Groupe de travail Réseau
Request for Comments : 4506
STD 67
RFC rendue obsolète : 1832
Catégorie : Norme

M. Eisler, éditeur.
Network Appliance, Inc.
mai 2006

Traduction Claude Brière de L'Isle

XDR : Norme de représentation des données externes

Statut du présent mémoire
Le présent document spécifie un protocole de normalisation Internet pour la communauté Internet, et appelle à discussion et suggestions en vue de son amélioration. Prière de se reporter à l'édition en cours des "Internet Official Protocol Standards" (normes officielles du protocole Internet) (STD 1) pour connaître l'état de la normalisation et le statut du présent protocole. La distribution du présent mémoire n'est soumise à aucune restriction.

Résumé
Le présent document décrit le protocole de la norme de représentation des données externes (XDR, *External Data Representation*) tel qu'il est actuellement développé et accepté. Il rend obsolète la RFC 1832.

Table des matières

1. Introduction

XDR est une norme de description et de codage des données. Elle est utile pour le transfert des données entre des architectures informatiques différentes, et a été utilisée pour communiquer des données entre des machines aussi diverses que les SUN WORKSTATION*, VAX*, IBM-PC*, et Cray*. XDR rentre dans la couche ISO de présentation des données et est en gros d'un objet analogue à la notation de syntaxe abstraite X.409/ISO. La différence majeure entre elles est que XDR utilise des types implicites, alors que X.409 utilise des types explicites.

XDR utilise un langage pour décrire les formats de données. Le langage ne peut être utilisé que pour décrire des données ; ce n'est pas un langage de programmation. Ce langage permet de décrire des formats de données imbriqués de manière concise. La solution de remplacement consistant à utiliser des représentations graphiques (elles-mêmes dans un langage informel) devient rapidement incompréhensible en face de situations complexes. Le langage XDR lui-même est similaire au langage C [KERN], juste comme Courier [COUR] est similaire à Mesa. Des protocoles comme celui d'appel de procédure à distance (RPC, *Remote Procedure Call*) d'ONC et le NFS* (Network File System, *système de fichier réseau*) utilisent XDR pour décrire le format de leurs données.

La norme XDR fait les hypothèses suivantes : que les octets sont portables, un octet étant défini comme 8 bits de données. Un matériel informatique donné devrait coder les octets sur les divers supports de façon telle que les autres matériels puissent décoder les octets sans perte de signification. Par exemple, la norme Ethernet* suggère que les octets soient codés en style "petit boutien" [COHE], c'est à dire le bit de moindre poids en premier.

2. Changements par rapport à la RFC 1832

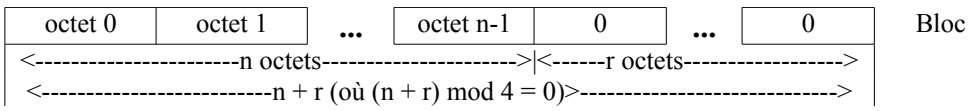
Le présent document n'apporte pas de changement technique à la RFC 1832 et il est publié afin de noter les considérations relatives à l'IANA, d'augmenter les considérations pour la sécurité, et de distinguer les références normatives des références pour information.

3. Taille de bloc de base

La représentation de tous les items requiert un multiple de quatre octets (ou 32 bits) de données. Les octets sont numérotés de 0 à n-1. Les octets sont lus ou écrits dans un flux d'octets de telle sorte que l'octet m précède toujours l'octet m+1. Si les n octets nécessaires pour contenir les données ne sont pas un multiple de quatre, les n octets sont alors suivis par suffisamment (de 0 à 3) d'octets résiduels à zéro, r, pour faire du compte total d'octets un multiple de 4.

On inclut la notation familière du graphique de boîtes pour illustrer et comparer. Dans la plupart des illustrations, chaque boîte (délimitée par des traits continus) décrit un octet.

Les points de suspension (...) entre les boîtes montrent zéro, un ou plusieurs octets additionnels si nécessaire.



4. Types de données XDR

Chacun des paragraphes qui suivent décrit un type de données défini dans la norme XDR, comment il est déclaré dans le langage, et comporte une illustration graphique de son codage.

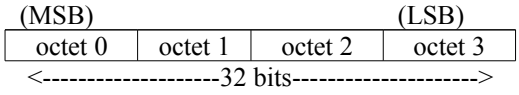
Pour chaque type de données dans le langage, on donne une déclaration générale du paradigme. Noter que les crochets angulaires (< et >) notent des séquences de longueur variable de données et que les crochets rectangulaires ([et]) notent des séquences de données de longueur fixe. "n", "m", et "r" notent des entiers. Pour la spécification complète du langage et des définitions plus formelles des termes comme "identifiant" et "déclaration", se reporter à la Section 6 "Spécification du langage XDR".

Pour certains types de données, des exemples plus spécifiques sont inclus. Un exemple plus complet de description des données est donné à la Section 7 "Exemple d'une description de données XDR".

4.1 Entier

Un entier signé XDR est une donnée de 32 bits qui code un entier dans la gamme [-2 147 483 648, 2 147 483 647]. L'entier est représenté en notation de complément à deux. Les octets de plus fort poids et de moindre poids sont respectivement les octets 0 et 3. Les entiers sont déclarés comme suit :

```
identifiant d'entier ;
```

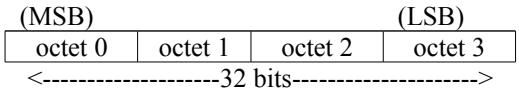


Entier

4.2 Entier non signé

Un entier XDR non signé est une donnée de 32 bits qui code un entier non négatif dans la gamme de [0, 4 294 967 295]. Il est représenté par un nombre binaire non signé dont les octets de plus fort poids et de moindre poids sont respectivement les octets 0 et 3. Un entier non signé est déclaré comme suit :

```
identifiant d'entier non signé ;
```



Entier non signé

4.3 Énumération

Les énumérations ont la même représentation que les entiers signés.
Les énumérations sont commodes pour décrire des sous-ensembles d'entiers.
Les données énumérées sont déclarées comme suit :

```
enum { identifiant-de-nom = constante, ... } identifiant ;
```

Par exemple, les trois couleurs rouge, jaune, et bleu pourraient être décrites par un type énuméré :

```
enum { ROUGE = 2, JAUNE = 3, BLEU = 5 } couleurs ;
```

C'est une erreur de coder comme enum tout entier autre que ceux qui ont reçu une allocation dans la déclaration d'énumération.

