# RFC 8949 : Concise Binary Object Representation (CBOR)

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 5 décembre 2020

Date de publication du RFC : Décembre 2020

https://www.bortzmeyer.org/8949.html

---

Il existait un zillion de formats binaires d'échange de données ? Et bien ça en fera un zillion plus un. CBOR ("*Concise Binary Object Representation*") est un format qui utilise un modèle de données très proche de celui de JSON, mais est encodé en binaire, avec comme but principal d'être simple à encoder et décoder, même par des machines ayant peu de ressources matérielles. Normalisé à l'origine dans le RFC 7049 [1], il est désormais spécifié dans ce nouveau RFC. Si le texte de la norme a changé, le format reste le même.

Parmi les autres formats binaires courants, on connait ASN.1 (plus exactement BER ou DER, utilisés dans plusieurs protocoles IETF), EBML ou MessagePack mais ils avaient des cahiers des charges assez différents (l'annexe E du RFC contient une comparaison). CBOR se distingue d'abord par sa référence à JSON (RFC 8259), dont le modèle de données sert de point de départ à CBOR, puis par le choix de faciliter le travail des logiciels qui devront créer ou lire du CBOR. CBOR doit pouvoir tourner sur des machines très limitées (« classe 1 », en suivant la terminologie du RFC 7228). Par contre, la taille des données encodées n'est qu'une considération secondaire (section 1.1 du RFC pour une liste prioritisée des objectifs de CBOR). Quant au lien avec JSON, l'idée est d'avoir des modèles de données suffisamment proches pour qu'écrire des convertisseurs CBOR-¿JSON et JSON-¿CBOR soit assez facile, et pour que les protocoles qui utilisent actuellement JSON puissent être adaptés à CBOR sans douleur excessive. CBOR se veut sans schéma, ou, plus exactement, sans schéma obligatoire. Et le but est que les fichiers CBOR restent utilisables pendant des dizaines d'années, ce qui impose d'être simple et bien documenté.

La spécification complète de CBOR est en section 3 de ce RFC. Chaque élément contenu dans le flot de données commence par un octet dont les trois bits de plus fort poids indiquent le **type majeur**. Les cinq bits suivants donnent des détails. Ce mécanisme permet de programmeur un décodeur CBOR avec une table de seulement 256 entrées (l'annexe B fournit cette table et l'annexe C un décodeur en pseudo-code

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc7049.txt

très proche de C). Pour un entier, si la valeur que codent ces cinq bits suivants est inférieure à 24, elle est utilisée telle quelle. Sinon, cela veut dire que les détails sont sur plusieurs octets et qu'il faut lire les suivants (la valeur des cinq bits codant la longueur à lire). Selon le type majeur, les données qui suivent le premier octet sont une valeur (c'est le cas des entiers, par exemple) ou bien un doublet {longueur, valeur} (les chaînes de caractères, par exemple). L'annexe A de notre RFC contient de nombreux exemples de valeurs CBOR avec leur encodage.

Quels sont les types majeurs possibles ? Si les trois premiers bits sont à zéro, le type majeur, 0, est un entier non signé. Si les cinq bits suivants sont inférieurs à 24, c'est la valeur de cet entier. S'ils sont égaux à 24, c'est que l'entier se trouve dans l'octet suivant l'octet initial, s'ils sont égaux à 25, que l'entier se trouve dans les deux octets suivants, et ainsi de suite (31 est réservé pour les tailles indéterminées, décrites plus loin). L'entier 10 se représentera donc 00001010, l'entier 42 sera 00011000 00101010, etc. Presque pareil pour un type majeur de 1, sauf que l'entier sera alors signé, et négatif. La valeur sera -1 moins la valeur encodée. Ainsi, -3 sera 00100010. Vous voulez vérifier ? L'excellent terrain de jeu `<http://cbor.me>` vous le permet, essayez par exemple `<http://cbor.me?diag=42>`.

