La sérialisation (suite)

[Mise à jour le 19/12/2021]

1. CBOR

CBOR

- **Vidéo** sur YouTube : **CBOR2**

JSON et CBOR sont tous les deux des modes de codage de la donnée.

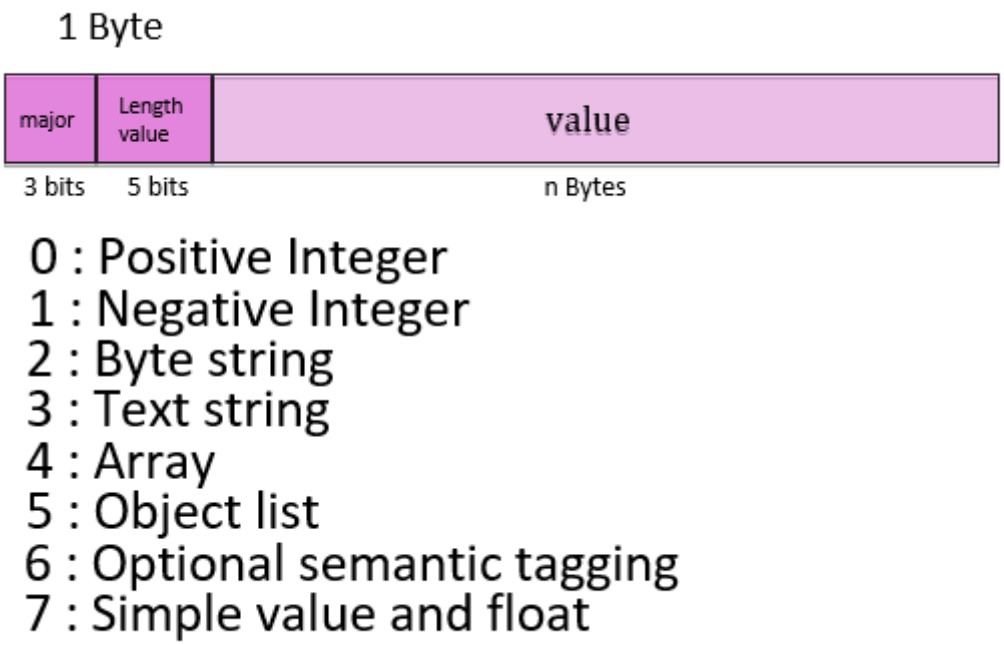
JSON introduit une notation très flexible permettant de représenter toutes les structures de données. Le choix de l'ASCII rend ce format universel et n'importe quel ordinateur pourra le comprendre. Mais l'utilisation de l'ASCII ne permet pas de transmettre de manière optimale l'information sur un réseau. Quand les réseaux ont un débit raisonnable, cela ne pose pas de problème. Quand on en vient à l'internet des objets, il faut prendre en compte la capacité de traitement limité des équipements et la faible taille des messages échangés.

Ainsi, en **ASCII**, la valeur 123 est codée sur 3 octets (un octet par caractère) tandis qu'en binaire elle n'occuperait qu'un seul octet : 0111 1011.

CBOR (**Concise Binaire Object Representation**), défini dans le **RFC 7049**, permet de représenter les structures de JSON, mais suivant une représentation binaire.Si CBOR est complètement compatible avec JSON, il est possible de représenter d'autres types d'information très utile dans l'internet des objets.

La taille de l'information est réduite et le traitement simplifié. Il faut savoir un peu jongler avec la représentation binaire, mais cela reste basique.

CBOR définit **8 types** majeurs qui sont représentés par les **3 premiers bits d'une structure CBOR**. Ces types majeurs ont donc des valeurs comprises entre **0** et **7** (000 à 111 en binaire).



Les **cinq bits suivants** contiennent soit une **valeur** soit une **longueur** indiquant combien d'octets sont nécessaires pour coder la valeur. CBOR offre ainsi des optimisations qui permettent de réduire la longueur totale de la structure des données.

1.1 Type Entier Positif

JSON ne fait pas de différence entre les nombres, entiers, décimaux, positifs ou négatifs. CBOR réintroduit une distinction pour **optimiser la représentation**.