4.4 Booléen

Les booléens sont assez importants et surviennent assez fréquemment pour affirmer leur propre type explicite dans la norme. Les booléens sont déclarés comme suit :

```
identifiant de booléen ;
```

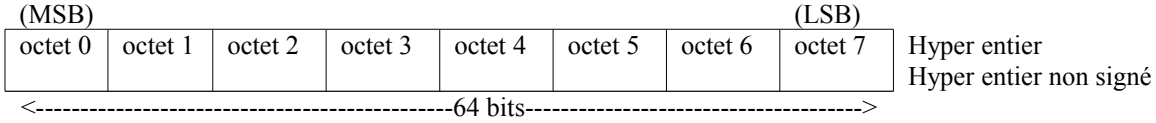
Ceci est équivalent à :

```
enum { FAUX = 0, VRAI = 1 } identifiant;
```

4.5 Hyper-entier et hyper-entier non signé

La norme définit aussi des nombres de 64 bits (8 octets) appelés hyper entiers et hyper entiers non signés. Leur représentation est l'extension évidente des entiers et des entiers non signés définis ci-dessus. Ils sont représentés en notation de complément à deux. Les octets de plus fort poids et de moindre poids sont respectivement les octets 0 et 7. Leur déclaration est :

identifiant d'hyper entier ; identifiant d'hyper entier non signé ;



4.6 Virgule flottante

La norme définit le type de donnée virgule flottante "float" (32 bits ou 4 octets). Le codage utilisé est la norme IEEE pour les nombres normalisés à virgule flottante en précision simple [IEEE]. Les trois champs suivants décrivent le nombre à virgule flottante en précision simple :
S : Le signe du nombre. Les valeurs 0 et 1 représentent respectivement le positif et le négatif. Un bit.
E : L'exposant du nombre, en base 2. 8 bits sont alloués à ce champ. L'exposant est biaisé de 127.
F : La partie décimale de la mantisse du nombre, en base 2. 23 bits sont alloués à ce champ.

Donc, le nombre à virgule flottante est décrit par :

(-1)**S * 2**(E-Bias) * 1.F

Il est déclaré comme suit :

identifiant de nombre à virgule flottante ;

octet 0	octet 1	octet 2	octet 3	
S	E	F		Nombre à virgule flottante en précision simple
1	<- 8 ->	<-------23 bits------>		
<----------32 bits---------->				

Tout comme les octets de poids fort et de moindre poids d'un nombre sont 0 et 3, les bits de poids fort et de moindre poids d'un nombre à virgule flottante en précision simple sont 0 et 31. Les décalages du bit de début (qui est le bit de poids fort) de S, E, et F sont, respectivement, 0, 1, et 9. Noter que ces nombres se réfèrent aux positions mathématiques des bits, et NON à leur localisation physique réelle (qui varie selon le support).

Les spécifications de l'IEEE devraient être consultées au sujet du codage du zéro signé, de l'infini signé *(overflow)*, et des nombres dénormalisés *(underflow)* [IEEE]. Selon les spécifications de l'IEEE, le "NaN" (pas un nombre, *Not a Number*) dépend du système et ne devrait pas être interprété au sein de XDR comme autre chose que "NaN".

4.7 Virgule flottante à double précision

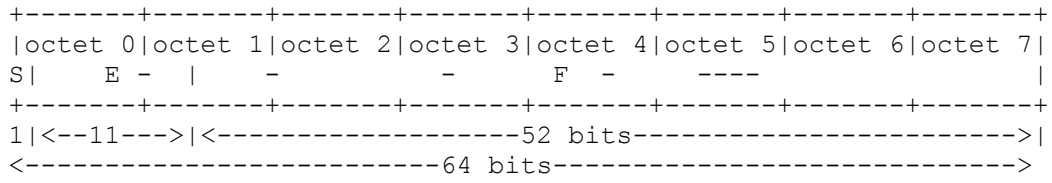
La norme définit le codage pour le type de donnée à virgule flottante à double précision "double" (64 bits ou 8 octets). Le codage utilisé est celui de la norme IEEE pour les nombres à virgule flottante à double précision normalisé [IEEE]. La norme code les trois champs suivants, qui décrivent le nombre à virgule flottante à double précision :
S : Le signe du nombre. Les valeurs 0 et 1 représentent respectivement le positif et le négatif. Un bit.
E : L'exposant du nombre, en base 2. 11 bits sont dédiés à ce champ. L'exposant est biaisé de 1023.
F : La partie fractionnaire de la mantisse du nombre, en base 2. 52 bits sont dédiés à ce champ.

Donc, le nombre à virgule flottante est décrit par :

(-1)**S * 2**(E-Bias) * 1.F

Il est déclaré comme suit :

identifiant double ;



Virgule flottante à double précision

Tout comme les octets de poids fort et de moindre poids d'un nombre sont 0 et 3, les bits de poids fort et de moindre poids d'un nombre à virgule flottante à double précision sont 0 et 63. Les décalages du bit de début (bit de poids fort) de S, E, et F sont respectivement 0, 1, et 12. Noter que ces nombres se réfèrent aux positions mathématiques des bits, et NON à leur localisation physique réelle (qui varie d'un support à l'autre).

Les spécifications IEEE devraient être consultées pour ce qui concerne le codage du zéro signé, de l'infini signé (débordement), et des nombres dénormalisés (dépassement de capacité négatif) [IEEE]. Selon les spécifications de l'IEEE, le "NaN" (pas un nombre, *not a number*) dépend du système et ne devrait pas être interprété dans XDR comme quelque chose d'autre que "NaN".

