order of the bytes as it sends or receives
- Typically, a little endian machine *always* converts, even when connected to another little endian machine
- This is simpler than having a protocol to negotiate endianness and having separate bits of code for each combination

Presentation

Number Encodings

- Then there is the problem for other types of numerical data, e.g., floating point
- Here there is not only the byte order problem, but which and how many bits are used for exponents and mantissas and so on
- Fortunately, most have plumped for the IEEE standard floating point representation

Presentation

XDR Encodings

- The IP model has no presentation layer, so presentation issues are not addressed by IP
- Instead, programmers must use things like the *XDR* package when they send data over IP
- The *External Data Representation* (XDR) package is one approach to the presentation problem

Presentation

XDR Encodings

- It is a collection of functions that convert data in and out of a standard network format
- XDR sees to swapping the byte orders in integers when necessary, converting between floating point standards, and so on
- Suppose we want to send some integers from one machine to another

Presentation

```
#include <stdio.h>
#include <rpc/rpc.h>

int main(int argc, char **argv)
{
  XDR xdrs;
  int i;

  xdrstdio_create(&xdrs, stdout,
XDR_ENCODE);
  for (i = 0; i < 10; i++) {
    if (xdr_int(&xdrs, &i) == 0) {
      perror("xdr_int failed");
      exit(1);
    }
  }
  xdr_destroy(&xdrs);

  return 0;
}
```

- This code produces 10 XDR encoded integers on the standard output
- xdrstdio_create makes a *handle* connected to the standard output stdout

Presentation

```
#include <stdio.h>
#include <rpc/rpc.h>

int main(int argc, char **argv)
{
  XDR xdrs;
  int i;

  xdrstdio_create(&xdrs, stdout,
XDR_ENCODE);
  for (i = 0; i < 10; i++) {
    if (xdr_int(&xdrs, &i) == 0) {
      perror("xdr_int failed");
      exit(1);
    }
  }
  xdr_destroy(&xdrs);

  return 0;
}
```

- Calls to xdr_int will ENCODE integers and write them to the standard output
- At the end we destroy the handle to tidy up

Presentation

```
#include <stdio.h>
#include <rpc/rpc.h>

int main(int argc, char **argv)
{
  XDR xdrs;
  int i, n;

  xdrstdio_create(&xdrs, stdin,
XDR_DECODE);
  for (i = 0; i < 10; i++) {
    if (xdr_int(&xdrs, &n) == 0) {
      perror("xdr_int failed");
      exit(1);
    }
    printf("%d ", n);
  }
  putchar('\n');

  xdr_destroy(&xdrs);

  return 0;
}
```

- This reads 10 integers from the standard input and prints them
- Now xdr_int reads integers and DECODEs them

Presentation

XDR

- These programs will work and print the correct values no matter which kinds of machines they run on
- There are similar functions for characters, long integers, floating points and many more types of data

Presentation

XDR

- Notice the symmetry of using xdr_int for both encoding and decoding: the direction is in the handle, not the function
- Lately an XML-based alternative to XDR has become common for use over the Web: we will see more later

Presentation

MIME

- Another approach is the *Multipurpose Internet Mail Extension* (MIME)
- It originally addressed presentation in email, but is now used more widely, e.g., in the Web
- Early email systems only supported ASCII text and MIME was developed to allow emailing of pictures, sounds and so on

Presentation

MIME

- MIME regards data as a sequence of 8 bit bytes and encodes them in one of a variety of ways
  - 7bit: no transformation, only useful if the data were already ASCII
  - 8bit: no transformation, for some of the basic extension to ASCII

Presentation

MIME

- quoted-printable: bytes with values less then 128 represent themselves. Values over 128 are represented by an '=' followed by a two digit hex value

Thus 193 (which might represent Á) becomes three characters =C1

Presentation

MIME

- base64: the input is transformed into a 65 character subset of ASCII, namely

  A-Z a-z 0-9 + /

plus = as a special pad character

The 64 non-pad characters can be represented in 6 bits

Presentation

MIME: base64

- The transform takes three 8-bit bytes and regards them as four 6-bit values
- These are encoded and output in the restricted subset as four characters
- The pad character is needed when the original is not a multiple of three bytes long

Presentation

0 A	16 Q	32 g	48 w
1 B	17 R	33 h	49 x
2 C	18 S	34 i	50 y
3 D	19 T	35 j	51 z
4 E	20 U	36 k	52 0
5 F	21 V	37 l	53 1
6 G	22 W	38 m	54 2
7 H	23 X	39 n	55 3
8 I	24 Y	40 o	56 4
9 J	25 Z	41 p	57 5
10 K	26 a	42 q	58 6
11 L	27 b	43 r	59 7
12 M	28 c	44 s	60 8
13 N	29 d	45 t	61 9
14 O	30 e	46 u	62 +
15 P	31 f	47 v	63 /

- Example message “bit”
- Binary: 01100010 01101001 01110100
- 6 bit: 011000 100110 100101 110100
- Encoding: “Yml0”

Presentation

MIME: base64

- Decoding is a simple reversal of the above
- Note there is a 33% expansion of the data
- Both base64 and printed-quotable reduce the range of values used in the hope that they will be transmitted correctly

Presentation

MIME

- MIME encapsulation adds many headers
  - Mime-version: 1.0
  - Content-Type: text/plain; charset=ISO-8859-15 the original data was text using the ISO-4489-15 character set, a simple extension to ASCII
  - Content-transfer-encoding: base64 the encoding this message uses

Presentation

MIME

- And lots more
- The data follows after a blank line after the MIME header

Presentation

MIME

- The message “£100 is about €150” could become

Content-Transfer-Encoding: quoted-printable

Content-Type: text/plain; charset=ISO-8859-15

MIME-Version: 1.0

=A3100 is about =A4150

Presentation

MIME

- or

Content-Transfer-Encoding: base64

Content-Type: text/plain; charset=ISO-8859-15

MIME-Version: 1.0

ozEwMCBpcyBhYm91dCCkMTUwCg=

Presentation

The End of the Line

- It would be easy to think that presentation is easy and is irrelevant or has been solved: not so
- For example: how to represent the end of a line in a text file?

Presentation

The End of the Line

- Unix-derived systems use a linefeed (LF, character 10 in ASCII)
- Windows systems use a carriage return (CR, ASCII 13) followed by a LF
- Pre-MacOS X used a single CR

Presentation

The End of the Line

- So to copy a file from one system to another you must know whether

  a) it is a text file and so you must do the translations, or

  b) it is *not* a text file, so you should *not* translate

Presentation

The End of the Line

- If we are still fumbling an issue as simple as this, just think on the general case!