Le premier type majeur correspond aux entiers positifs. Il est codé par **3 bits à 0** ; les 5 bits suivants finissent l'octet et, suivant leur valeur, vont avoir une signification différente :

- de 0 à 23, il s'agit de la valeur de l'entier à coder ;
- 24 indique que l'entier est codé sur 1 octet qui sera codé dans l'octet suivant ;
- 25 indique que l'entier est codé sur 2 octets qui seront codés dans les deux octets suivants ;
- 26 indique que l'entier est codé sur 4 octets qui seront codés dans les quatre octets suivants ;
- 27 indique que l'entier est codé sur 8 octets qui seront codés dans les huit octets suivants.

On peut noter qu'il n'y a pas de surcoût pour coder un entier de 0 à 23. Ainsi, la valeur 15 sera codée 0x0F (000-0 1111) tandis que, pour toutes les autres valeurs supérieures, le surcoût ne sera que d'un octet. 100 sera codé 000-1 1000 (11000 correspond à 24) suivi de la valeur 100 (0110 0100).

Le programme **cbor-integer-ex1.py** va afficher les puissances de 10 entre 10^0 et 10^{18}.

cbor-integer-ex1.py

```
import cbor2 as cbor
```

```
v = 1

for i in range (0, 19):
    c = cbor.dumps(v)
    print ("{0:3} {1:30} {2}".format(i, v, c.hex()))

    v *= 10
```

Le module cbor utilise comme pour JSON la méthode **dumps** pour sérialiser une structure interne de Python dans la représentation demandée. À l'inverse, la méthode **loads** sera utilisée pour **importer** une structure CBOR dans une représentation interne.

La boucle permet d'avoir les multiples de 10 (variable v). Le print en ligne 7 permet d'aligner les données pour que l'affichage soit plus clair ; entre les accolades, le premier chiffre indique la position dans les arguments de format ; le second, après le :, le nombre de caractères.

Par exemple, {1:30} indique l'argument v de format affiché sur 30 caractères.

Le programme donne le résultat suivant :

```
  0                              1 01
  1                             10 0a
  2                            100 1864
  3                           1000 1903e8
  4                          10000 192710
  5                         100000 1a000186a0
  6                        1000000 1a000f4240
  7                       10000000 1a00989680
  8                      100000000 1a05f5e100
  9                     1000000000 1a3b9aca00
 10                    10000000000 1b00000002540be400
 11                   100000000000 1b000000174876e800
 12                  1000000000000 1b000000e8d4a51000
 13                 10000000000000 1b000009184e72a000
 14                100000000000000 1b00005af3107a4000
 15               1000000000000000 1b00038d7ea4c68000
 16              10000000000000000 1b002386f26fc10000
 17             100000000000000000 1b016345785d8a0000
 18            1000000000000000000 1b0de0b6b3a7640000
```

On voit facilement que les valeurs 1 et 10 sont codées sur 1 octet ; que 100 est codé sur 2 octets tandis que les valeurs 1 000 et 10 000 sont codées sur 3 octets. Les valeurs entre 100 000 et 1 000 000 000 nécessitent 5 octets et les suivantes 9 octets.

La taille de la représentation s'adapte à la valeur. Ainsi, il n'est pas nécessaire de définir une taille fixe pour coder une donnée.

On peut aussi noter que comme le type majeur est sur 3 bits, ce type peut être reconnu, car il commence par la valeur “0” ou “1

1.2 Type entier négatif

Le type majeur entier négatif est à peu près similaire à l'entier positif. Le type majeur est 001 et le codage de la valeur se fait sur la valeur absolue du nombre à laquelle on retranche 1. Cela évite deux codes différents pour les valeurs 0 et -0.

Ainsi, pour coder -15, on va coder la valeur 14, ce qui donne en binaire 001-1 1110. Ainsi, -24 peut également être codé sur 1 octet tandis que +24 sera codé sur 2 octets.