4.8 Virgule flottante à quadruple précision

La norme définit le codage pour le type de données à virgule flottante à quadruple précision "quadruple" (128 bits ou 16 octets). Le codage utilisé est conçu comme une simple analogie du codage utilisé pour les nombres à virgule flottante en simple et double précision qui utilisent une forme de la double précision étendue de l'IEEE. La norme code les trois champs suivants, qui décrivent le nombre à virgule flottante à quadruple précision :

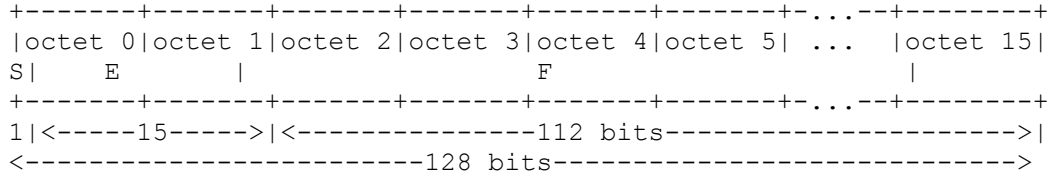
S : Le signe du nombre. Les valeurs 0 et 1 représentent respectivement le positif et le négatif. Un bit.
E : L'exposant du nombre, en base 2. 15 bits sont dédiés à ce champ. L'exposant a un biais de 16383.
F : La partie fractionnaire de la mantisse du nombre, en base 2. 112 bits sont dédiés à ce champ.

Donc, le nombre à virgule flottante est décrit par :

(-1)**S * 2**(E-Biais) * 1.F

Il est déclaré comme suit :

identifiant quadruple ;



Virgule flottante à quadruple précision

Tout comme les octets de poids fort et de moindre poids d'un nombre sont 0 et 3, les bits de poids fort et de moindre poids d'un nombre à virgule flottante à quadruple précision sont 0 et 127. Les décalages du bit de début (bit de poids fort) de S, E, et F sont respectivement 0, 1, et 16. Noter que ces nombres se réfèrent aux positions mathématiques des bits, et NON à leur localisation physique réelle (qui varie d'un support à l'autre).

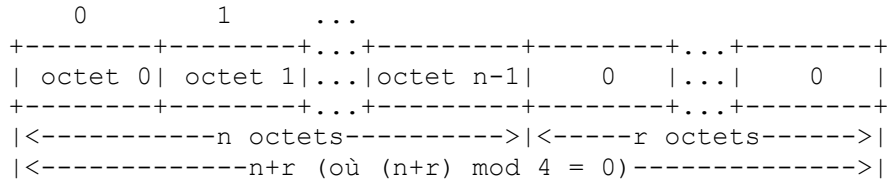
Le codage pour le zéro signé, l'infini signé (débordement) et les nombres dénormalisés est analogue au codage correspondant des nombres à virgule flottante à simple et double précision [SPAR], [HPRE]. Le codage "NaN" tel qu'appliqué aux nombres à virgule flottante à quadruple précision dépend du système et ne devrait pas être interprété au sein de XDR comme quelque chose d'autre que "NaN".

4.9 Données opaques de longueur fixe

Parfois, des données non interprétées de longueur fixe doivent être passées dans les machines. Ces données sont appelées "opaques" et sont déclarée comme suit :

opaque identifiant[n];

où la constante n est le nombre (statique) d'octets nécessaires pour contenir les données opaques. Si n n'est pas un multiple de quatre, les n octets sont alors suivis par assez (0 à 3) d'octets zéro résiduels, r, pour faire du compte total d'octet de l'objet opaque un multiple de quatre.



Données opaques de longueur fixe

4.10 Données opaques de longueur variable

La norme traite aussi des données opaques de longueur variable (comptée), définies comme une séquence de n (numérotés de 0 à n-1) octets arbitraires pour être le nombre n codé comme un entier non signé (comme décrit ci-dessous), et suivi par les n octets de la séquence.

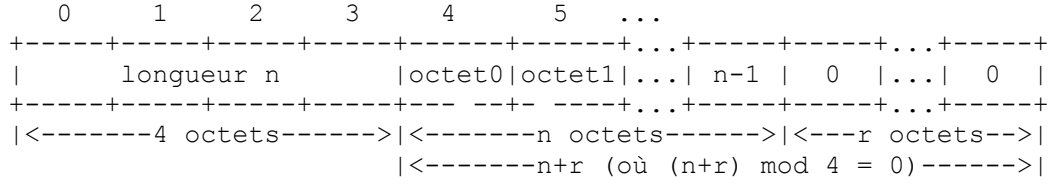
L'octet m de la séquence précède toujours l'octet m+1 de la séquence, et l'octet 0 de la séquence suit toujours la longueur de la séquence (son compte). Si n n'est pas un multiple de quatre, les n octets sont alors suivis par assez (de 0 à 3) d'octets à zéro résiduels, r, pour faire du compte d'octets total un multiple de quatre. Les données opaques de longueur variable sont déclarées de la façon suivante :

opaque identifiant<m>;
ou
opaque identifiant<>;

La constante m note une limite supérieure du nombre d'octets que la séquence peut contenir. Si m n'est pas spécifié, comme dans la seconde déclaration, on suppose que c'est (2**32) - 1, la longueur maximum.

La constante m devrait normalement se trouver dans une spécification de protocole. Par exemple, un protocole de remplissage peut déclarer que la taille maximum de transfert de données est 8192 octets, comme suit :

opaque filedata<8192>;



Données opaques de longueur variable

C'est une erreur de coder une longueur supérieure au maximum décrit dans la spécification.

4.11 Chaîne

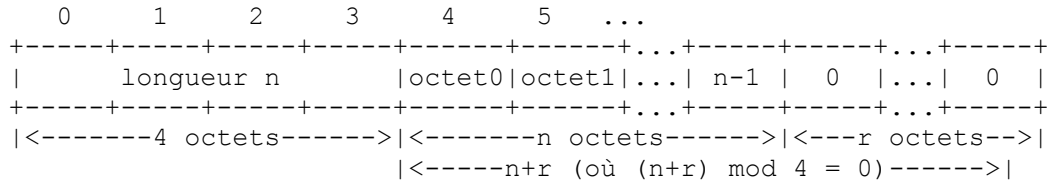
La norme définit une chaîne ASCII de n octets (numérotés de 0 à n-1) comme étant le nombre n codé comme un entier non signé (comme décrit ci-dessus) et suivi par les n octets de la chaîne. L'octet m de la chaîne précède toujours l'octet m+1 de la chaîne, et l'octet 0 de la chaîne suit toujours la longueur de la chaîne. Si n n'est pas un multiple de quatre, les n octets sont alors suivis par assez (de 0 à 3) d'octets zéro résiduels, r, pour rendre le compte total d'octets multiple de quatre. Les

chaînes d'octets comptés sont déclarées comme suit :

objet chaîne<m>;
ou
objet chaîne<>;