Le type majeur 2 sera une chaîne d'octets (principal ajout par rapport au modèle de données de JSON). La longueur est codée d'abord, en suivant la même règle que pour les entiers. Puis viennent les données. Le type 3 indique une chaîne de caractères et non plus d'octets. Ce sont forcément des caractères Unicode, encodés en UTF-8 (RFC 3629). Le champ longueur (codé comme un entier) indique le nombre d'octets de l'encodage UTF-8, pas le nombre de caractères (pour connaître ce dernier, il faut un décodeur UTF-8). Vous voulez des exemples ? Connectez-vous à `<http://www.cbor.me/?diag=%22lait%22>` et vous voyez que la chaîne « lait » est représentée par 646c616974 : 64 = 01100100, type majeur 3 puis une longueur de 4. Les codes ASCII suivent (rappelez-vous qu'ASCII est un sous-ensemble d'UTF-8). Avec des caractères non-ASCII comme `<http://www.cbor.me/?diag=%22caf%C3%A9%22>`, on aurait 65636166c3a9 (même type majeur, longueur 5 **octets**, puis les caractères, avec c3a9 qui code le é en UTF-8).

Le type majeur 4 indique un tableau. Rappelez-vous que CBOR utilise un modèle de données qui est très proche de celui de JSON. Les structures de données possibles sont donc les tableaux et les objets (que CBOR appelle les "*maps*"). Un tableau est encodé comme une chaîne d'octets, longueur (suivant les règles des entiers) puis les éléments du tableau, à la queue leu leu. La longueur est cette fois le nombre d'éléments, pas le nombre d'octets. Les éléments d'un tableau ne sont pas forcément tous du même type. Les tableaux (et d'autres types, cf. section 3.2) peuvent aussi être représentés sans indiquer explicitement la longueur ; le tableau est alors terminé par un élément spécial, le "*break code*". Par défaut, un encodeur CBOR est libre de choisir la forme à longueur définie ou celle à longueur indéfinie, et le décodeur doit donc s'attendre à rencontrer les deux.

Le type majeur 5 indique une "*map*" (ce qu'on appelle objet en JSON et dictionnaire ou "*hash*" dans d'autres langages). Chaque élément d'une "*map*" est un doublet {clé, valeur}. L'encodage est le même que pour les tableaux, la longueur étant le nombre de doublets. Chaque doublet est encodé en mettant la clé, puis la valeur. Donc, le premier scalaire est la clé de la première entrée de la "*map*", le deuxième la valeur de la première entrée, le troisième la clé de la deuxième entrée, etc.

Les clés doivent être uniques (une question problématique en JSON où les descriptions existantes de ce format ne sont ni claires ni cohérentes sur ce point).

Je passe sur le type majeur 6, voyez plus loin le paragraphe sur les étiquettes. Le type majeur 7 sert à coder les flottants (encodés ensuite en IEEE 754) et aussi d'autres types scalaires et le "*break code*" utilisé dans le paragraphe suivant. Les autres types scalaires, nommés « valeurs simples » ("*simple values*") sont des valeurs spéciales comme 20 pour le booléen Faux, 21 pour le Vrai, et 22 pour le néant `<https://www.bortzmeyer.org/exprimer-neant.html>`. Elles sont stockées dans un registre

IANA <https://www.iana.org/assignments/cbor-simple-values/cbor-simple-values.xml#simple>.