Le programme *cbor-integer-ex2.py* reprend le même code, mais pour des puissances de 10 négatives.

cbor-integer-ex2.py

```
# On remplace v = 1 par v = -1 dans le programme précédent.
...
```

Résultat

```
 0                            -1 20
 1                           -10 29
 2                          -100 3863
 3                         -1000 3903e7
 4                        -10000 39270f
 5                       -100000 3a0001869f
 6                      -1000000 3a000f423f
 7                     -10000000 3a0098967f
 8                    -100000000 3a05f5e0ff
 9                   -1000000000 3a3b9ac9ff
10                  -10000000000 3b00000002540be3ff
11                 -100000000000 3b000000174876e7ff
12                -1000000000000 3b000000e8d4a50fff
13               -10000000000000 3b000009184e729fff
14              -100000000000000 3b00005af3107a3fff
15             -1000000000000000 3b00038d7ea4c67fff
16            -10000000000000000 3b002386f26fc0ffff
17           -100000000000000000 3b016345785d89ffff
18          -1000000000000000000 3b0de0b6b3a763ffff
```

1.3 Type Séquence binaire ou Chaîne de caractères

Les séquences binaires et les chaînes de caractères ont le même comportement. Le type majeur est respectivement 010 et 011. Il est suivi par la longueur de la séquence ou de la chaîne. Le même type de codage que pour les entiers est utilisé :

- si la longueur est inférieure à 23, elle est codée dans la suite du premier octet. On trouve ensuite le nombre d'octets ou de caractères correspondant à cette longueur ;
- si la longueur peut être codée dans 1 octet (donc inférieure à 255), la suite du premier octet contient 24 puis l'octet suivant contient la longueur suivie du nombre d'octets ou de caractères

correspondant.
- si la longueur peut être codée dans 2 octets (donc inférieure à 65535), la suite du premier octet contient 25 puis l'octet suivant contient la longueur suivie du nombre d'octets ou de caractères correspondant.
- si la longueur peut être codée dans 4 octets, la suite du premier octet contient 26 puis l'octet suivant contient la longueur suivie du nombre d'octets ou de caractères correspondant.
- si la longueur peut être codée dans 8 octets, la suite du premier octet contient 27 puis l'octet suivant contient la longueur suivie du nombre d'octets ou de caractères correspondant.

Ce codage est aussi assez optimal. Il est rare d'envoyer plus de 23 caractères.

Le programme *cbor-string.py* montre la représentation de chaînes de caractères de longueur croissante ainsi qu'une séquence binaire.

cbor-string.py

```
import cbor2 as cbor

for i in range (1, 10):
    c = cbor.dumps("LoRaWAN"*i)

    print ("{0:3} {1}".format(i, c.hex()))

bs = cbor.dumps(b"\x01\x02\x03")
print (bs.hex())
```

La variable i prend des valeurs de 1 à 9. La multiplication d'une chaîne de caractères par un entier (ligne 4) indique le nombre de répétitions de celle-ci. La variable bs contient la représentation en CBOR d'un tableau d'octets Python (représenté par le caractère “b” avant les guillemets).

Le résultat est le suivant :

```
  1 674c6f526157414e
  2 6e4c6f526157414e4c6f526157414e
  3 754c6f526157414e4c6f526157414e4c6f526157414e
  4 781c4c6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e
  5
78234c6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e
  6
782a4c6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e4c
6f526157414e
  7
78314c6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e4c
6f526157414e4c6f526157414e
  8
78384c6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e4c
6f526157414e4c6f526157414e4c6f526157414e
  9
```

```
783f4c6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e4c
6f526157414e4c6f526157414e4c6f526157414e4c6f526157414e
43010203
```

Jusqu'à 3 répétitions de la chaine de caractères “LoRaWAN”, le codage de la longueur est optimal (codé sur 2 octets).

1.4 Type tableau

Le type tableau va regrouper un ensemble d'éléments. Chacun de ces éléments étant une structure CBOR, la seule information nécessaire pour connaitre le début et la fin d'un tableau est son nombre d'éléments. Le type majeur est **100**. Il existe deux méthodes pour coder la longueur d'un tableau :

- si celle-ci est connue au moment du codage, il suffit de l'indiquer avec un codage identique à celui utilisé pour indiquer la longueur d'une chaîne de caractères ;
- si celle-ci n'est pas connue au moment du codage, il existe un code spécial pour indiquer la fin du tableau.