La constante m note une limite supérieure de nombre d'octets que peut contenir une chaîne. Si m n'est pas spécifié, comme dans la seconde déclaration, elle est supposée être (2**32) - 1, la longueur maximum. La constante m se trouvera normalement dans une spécification de protocole. Par exemple, un protocole de remplissage peut déclarer qu'un nom de fichier ne peut pas être plus long que 255 octets, comme suit :

chaîne nom-de-fichier<255>;



Chaîne

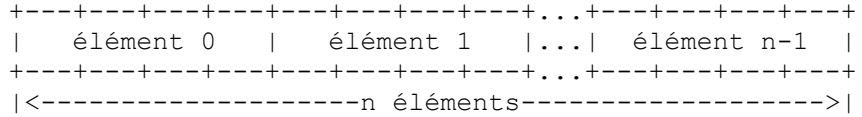
C'est une erreur de coder une longueur supérieure au maximum décrit dans la spécification.

4.12 Dispositif de longueur fixe

Les déclarations pour les dispositifs de longueur fixe d'éléments homogènes sont de la forme suivante :

nom-de-type identifiant[n];

Les dispositifs de longueur fixe d'éléments numérotés de 0 à n-1 sont codés en codant individuellement les éléments du dispositif dans leur ordre naturel, de 0 à n-1. La taille de chaque élément est un multiple de quatre octets. Bien que tous les éléments soient du même type, les éléments peuvent avoir des tailles différentes. Par exemple, dans un dispositif de longueur fixe de chaînes, tous les éléments sont du type "chaîne", et chaque élément va varier dans sa longueur.



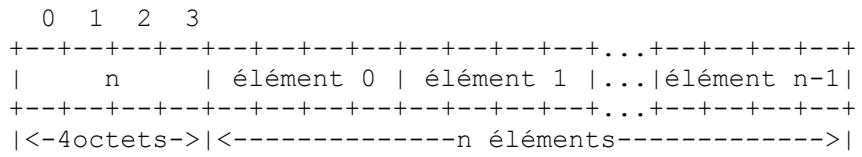
Dispositif de longueur fixe

4.13 Dispositif de longueur variable

Les dispositifs comptés permettent de coder des dispositifs de longueur variable d'éléments homogènes. Le dispositif est codé comme le compte d'éléments n (un entier non signé) suivi du codage de chaque élément du dispositif, de l'élément 0 jusqu'à l'élément n-1. La déclaration pour les dispositifs de longueur variable a la forme suivante :

nom-de-type identifiant<m>;
ou
nom-de-type identifiant<>;

La constante m spécifie le compte maximum acceptable d'éléments d'un dispositif ; si m n'est pas spécifié, comme dans la seconde déclaration, il est supposé être (2**32) - 1.



Dispositif compté

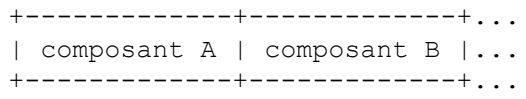
C'est une erreur de coder une valeur de n supérieure au maximum décrit dans la spécification.

4.14 Structure

Les structures sont déclarées comme suit :

struct {
composant-declaration-A;
composant-declaration-B;
...
} identifiant;

Les composants de la structure sont codés dans l'ordre de leur déclaration dans la structure. La taille de chaque composant est un multiple de quatre octets, bien que les composants puissent avoir des tailles différentes.



Structure

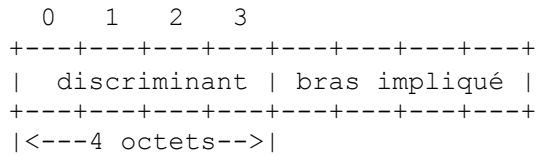
4.15 Union discriminée

Une union discriminée est un type composé d'un discriminant suivi par un type choisi dans un ensemble de types préarrangés selon la valeur du discriminant. Le type du discriminant est soit "entier", "entier non signé", soit un type énuméré tel que "bool". Les types de composant sont appelés "bras" de l'union et sont précédés par la valeur du discriminant qui implique leur codage. Les unions discriminées sont déclarées comme suit :

union switch (discriminant-declaration) {
    cas discriminant-valeur-A:
    bras-declaration-A;
    cas discriminant-valeur-B:
    bras-declaration-B;
    ...
    defaut: defaut-declaration;
} identifiant;

Chaque mot clé "cas" est suivi par une valeur légale du discriminant. Le bras par défaut est facultatif. Si il n'est pas spécifié, un codage valide de l'union ne peut pas prendre de valeurs non spécifiées du discriminant. La taille du bras impliqué est toujours un multiple de quatre octets.

L'union discriminée est codée comme son discriminant suivi par le codage du bras impliqué.



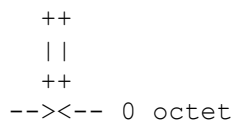
Union discriminée

4.16 Vide

Un XDR vide est une quantité de 0 octet. Les vides sont utiles pour décrire des opérations qui ne prennent pas de données en entrée ou pas de données en sortie. Elles sont aussi utiles dans les unions, où certains bras peuvent contenir des données et d'autres non. La déclaration est simplement la suivante :

```
vide ;
```

Les vides sont illustrés comme suit :



Vide

4.17 Constante

La déclaration des données pour une constante a la forme suivante :

```
const nom-identifiant = n;
```

"const" est utilisé pour définir un nom symbolique pour une constante ; il ne déclare aucune données. La constante symbolique peut être utilisée partout où une constante régulière peut être utilisée. Par exemple, ce qui suit définit une constante symbolique DOUZAINE, égale à 12.

```
const DOUZAINE = 12;
```

4.18 Typedef

"typedef" ne sert à déclarer aucune données, mais sert à définir de nouveaux identifiants pour déclarer des données. La syntaxe est :

```
typedef declaration;
```

Le nom du nouveau type est en fait le nom de la variable dans la partie déclaration de la typedef. Par exemple, ce qui suit définit un nouveau type appelé "boîte-à-œufs" en utilisant un type existant appelé "œuf" :

```
typedef œuf boîte-à-œufs[DOUZAINE];
```