Dans la description ci-dessus, les types vectoriels (tableaux, chaînes, "*maps*") commencent par la longueur du vecteur. Pour un encodeur CBOR, cela veut dire qu'il faut connaître cette longueur avant même d'écrire le premier élément. Cela peut être contraignant, par exemple si on encode au fil de l'eau ("*streaming*") des données en cours de production. CBOR permet donc d'avoir des longueurs indéterminées. Pour cela, on met 31 comme « longueur » et cette valeur spéciale indique que la longueur n'est pas encore connue. Le flot des éléments devra donc avoir une fin explicite cette fois, le "*break code*". Celui-ci est représenté par un élément de type majeur 7 et de détails 31, donc tous les bits de l'octet à 1. Par exemple, <http://cbor.me/?diag=%28_%20%22lait%22%29> nous montre que la chaîne « lait » ainsi codée (le _ indique qu'on veut un codage en longueur indéterminée) sera 7f646c616974ff. 7f est le type majeur 3, chaîne de caractères, avec la longueur 31, indiquant qu'elle est indéterminée. Puis suit la chaîne elle-même (les chaînes indéterminées en CBOR sont faites par concaténation de châines de longueur déterminée), puis le "*break code*" ff.

La même technique peut être utilisée pour les chaînes d'octets et de caractères, afin de ne pas avoir à spécifier leur longueur au début. Cette possibilité de listes de longueur indéterminée a été ajoutée pour faciliter la vie du "*streaming*".

Revenons au type majeur 6. Il indique une étiquette ("*tag*"), qui sert à préciser la sémantique de l'élément qui suit (cf. section 3.4). Un exemple typique est pour indiquer qu'une chaîne de caractères est un fait une donnée structurée, par exemple une date ou un numéro de téléphone. Un décodeur n'a pas besoin de comprendre les étiquettes, il peut parfaitement les ignorer. Les valeurs possibles pour les étiquettes sont stockées dans un registre IANA <https://www.iana.org/assignments/cbor-tags/cbor-tags.xml#tags>, et on voit que beaucoup ont déjà été enregistrées, en plus de celles de ce RFC (par exemple par le RFC 8746 ou par le RFC 9132).

Quelques valeurs d'étiquette intéressantes ? La valeur 0 indique une date au format du RFC 3339 (une chaîne de caractères). La valeur 1 étiquette au contraire un entier, et indique une date comme un nombre de secondes depuis le 1er janvier 1970. Les valeurs négatives sont autorisées mais leur sémantique n'est pas normalisée (UTC n'est pas défini pour les dates avant l'"*epoch*", surtout quand le calendrier a changé).

Les valeurs 2 et 3 étiquettent une chaîne d'octets et indiquent qu'on recommande de l'interpréter comme un grand entier (dont la valeur n'aurait pas tenu dans les types majeurs 0 ou 1). Les décodeurs qui ne gèrent pas les étiquettes se contenteront de passer à l'application cette chaîne d'octets, les autres passeront un grand entier.

Autre cas rigolos, les nombres décimaux non entiers. Certains ne peuvent pas être représentés de manière exacte sous forme d'un flottant. On peut alors les représenter par un couple [exposant, mantisse]. Par exemple, 273,15 est le couple [-2, 27315] (l'exposant est en base 10). On peut donc l'encoder en CBOR sous forme d'un tableau de deux éléments, et ajouter l'étiquette de valeur 4 pour préciser qu'on voulait un nombre unique.

D'autres étiquettes précisent le contenu d'une chaîne de caractères : l'étiquette 32 indique que la chaîne est un URI, la 34 que la chaîne est du Base64 (RFC 4648) et la 36 que cela va être un message MIME (RFC 2045). Comme l'interprétation des étiquettes est optionnelle, un décodeur CBOR qui n'a pas envie de s'embêter peut juste renvoyer à l'application cette chaîne.

Une astuce amusante pour finir les étiquettes, et la spécification du format : l'étiquette 55799 signifie juste que ce qui suit est du CBOR, sans modifier sa sémantique. Encodée, elle sera représentée par 0xd9d9f7 (type majeur 6 sur trois bits, puis détails 25 qui indiquent que le nombre est sur deux octets puis le nombre lui-même, d9f7 en hexa). Ce nombre 0xd9d9f7 peut donc servir de nombre magique. Si on le trouve au début d'un fichier, c'est probablement du CBOR (il ne peut jamais apparaître au début d'un fichier JSON, donc ce nombre est particulièrement utile quand on veut distinguer tout de suite si on a affaire à du CBOR ou à du JSON).