Le programme *cbor-array.py* donne quelques exemples de codage de tableau :

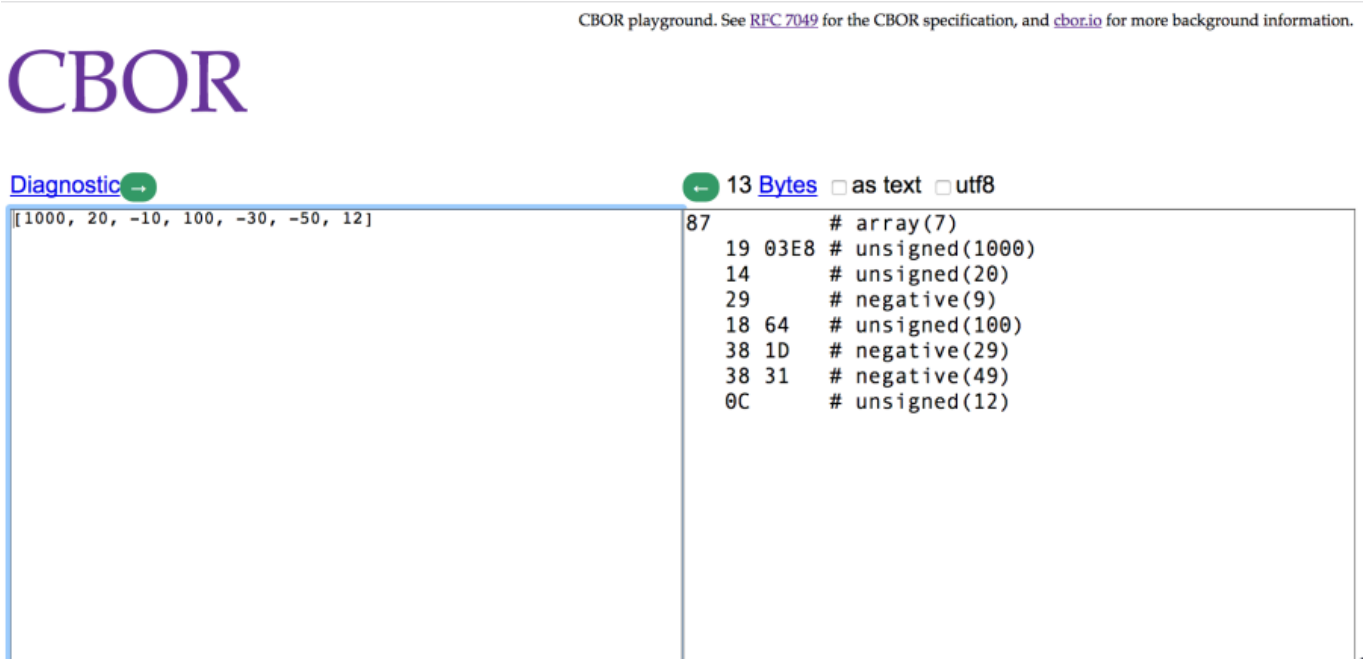
cbor-array.py

```
import cbor2 as cbor

c1 = cbor.dumps([1, 2, 3, 4])
print(c1.hex()) # Résultat : 8401020304 - On peut deviner la structure du message CBOR : 0x84 indique un tableau de 4 éléments (attention le décodage n'est pas toujours aussi simple). Les 4 éléments sont des entiers inférieurs à 23 ;
print()

c2 = cbor.dumps([1, [2, 3], 4])
print(c2.hex()) # Résultat : 830182020304 -  Il s'agit d'un tableau de 3 éléments dont le deuxième est un tableau de deux éléments ;
print()

c3 = cbor.dumps([1000, +20, -10, +100, -30, -50, 12])
print(c3.hex()) # Résultat : 871903e814291864381d38310c
```

Le site web **cbor.me** permet de faire automatiquement le codage dans un sens ou dans l'autre.

La colonne de gauche représente la donnée en JSON et celle de droite en CBOR (dite “représentation canonique” qui facilite la lecture).