Les variables déclarées en utilisant le nom du nouveau type ont le même type que celui que le nom du nouveau type aurait eu dans la typedef, si il était considéré comme une variable. Par exemple, les deux déclarations suivantes sont équivalentes en déclarant la variable "œufs-frais":

```
boîte-à-œufs  œufs-frais; œuf    œufs-frais[DOUZAINE];
```

Lorsque une typedef implique une définition de struct, enum, ou union, il y a une autre syntaxe (préférée) qui peut être utilisée pour définir le même type. En général, une typedef de la forme suivante :

```
typedef <<définition de struct, union, ou enum>> identifiant;
```

peut être convertie dans l'autre forme en retirant la partie "typedef" et en plaçant l'identifiant après le mot clé "struct", "union", ou "enum", au lieu de le mettre à la fin. Par exemple, voici les deux façons de définir le type "bool":

```
typedef enum {        /* utilisant typedef */
  FAUX = 0,
  VRAI = 1
} bool;

enum bool {           /* solution préférée */
```

```
    FAUX = 0,
    VRAI = 1
};
```

Cette syntaxe est préférée parce que l'une n'a pas à attendre la fin d'une déclaration pour exprimer le nom du nouveau type.

4.19 Données facultatives

Les données facultatives sont une forme d'union qui survient si fréquemment qu'on lui donne une syntaxe particulière propre pour la déclarer. Elle est déclarée comme suit :

```
nom-de-type *identifiant;
```

Ceci est équivalent à l'union suivante :

```
union switch (bool opted) {
cas VRAI:
  nom-de-type élément;
cas FAUX:
  vide;
} identifiant;
```

C'est aussi équivalent à la déclaration de dispositif de longueur variable suivant, car le booléen "opted" peut être interprété comme la longueur du dispositif :

```
nom-de-type identifiant<1>;
```

Les données facultatives ne sont pas aussi intéressantes par elles-mêmes, mais elles sont très utiles pour décrire des structures de données récurrentes telles que des listes liées et des arborescences. Par exemple, ce qui suit définit un type "chaîneliste" qui code des listes de zéro, une ou plusieurs chaînes de longueur arbitraire :

```
struct chaîneentrée {
  chaîne item<>;
  chaîneentrée *suivante;
};

typedef chaîneentrée *chaîneliste;
```

Il aurait été équivalent de déclarer l'union suivante :

```
union chaîneliste switch (bool opted) {
cas VRAI:
  struct {
    chaîne item<>;
    chaîneliste suivante;
  } élément;
cas FAUX:
  vide;
};
```

ou un dispositif de longueur variable :

```
struct chaîneentrée {
  chaîne item<>;
  chaîneentrée suivante<1>;
};

typedef chaîneentrée chaîneliste<1>;
```

Ces deux déclarations rendent obscure l'intention du type chaîneliste, de sorte que la déclaration de données facultatives est préférée à ces deux là. Le type données facultatives est aussi en étroite corrélation avec la façon dont les structures de

données récurrentes sont représentées dans les langages de haut niveau tels que le Pascal ou le C grâce à l'utilisation de pointeurs. En fait, la syntaxe est la même que celle du langage C pour les pointeurs.

4.20 Zones pour des améliorations futures

La norme XDR manque d'une représentation pour les champs binaires et les matrices binaires, car la norme se fonde sur les octets. Manquent aussi les décimales empaquetées (ou codées en binaire).

L'intention de la norme XDR n'était pas de décrire toutes les sortes de données qui ont jamais été envoyées ou qu'on voudra envoyer un jour entre machines. Elle décrit plutôt les types de données les plus communément utilisés dans les langages de haut niveau tels que le Pascal ou le langage C de sorte que les auteurs d'applications dans ces langages soient capables de communiquer facilement sur un support.

On pourrait imaginer des extensions à XDR qui lui permettent de décrire presque tout protocole existant, comme TCP. Le minimum nécessaire pour cela est la prise en charge de différentes tailles de bloc et d'ordre des octets. Le XDR exposé ici pourrait alors être considéré comme le membre gros boutien à quatre octets d'une famille XDR plus large.

5. Discussion

(1) Pourquoi utiliser un langage de description des données ? Qu'est ce qui ne va pas avec les diagrammes ?
L'utilisation d'un langage de description de données tel que XDR présente de nombreux avantages sur l'utilisation de diagrammes. Les langages sont plus formels que les diagrammes et conduisent à des descriptions moins ambiguës des données. Les langages sont aussi plus faciles à comprendre et permettent de penser aux autre questions que les détails de niveau inférieurs du codage de bit. Il y a aussi une étroite analogie entre les types de XDR et un langage de haut niveau tel que le C ou le Pascal. Cela rend la mise en œuvre des modules de codage et de décodage XDR une tâche aisée. Finalement, la spécification du langage lui-même est une chaîne ASCII qui peut être passée d'une machine à l'autre pour effectuer une interprétation des données en direct.

(2) Pourquoi y a t-il seulement un ordre des données pour une unité XDR ?
La prise en charge de deux ordres de rangement des octets exige un protocole de niveau supérieur pour déterminer dans quel ordre des octets sont codées les données. Comme XDR n'est pas un protocole, cela ne peut pas être fait. L'avantage de cela est cependant que les données en format XDR peuvent être écrites, par exemple, sur un support magnétique, et que toute machine sera capable de les interpréter, car aucun protocole de niveau supérieur n'est nécessaire pour déterminer l'ordre des octets.

(3) Pourquoi l'ordre des octets de XDR est il gros boutien plutôt que petit boutien ? Est-ce que ce n'est pas injuste pour les machines petites boutiennes telles que le VAX(r), qui doit convertir d'une forme à l'autre ?
Oui, c'est injuste, mais avoir un seul ordre des octets signifie que vous êtes obligés d'être injuste envers quelqu'un. De nombreuses architectures, telles que celle du Motorola 68000* et de l'IBM 370*, prennent en charge l'ordre gros boutien.

(4) Pourquoi l'unité XDR est elle de quatre octets ?
Il y a un compromis pour choisir la taille de l'unité XDR. Choisir une petite taille, comme deux, rend les données codées petites, mais cause des problèmes d'alignement pour les machines qui ne sont pas alignées sur ces frontières. Une grande taille, comme huit octets, signifie que les données seront alignées virtuellement sur toutes les machines, mais cause une trop forte croissance des données codées. On a choisi quatre comme compromis. Quatre est assez gros pour prendre en charge efficacement la plupart des architectures, sauf quelques rares machines comme le Cray* à alignement sur huit octets. Quatre est aussi assez petit pour garder une taille raisonnable aux données codées.