Maintenant que le format est défini rigoureusement, passons à son utilisation. CBOR est conçu pour des environnements où il ne sera souvent pas possible de négocier les détails du format entre les deux parties. Un décodeur CBOR générique peut décoder sans connaître le schéma utilisé en face. Mais, en pratique, lorsqu'un protocole utilise CBOR pour la communication, il est autorisé (section 5 du RFC) à mettre des restrictions, ou des informations supplémentaires, afin de faciliter la mise en œuvre de CBOR dans des environnements très contraints en ressources. Ainsi, on a parfaitement le droit de faire un décodeur CBOR qui ne gérera pas les nombres flottants, si un protocole donné n'en a pas besoin.

Un cas délicat est celui des "*maps*" (section 5.6). CBOR ne place guère de restrictions sur le type des clés et un protocole ou format qui utilise CBOR voudra souvent être plus restrictif. Par exemple, si on veut absolument être compatible avec JSON, restreindre les clés à des chaînes en UTF-8 est souhaitable. Si on tient à utiliser d'autres types pour les clés (voire des types différents pour les clés d'une même "*map*" !), il faut se demander comment on les traduira lorsqu'on enverra ces "*maps*" à une application. Par exemple, en JavaScript, la clé formée de l'entier 1 est indistinguable de celle formée de la chaîne de caractères "1". Une application en JavaScript ne pourra donc pas se servir d'une "*map*" qui aurait de telles clés, de types variés.

On a vu que certains éléments CBOR pouvaient être encodés de différentes manières, par exemple un tableau peut être représenté par {longueur, valeurs} ou bien par {valeurs, "*break code*"}. Cela facilite la tâche des encodeurs mais peut compliquer celle des décodeurs, surtout sur les machines contraintes en ressources, et cela peut rendre certaines opérations, comme la comparaison de deux fichiers, délicates. Existe-t-il une forme canonique de CBOR ? Pas à proprement parler mais la section 4.1 décrit la notion de sérialisation favorite, des décodeurs étant autorisés à ne connaitre qu'une sérialisation possible. Notamment, cela implique pour l'encodeur de :

— Mettre les entiers sous la forme la plus compacte possible. L'entier 2 peut être représenté par un octet (type majeur 0 puis détails égaux à 2) ou deux (type majeur 0, détails à 24 puis deux octets contenant la valeur 2), voire davantage. La forme recommandée est la première (un seul octet). Même règle pour les longueurs (qui, en CBOR, sont encodées comme les entiers).
— Autant que possible, mettre les tableaux et les chaînes sous la forme {longueur, valeurs}, et pas sous la forme où la longueur est indéfinie.

Tous les encodeurs CBOR qui suivent ces règles produiront, pour un même jeu de données, le même encodage.

Plus stricte est la notion de sérialisation déterministe de la section 4.2. Là encore, chacun est libre de la définir comme il veut (il n'y a pas de forme canonique officielle de CBOR, rappelez-vous) mais elle ajoute des règles minimales à la sérialisation favorite :

— Trier les clés d'une "*map*" de la plus petite à la plus grande. (Selon leur représentation en octets, pas selon l'ordre alphabétique.)
— Ne jamais utiliser la forme à longueur indéfinie des tableaux et chaînes.
— Ne pas utiliser les étiquettes si elles ne sont pas nécessaires.