En ayant entré la séquence ci-dessus, on trouve l'information suivante :



On peut calculer le degré de compression de CBOR. Ainsi, le tableau JSON [1,2,3,4] est codé sur 9 caractères tandis que la représentation CBOR 8401020304 n'en fait que 5. Pour l'illustration ci dessus, les 13 octets de la représentation CBOR sont transformés en 34 caractères (donc 34 octets) en JSON.

1.5 Type objets (Liste de paires)

Le type **Liste de paires** ou **Dictionnaire** est indiqué par la valeur **101**. Il fonctionne de la même manière que les tableaux en comptant le nombre d'éléments. Mais cette fois-ci, la valeur représente une paire, c'est-à-dire deux objets CBOR.

Le programme *cbor-mapped.py* donne un exemple. La structure à encoder est la suivante :

cbor-mapped.py

```
import cbor2 as cbor

c1 = {"type" : "hamster",
      "taille" : 300,
      2 : "test",
      0x0F: 0b01110001,
      2 : "program"};

print(cbor.dumps(c1).hex())
```

En entrant l'exemple ci-dessus dans le site cbor.me on obtient :

```
A4                  # map(4)
   64               # text(4)
      74797065      # "type"
   67               # text(7)
      68616D73746572 # "hamster"
   66               # text(6)
      7461696C6C65  # "taille"
   19 012C          # unsigned(300)
   02               # unsigned(2)
   67               # text(7)
      70726F6772616D # "program"
   0F               # unsigned(15)
   18 71            # unsigned(113)
```

ou, en notation JSON :

{“type”: “hamster”, “taille”: 300, 2: “program”, 15: 113}

On peut voir des différences entre JSON, CBOR et la représentation des variables en Python. Les codages hexadécimaux et binaires de Python ont été convertis en décimal pour JSON.

De plus, même si JSON n'autorise que des clés en ASCII pour indexer les paires , nous avons pu mettre des clés numériques. Néanmoins Python a effacé la première clé 2 par la dernière. CBOR définit un mode strict dans lequel ces clés doivent être codées en ASCII pour être compatibles avec JSON, mais autorise également des représentations qui diffèrent de JSON en enlevant les contraintes sur les clés.

1.6 Type étiquette

CBOR enrichit le typage des données ; ce qui permet de manipuler plus facilement des données. Par exemple, une chaîne de caractères peut représenter une date, une URI, voire une URI codée en base 64.

Le type **110** peut être suivi d'une valeur dont une liste exhaustive est donnée **ici**.

Par exemple, le programme *cbor-tag.py* affiche la date du jour, la convertit en notation CBOR et la reconvertit en variable Python grâce à la méthode loads.

cbor-tag.py

```
import cbor2 as cbor
from datetime import date, timezone

print (date.today()) # 2018-05-22
c1 = cbor.dumps(date.today(), timezone=timezone.utc,
date_as_datetime=True)

print (c1.hex()) # c074323031382d30352d32325430303a30303a30305a
```

```
print (cbor.loads(c1)) # 2018-05-22 00:00:00+00:00
print (type(cbor.loads(c1))) # <class 'datetime.datetime'>
```

La représentation canonique montre plus facilement le tag dans la séquence binaire :

```
C0                                      # tag(0)
   74                                   # text(20)
      323031382D30352D32325430303A30303A30305A # "2018-05-22T00:00:00Z"
```

Le tag 0 implique un format normalisé pour la date ; d'où l'ajout des heures, minutes et secondes, alors qu'elles n'ont pas été spécifiées initialement.

On peut également remarquer que loads retourne un type date et non une chaîne de caractères.

1.7 Le type flottant et les valeurs particulières

Le dernier type majeur (**111**) permet de coder les nombres flottants en utilisant la représentation définie par l'**IEEE 754**. Suivant la taille de la représentation, la suite de l'octet contient les valeurs 25 (demi précision sur 16 bits), 26 (simple précision sur 32 bits) ou 27 (double précision sur 64 bits).

Ce type permet également de coder les valeurs définies par JSON : True (valeur 20), False (valeur 21) ou None (valeur 22).

Finalement, ce type peut indiquer la fin d'un tableau ou d'une liste de paires quand la taille n'est pas connue au début du codage.

1.8 CBOR sur des exemples

- **Vidéo** Youtube : site **cbor.me**