(5) Pourquoi les données de longueur variable doivent elles être bourrées de zéros ?
Il est souhaitable que les mêmes données codent les mêmes choses sur toutes les machines, de sorte que les données codées puissent être comparées ou avoir des sommes de contrôle significatives. Forcer les octets de bourrage à être à zéro permet de l'assurer.

(6) Pourquoi n'y a t-il pas de frappe de données explicite ?
La frappe de données a un coût relativement élevé par rapport au faibles avantages qu'elle peut avoir. Un coût est celui de l'expansion des données due à l'insertion de champs de type. Un autre est le coût supplémentaire de l'interprétation

de ces champs de type et de l'action qui en découle. Et la plupart des protocoles connaissent déjà le type qu'il attendent, de sorte que la frappe des données fournit seulement des informations redondantes. Cependant, on peut encore bénéficier de la frappe de données en utilisant XDR. Une façon de le faire est de coder deux choses : d'abord, une chaîne qui est la description XDR des données codées, et ensuite les données codées elles-mêmes. Une autre façon est d'allouer une valeur à tous les types dans XDR, puis de définir un type universel qui prend cette valeur comme discriminant et pour chaque valeur, décrit le type de données correspondant.

6. Spécification du langage XDR

6.1 Conventions de notation

La présente spécification utilise une notation de forme Backus-Naur étendue pour la description du langage XDR. Voici une brève description de la notation :

(1) Les caractères '|', '(', ')', '[', ']', "", et '*' sont spéciaux.
(2) Les symboles terminaux sont des chaînes de tout caractère entourées de guillemets.
(3) Les symboles non terminaux sont des chaînes de caractères non spéciaux.
(4) Les éléments alternatifs sont séparés par une barre verticale ("|").
(5) Les éléments facultatifs sont entre guillemets .
(6) Les éléments sont groupés en les incluant entre des parenthèses.
(7) Une '*' à la suite d'un élément signifie 0, une ou plusieurs occurrences de cet élément.

Par exemple, considérons le schéma suivant :

"un " "très" (", " "très")* [" froid " "et "] " pluvieux " ("jour" | "nuit")

Un nombre infini de chaînes correspondent à ce schéma. Quelques uns d'entre elles sont :
"un jour très pluvieux"
"un jour très, très pluvieux"
"un jour très froid et pluvieux"
"un jour très, très, très froid et une nuit pluvieuse"

6.2 Notes lexicales

(1) Les commentaires commencent par "/*" et se terminent par "*/".
(2) Les espaces servent à séparer les éléments et sont ignorées autrement.
(3) Un identifiant est une lettre suivie par une séquence facultative de lettres, chiffres, ou souligné ("_"). La casse des identifiants n'est pas ignorée.
(4) Une constante décimale exprime un nombre en base 10 et est une séquence d'un ou plusieurs chiffres décimaux, où le premier chiffre n'est pas un zéro, et il est facultativement précédé d'un signe moins ('-').
(5) Une constante hexadécimale exprime un nombre en base 16, et doit être précédée de '0x', suivi d'un ou plusieurs chiffres hexadécimaux ('A', 'B', 'C', 'D', E', 'F', 'a', 'b', 'c', 'd', 'e', 'f', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9').
(6) Une constante octale exprime un nombre en base 8, commençant toujours par le chiffre 0, et est une séquence d'un ou plusieurs chiffres octaux ('0', '1', '2', '3', '4', '5', '6', '7').

6.3 Informations syntaxiques

déclaration :
```
      type-spécifieur identifiant
    | type-spécifieur identifiant "[" valeur "]"
    | type-spécifieur identifiant "<" [ valeur ] ">"
    | "opaque" identifiant "[" valeur "]"
    | "opaque" identifiant "<" [ valeur ] ">"
    | "chaîne" identifiant "<" [ valeur ] ">"
    | type-spécifieur "*" identifiant
    | "vide"
```

```
valeur :
         constante
       | identifiant
    constante:
       décimal-constante | hexadécimal-constante | octal-constante

type-spécifieur :
         [ "non-signé" ] "entier"
       | [ "non-signé" ] "hyper"
       | "float"
       | "double"
       | "quadruple"
       | "bool"
       | enum-type-spec
       | struct-type-spec
       | union-type-spec
       | identifiant

enum-type-spec :
       "enum" enum-corps

enum-corps :
       "{"
         ( identifiant "=" valeur )
         ( "," identifiant "=" valeur )*
       "}"

struct-type-spec :
       "struct" struct-corps

struct-corps :
       "{"
         ( declaration ";" )
         ( declaration ";" )*
       "}"

union-type-spec :
       "union" union-corps

union-corps :
       "switch" "(" declaration ")" "{"
         cas-spec
         cas-spec *
         [ "defaut" ":" declaration ";" ]
       "}"

cas-spec :
      ( "cas" valeur ":")
      ( "cas" valeur ":") *
      declaration ";"

constant-def :
       "const" identifiant "=" constante ";"

type-def :
         "typedef" declaration ";"
       | "enum" identifiant enum-corps ";"
       | "struct" identifiant struct-corps ";"
       | "union" identifiant union-corps ";"

definition :
         type-def
       | constant-def
```

```
specification :
        definition *
```

6.4 Notes de syntaxe

(1) Les mots clés qui suivent ne peuvent pas être utilisés comme identifiants :
"bool", "cas", "const", "default", "double", "quadruple", "enum", "float", "hyper", "entier", "opaque", "chaîne", "struct", "switch", "typedef", "union", "non-signé", et "vide".

(2) Seules les constantes non signées peuvent être utilisées comme spécifications de taille pour les matrices. Si un identifiant est utilisé, il doit avoir été déclaré précédemment comme constante non signée dans une définition "const".

(3) Les identifiants de constante et de type au sein du domaine d'application d'une spécification sont dans le même espace de noms et doivent être déclarés uniquement au sein de ce domaine d'application.

(4) De même, les noms de variable doivent être uniques au sein du domaine d'application des déclarations de struct et union. Les déclarations de struct et d'union incorporées créent de nouveaux domaines d'application.