Autre question pratique importante, le comportement en cas d'erreurs. Que doit faire un décodeur CBOR si deux clés sont identiques dans une "*map*", ce qui est normalement interdit en CBOR ? Ou si un champ longueur indique qu'on va avoir un tableau de 5 éléments mais qu'on n'en rencontre que 4 avant la fin du fichier ? Ou si une chaîne de caractères, derrière son type majeur 3, n'est pas de l'UTF-8 correct ? D'abord, un point de terminologie important : un fichier CBOR est **bien formé** si sa syntaxe est bien celle de CBOR, il est **valide** s'il est bien formé **et** que les différents éléments sont conformes à leur sémantique (par exemple, la date après une étiquette de valeur 0 doit être au format du RFC 3339). Si le document n'est pas bien formé, ce n'est même pas du CBOR et doit être rejeté par un décodeur. S'il n'est pas valide, il peut quand même être utile, par exemple si l'application fait ses propres contrôles. Les sections 5.2 et 5.3 décrivent la question. CBOR n'est pas pédant : un décodeur a le droit d'ignorer certaines erreurs, de remplacer les valeurs par ce qui lui semble approprié. CBOR penche nettement du côté « être indulgent avec les données reçues `<https://www.bortzmeyer.org/principe-robustesse.html>` » ; il faut dire qu'une application qui utilise CBOR peut toujours le renforcer en ajoutant l'obligation de rejeter ces données erronées. Un décodeur strict peut donc s'arrêter à la première erreur. Ainsi, un pare-feu qui analyse du CBOR à la recherche de contenu malveillant a tout intérêt à rejeter les données CBOR incorrectes (puisqu'il ne sait pas trop comment elles seront interprétées par la vraie application). Bref, la norme CBOR ne spécifie pas de traitement d'erreur unique. Je vous recommande la lecture de l'annexe C, qui donne en pseudo-code un décodeur CBOR minimum qui ne vérifie pas la validité, uniquement le fait que le fichier est bien formé, et l'annexe F, qui revient sur cette notion de « bien formé » et donne des exemples.

Comme CBOR a un modèle de données proche de celui de JSON, on aura souvent envie d'utiliser CBOR comme encodage efficace de JSON. Comment convertir du CBOR en JSON et vice-versa sans trop de surprises ? La section 6 du RFC se penche sur ce problème. Depuis CBOR vers JSON, il est recommandé de produire du I-JSON (RFC 7493). Les traductions suivantes sont suggérées :

— Les entiers deviennent évidemment des nombres JSON.
— Les chaînes d'octets sont encodées en Base64 et deviennent des chaînes de caractères JSON (JSON n'a pas d'autre moyen de transporter du binaire).
— Les chaînes de caractères deviennent des chaînes de caractères JSON (ce qui nécessite d'en échapper certains, RFC 8259, section 7).
— Les tableaux deviennent des tableaux JSON et les "*maps*" des objets JSON (ce qui impose de convertir les clés en chaînes UTF-8, si elles ne l'étaient pas déjà).
— Etc.

En sens inverse, de JSON vers CBOR, c'est plus simple, puisque JSON n'a pas de constructions qui seraient absentes de CBOR.

Pour les amateurs de futurisme, la section 7 discute des éventuelles évolutions de CBOR. Pour les faciliter, CBOR a réservé de la place dans certains espaces. Ainsi, le type majeur 7 permettra d'encoder encore quelques valeurs simples (cela nécessitera un RFC sur le chemin des normes, cf. RFC 8126 et la section 9.1 de notre RFC). Et on peut ajouter d'autres valeurs d'étiquettes (selon des règles qui dépendent de la valeur numérique : les valeurs les plus faibles nécessiteront une procédure plus complexe, cf. section 9.2).