(5) Le discriminant d'une union doit être d'un type qui s'évalue comme entier. C'est à dire que "int", "non-signé entier", "bool", un type énuméré, ou tout type typedefed qui s'évalue comme l'un d'eux est légal. De plus, les valeurs de cas doivent être de l'une des valeurs légales du discriminant. Finalement, une valeur de cas peut n'être pas spécifiée plus d'une fois au sein du domaine d'application d'une déclaration d'union.

7. Exemple de description de données XDR

Voici une brève description de données XDR d'une chose appelée un "fichier", qui pourrait être utilisée pour transférer des fichiers d'une machine à une autre.

```
         const MAXUSERNAME = 32;          /* longueur maximale d'un nom d'usager */
         const MAXFILELEN = 65535;        /* longueur maximale d'un fichier     */
         const MAXNAMELEN = 255;          /* longueur maximale d'un nom de fichier */

/*
* Types de fichiers :
*/
       enum filekind {
         TEXT = 0,                        /* données ascii */
         DATA = 1,                        /* données  brutes   */
         EXEC = 2                         /* exécutable */
       };

/*
* Informations de fichier, par type de fichier :
/
union filetype switch (filekind kind) {
cas TEXT:
         vide;                            /* pas d'informations supplémentaires */
cas DATA:
         chaîne createur<MAXNAMELEN>;     /* créateur des données       */
       cas EXEC:
         chaîne interpreteur<MAXNAMELEN>;  * interpréteur de programme   */
       };

/*
* Un fichier complet :
*/
       struct fichier {
         chaîne nomfichier<MAXNAMELEN>;   /* nom du fichier    */
         filetype type;                   /* info sur le fichier */
```

```
    chaîne owner<MAXUSERNAME>;      /* propriétaire du fichier   */
    opaque data<MAXFILELEN>;        /* données du fichier        */
  };
```

Supposons maintenant qu'il y ait un usager dénommé "john" qui veut mémoriser son programme lisp "sillyprog" qui contient juste les données "(quit)". Son fichier serait codé comme suit :

Décalage	Octets HEX	ASCII	Commentaires
0	00 00 00 09	....	-- longueur du nomfichier = 9
4	73 69 6c 6c	sill	-- caractères du nomfichier
8	79 70 72 6f	ypro	-- ... et plus de caractères ...
12	67 00 00 00	g...	-- ... et 3 octets de remplissage à zéro
16	00 00 00 02	....	-- filekind est EXEC = 2
20	00 00 00 04	....	-- longueur de interpreteur = 4
24	6c 69 73 70	lisp	-- caractères de interpreteur
28	00 00 00 04	....	-- longueur de owner = 4
32	6a 6f 68 6e	john	-- caractères de owner
36	00 00 00 06	....	-- longueur des données du fichier = 6
40	28 71 75 69	(qui	-- octets de données du fichier ...
44	74 29 00 00	t)..	-- ... et 2 octets de remplissage à zéro

8. Considérations pour la sécurité

XDR est un langage de description de données, et non un protocole, et par conséquent il ne soulève par lui-même aucune considération de sécurité particulière.
Les protocoles qui portent des données formatéées en XDR, telles que NFSv4, sont chargées de fournir tous les services de sécurité nécessaires pour sécuriser les données qu'ils transportent.

Il faut veiller à coder et décoder correctement les données pour éviter les attaques. Parmi les risques connus et évitables figurent :

* Les attaques de débordement de mémoire tampon. Lorsque c'est faisable, les protocoles devraient être définis avec des limites explicites (via la notation "<" [valeur] ">" au lieu de "<" ">") sur les éléments qui ont des types de données de longueur variable. Sans considération de la faisabilité d'une limite explicite sur la longueur variable d'un élément d'un protocole donné, les décodeurs ont besoin de s'assurer que la taille des données entrantes ne dépasse pas la longueur de toute mémoire tampon de réception provisionnée.

* Les octets nuls incorporés dans une valeur codée de chaîne de type. Si le format natif de chaîne du décodeur utilise des chaînes à terminaison nulle, la taille apparente de l'objet décodé sera alors inférieure à la quantité de mémoire allouée pour la chaîne. Certaines interfaces de désallocation de mémoire prennent un argument de taille. L'appelant de l'interface de désallocation déterminerait vraisemblablement la taille de la chaîne en comptant jusqu'à la localisation de l'octet nul et en ajoutant un. Cette divergence peut causer une fuite de mémoire (parce que moins de mémoire que ce qui est alloué sera en fait retourné au pôle de réallocation), ce qui conduit à des défaillances système et à une attaque de déni de service.

* Le décodage dans des chaînes, de caractères qui sont des caractères ASCII légaux mais sont néanmoins illégaux pour l'application prévue. Par exemple, certains systèmes d'exploitation traitent le caractère '/'comme un séparateur de composant dans les noms de chemins. Pour un protocole qui code une chaîne dans l'argument comme opération de création de fichier, le décodeur doit s'assurer que '/' n'est pas à l'intérieur du nom du composant. Autrement, un fichier avec un '/' illégal dans son nom sera créé, rendant difficile sa suppression, et c'est donc une attaque de déni de service.

* Le déni de service causé par des instructions de décodeur ou codeur récurrent. Un codeur ou décodeur récurrent pourrait traiter des données qui ont un type structuré avec un membre de données facultatives de type qui réfère directement ou indirectement au type structuré (c'est-à-dire, une liste liée). Par exemple,

```
struct m {
  int x;
  struct m *prochain;
};
```

Une instruction de codeur ou décodeur peut être écrite pour s'appeler elle-même de façon récurrente chaque fois qu'elle trouve un autre élément de type "struct m". Un attaquant pourrait construire une longue liste liée d'éléments "struct m" dans la demande ou la réponse, qui causera alors un débordement de la pile au décodeur ou codeur. Les décodeurs et codeurs devaient être écrits sans récurrence ou imposer une limite à la longueur de la liste.

9. Considérations relatives à l'IANA

Il est possible, bien que peu vraisemblable, que de nouveaux types de données soient ajoutés à l'avenir à XDR. Le processus de l'ajout de nouveaux types est via une RFC en cours de normalisation et non par l'enregistrement de nouveaux types auprès de l'IANA. Les RFC en cours de normalisation qui mettront à jour ou remplaceront le présent document devraient être référencées comme telles dans la base de données des RFC de l'éditeur de RFC.