CBOR est un format binaire. Cela veut dire, entre autres, qu'il n'est pas évident de montrer des valeurs CBOR dans, mettons, une documentation, contrairement à JSON. La section 8 décrit donc un format texte (volontairement non spécifié en détail) qui permettra de mettre des valeurs CBOR dans du texte. Nulle grammaire formelle pour ce format de diagnostic : il est prévu pour l'utilisation par un humain, pas par un analyseur syntaxique. Ce format ressemble à JSON avec quelques extensions pour les nouveautés de CBOR. Par exemple, les étiquettes sont représentées par un nombre suivi d'une valeur entre parenthèses. Ainsi, la date (une chaîne de caractères étiquetée par la valeur 0) sera notée :

```
0("2013-10-12T11:34:00Z")
```

Une "*map*" de deux éléments sera notée comme en JSON :

```
{"Fun": true, "Amt": -2}
```

Même chose pour les tableaux. Ici, avec étiquette sur deux chaînes de caractères :

```
[32("http://cbor.io/"), 34("SW5zw6lyZXogaWNpIHVuIMWTdWYgZGUgUMOicXVlcw==")]
```

L'annexe G du RFC 8610 ajoute quelques extensions utiles à ce format de diagnostic.

Lors de l'envoi de données encodées en CBOR, le type MIME à utiliser sera `application/cbor`. Comme l'idée est d'avoir des formats définis en utilisant la syntaxe CBOR et des règles sémantiques spécifiques, on verra aussi sans doute des types MIME utilisant la notation plus du RFC 6839, par exemple `application/monformat+cbor`.

Voici par exemple un petit service Web qui envoie la date courante en CBOR (avec deux étiquettes différentes, celle pour les dates au format lisible et celle pour les dates en nombre de secondes, et en prime celles du RFC 8943). Il a été réalisé avec la bibliothèque flunn <`https://github.com/funny-falcon/flunn`>. Il utilise le type MIME `application/cbor` :

```
% curl -s https://www.bortzmeyer.org/apps/date-in-cbor  | read-cbor -
...
Tag 0
String of length 20: 2020-11-16T15:02:24Z
Tag 1
Unsigned integer 1605538944
...
```

(Le programme `read-cbor` est présenté plus loin.)

Un petit mot sur la sécurité (section 10) : il est bien connu qu'un analyseur mal écrit est un gros risque de sécurité et d'innombrables attaques ont déjà été réalisées en envoyant à la victime un fichier délibérement incorrect, conçu pour déclencher une faille de l'analyseur. Ainsi, en CBOR, un décodeur qui lirait une longueur, puis chercherait le nombre d'éléments indiqué, sans vérifier qu'il est arrivé au bout du fichier, pourrait déclencher un débordement de tampon. Les auteurs de décodeurs CBOR sont donc priés de programmer de manière défensive, voire paranoïaque : ne faites pas confiance au contenu venu de l'extérieur.

Autre problème de sécurité, le risque d'une attaque par déni de service. Un attaquant taquin peut envoyer un fichier CBOR où la longueur d'un tableau est un très grand nombre, dans l'espoir qu'un analyseur naïf va juste faire `malloc(length)` sans se demander si cela ne consommera pas toute la mémoire.

Enfin, comme indiqué plus haut à propos du traitement d'erreur, comme CBOR ne spécifie pas de règles standard pour la gestion des données erronées, un attaquant peut exploiter cette propriété pour faire passer des données « dangereuses » en les encodant de telle façon que l'IDS n'y voit que du feu. Prenons par exemple cette "*map*" :

```
{"CodeToExecute": "OK",
 "CodeToExecute": "DANGER"}
```

Imaginons qu'une application lise ensuite la donnée indexée par `CodeToExecute`. Si, en cas de clés dupliquées, elle lit la dernière valeur, elle exécutera le code dangereux. Si un IDS lit la première valeur, il ne se sera pas inquiété. Voilà une bonne raison de rejeter du CBOR invalide (les clés dupliquées sont interdites) : il peut être interprété de plusieurs façons. Notez quand même que ce problème des clés dupliquées, déjà présent en JSON, a suscité des discussions passionnées à l'IETF, entre ceux qui réclamaient une interdiction stricte et absolue et ceux qui voulaient laisser davantage de latitude aux décodeurs. (La section 5.6 est une bonne lecture ici.)

Pour les amateurs d'alternatives, l'annexe E du RFC compare CBOR à des formats analogues. Attention, la comparaison se fait à la lumière du cahier des charges de CBOR, qui n'était pas forcément le cahier des charges de ces formats. Ainsi, ASN.1 (ou plutôt ses sérialisations comme BER ou DER, PER étant nettement moins courant puisqu'il nécessite de connaître le schéma des données) est utilisé par plusieurs protocoles IETF (comme LDAP) mais le décoder est une entreprise compliquée.

MessagePack est beaucoup plus proche de CBOR, dans ses objectifs et ses résultats, et a même été le point de départ du projet CBOR. Mais il souffre de l'absence d'extensibilité propre. Plusieurs propositions d'extensions sont restées bloquées à cause de cela.

BSON (connu surtout via son utilisation dans MongoDB) a le même problème. En outre, il est conçu pour le stockage d'objets JSON dans une base de données, pas pour la transmission sur le réseau (ce qui explique certains de ses choix). Enfin, MSDTP, spécifié dans le RFC 713, n'a jamais été réellement utilisé.

Rappelez-vous que CBOR prioritise la simplicité de l'encodeur et du décodeur plutôt que la taille des données encodées. Néanmoins, un tableau en annexe E.5 compare les tailles d'un même objet encodé avec tous ces protocoles : BSON est de loin le plus bavard (BER est le second), MessagePack et CBOR les plus compacts.

Une liste des implémentations est publiée en `<https://cbor.io/>`. Au moins quatre existent, en Python, Ruby, JavaScript et Java. J'avais moi-même écrit un décodeur CBOR très limité (pour un besoin ponctuel `<https://www.bortzmeyer.org/c-dns-tests.html>`) en Go. Il est disponible ici (en ligne sur `https://www.bortzmeyer.org/files/read-cbor.go`) et son seul rôle est d'afficher le CBOR sous forme arborescente, pour aider à déboguer un producteur de CBOR. Cela donne quelque chose du genre :

```
% ./read-cbor test.cbor
Array of 3 items
String of length 5: C-DNS
Map of 4 items
Unsigned integer 0
 =>  Unsigned integer 0
Unsigned integer 1
 =>  Unsigned integer 5
Unsigned integer 4
 =>  String of length 70: Experimental dnstap client, IETF 99 hackathon, data from unbound 1.6.4
Unsigned integer 5
 =>  String of length 5: godin
Array of indefinite number of items
Map of 3 items
Unsigned integer 0
 =>  Map of 1 items
Unsigned integer 1
```

```
 =>  Array of 2 items
Unsigned integer 1500204267
Unsigned integer 0
Unsigned integer 2
 =>  Map of indefinite number of items
Unsigned integer 0
 =>  Array of 2 items
Byte string of length 16
Byte string of length 16
...
```

L'annexe G de notre RFC résume les changements depuis le RFC 7049. Le format reste le même, les fichiers CBOR d'avant sont toujours du CBOR. Il y a eu dans ce nouveau RFC des corrections d'erreurs (comme un premier exemple erroné <https://www.rfc-editor.org/errata/eid3764>, un autre <https://www.rfc-editor.org/errata/eid3770>, et encore un <https://www.rfc-editor.org/errata/eid5917>), un durcissement des conditions d'enregistrement des nouvelles étiquettes pour les valeurs les plus basses, une description plus détaillée du modèle de données (la section 2 est une nouveauté de notre RFC), un approfondissement des questions de représentation des nombres, etc. Notre RFC 8949 est également plus rigoureux sur les questions de sérialisation préférée et déterministe. L'étiquette 35, qui annonçait qu'on allait rencontrer une expression rationnelle a été retirée (le RFC note qu'il existe plusieurs normes pour ces expressions et que l'étiquette n'est pas définie de manière assez rigoureuse pour trancher).