10. Marques commerciales et de propriété

SUN WORKSTATION	Sun Microsystems, Inc.
VAX	Hewlett-Packard Company
IBM-PC	International Business Machines Corporation
Cray	Cray Inc.
NFS	Sun Microsystems, Inc.
Ethernet	Xerox Corporation.
Motorola 68000	Motorola, Inc.
IBM 370	International Business Machines Corporation

11. Norme ANSI/IEEE 754-1985

La définition des NaN, des zéro et infini signés, et des nombres dénormalisés de la norme [IEEE] est reproduite ici pour des raisons pratiques. Les définitions des nombres à virgule flottante à quadruple précision sont analogues à celles des nombres à virgule flottante à précision simple et double et sont données dans [IEEE].

Dans ce qui suit, 'S' indique le bit de signe, 'E' l'exposant, et 'F' la partie fractionnaire. Le symbole 'u' signifie un bit indéfini (0 ou 1).

Pour les nombres à virgule flottante à simple précision :

Type	**S (1 bit)**	**E (8 bits)**	**F (23 bits)**	
NaN de signalisation	u	255 (max)	.0uuuuu---u	(avec au moins un bit 1)
NaN quiet	u	255 (max)	.1uuuuu---u	
infini négatif	1	255 (max)	.000000---0	
infini positif	0	255 (max)	.000000---0	
zéro négatif	1	0	.000000---0	
zéro positif	0	0	.000000---0	

Pour les nombres à virgule flottante à double précision :

Type	**S (1 bit)**	**E (11 bits)**	**F (52 bits)**	
NaN de signalisation	u	2047 (max)	.0uuuuu---u	(avec au moins un bit 1)
NaN quiet	u	2047 (max)	.1uuuuu---u	
infini négatif	1	2047 (max)	.000000---0	
infini positif	0	2047 (max)	.000000---0	
zéro négatif	1	0	.000000---0	
zéro positif	0	0	.000000---0	

Pour les nombres à virgule flottante à quadruple précision :

Type	**S (1 bit)**	**E (15 bits)**	**F (112 bits)**	
NaN de signalisation	u	32767 (max)	.0uuuuu---u	(avec au moins un bit 1)
NaN quiet	u	32767 (max)	1uuuuu---u	
infini négatif	1	32767 (max)	.000000---0	

infini positif	0	32767 (max)	.000000---0
zéro négatif	1	0	.000000---0
zéro positif	0	0	.000000---0

Les nombre sous normaux sont représentés comme suit :

Précision	Exposant	Valeur
Simple	0	(-1)**S * 2**(-126) * 0.F
Double	0	(-1)**S * 2**(-1022) * 0.F
Quadruple	0	(-1)**S * 2**(-16382) * 0.F

12 Référence normative

[IEEE] "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, août 1985.

13 Références pour information

[KERN] Brian W. Kernighan & Dennis M. Ritchie, "The C Programming Language", Bell Laboratories, Murray Hill, New Jersey, 1978.

[COHE] Danny Cohen, "On Holy Wars and a Plea for Peace", IEEE Computer, octobre 1981.

[COUR] "Courier: The Remote Procedure Call Protocol", XEROX Corporation, XSIS 038112, décembre 1981.

[SPAR] "The SPARC Architecture Manual: Version 8", Prentice Hall, ISBN 0-13-825001-4.

[HPRE] "HP Precision Architecture Handbook", juin 1987, 5954-9906.

14. Remerciements

Bob Lyon a été la cheville ouvrière de Sun derrière ONC RPC dans les années 1980. Sun Microsystems, Inc., figure sur la liste des auteurs de la RFC 1014. Raj Srinivasan et le reste de l'ancien groupe de travail ONC RPC a transformé la RFC 1014 en RFC 1832, dont le présent document est dérivé. Mike Eisler et Bill Janssen ont soumis les rapports de mise en œuvre de la présente norme. Kevin Coffman, Benny Halevy, et Jon Peterson ont relu le document et fait des commentaires. Peter Astrand et Bryan Olson ont découvert plusieurs erreurs dans la RFC 1832 qui sont corrigées dans le présent document.

Adresse de l'éditeur

Mike Eisler
5765 Chase Point Circle
Colorado Springs, CO 80919
USA
téléphone : 719-599-9026
mél : email2mre-rfc4506@yahoo.com

Prière d'adresser les commentaires à : nfsv4@ietf.org

Déclaration de copyright

l'organisation qu'il ou elle représente ou qui le/la finance (s'il en est), la INTERNET SOCIETY et la INTERNET ENGINEERING TASK FORCE déclinent toutes garanties, exprimées ou implicites, y compris mais non limitées à toute garantie que l'utilisation des informations ci encloses ne violent aucun droit ou aucune garantie implicite de commercialisation ou d'aptitude à un objet particulier.

Propriété intellectuelle

L'IETF ne prend pas position sur la validité et la portée de tout droit de propriété intellectuelle ou autres droits qui pourraient être revendiqués au titre de la mise en œuvre ou l'utilisation de la technologie décrite dans le présent document ou sur la mesure dans laquelle toute licence sur de tels droits pourrait être ou n'être pas disponible ; pas plus qu'elle ne prétend avoir accompli aucun effort pour identifier de tels droits. Les informations sur les procédures de l'ISOC au sujet des droits dans les documents de l'ISOC figurent dans les BCP 78 et BCP 79.

Des copies des dépôts d'IPR faites au secrétariat de l'IETF et toutes assurances de disponibilité de licences, ou le résultat de tentatives faites pour obtenir une licence ou permission générale d'utilisation de tels droits de propriété par ceux qui mettent en œuvre ou utilisent la présente spécification peuvent être obtenues sur répertoire en ligne des IPR de l'IETF à http://www.ietf.org/ipr.

L'IETF invite toute partie intéressée à porter son attention sur tous copyrights, licences ou applications de licence, ou autres droits de propriété qui pourraient couvrir les technologies qui peuvent être nécessaires pour mettre en œuvre la présente norme. Prière d'adresser les informations à l'IETF à ietf- ipr@ietf.org.

Remerciement

Le financement de la fonction d'édition des RFC est actuellement fourni par l'activité de soutien administratif (IASA) de l'IETF.