Groupe de travail Réseau
Request for Comments : 1014
Traduction Claude Brière de L'Isle

Sun Microsystems, Inc.
juin 1987

XDR : Norme de représentation de données externes

Statut de ce mémoire
La présente RFC décrit un standard que Sun Microsystems, Inc., et d'autres, utilisent, et qu'on souhaite proposer à l'attention de la communauté de l'Internet. La distribution du présent mémoire n'est soumise à aucune restriction.

1. Introduction

XDR est un standard pour la description et le codage des données. Il est utile pour transférer des données entre différentes architectures d'ordinateur, et a été utilisé pour communiquer des données entre des machines aussi diverses que SUN WORKSTATION*, VAX*, IBM-PC*, et Cray*. XDR se tient dans la couche de présentation de l'ISO, et est en gros analogue dans son objet à X.409, la notation de syntaxe abstraite de l'ISO. La différence majeure entre les deux est que XDR utilise la frappe implicite, alors que celle de X.409 est explicite.

XDR utilise un langage pour décrire le format des données. Le langage peut seulement être utilisé pour décrire des données ; ce n'est pas un langage de programmation. Ce langage permet de décrire des formats de données intriqués d'une manière concise. L'autre solution qui est d'utiliser des représentations graphiques (elles-mêmes étant un langage informel) devient rapidement incompréhensible en présence de complexité. Le langage XDR lui-même est similaire au langage C [1], tout comme Courier [4] est similaire à Mesa. Des protocoles tels que le Sun RPC (Remote Procedure Call) et le NFS* (Network File System) utilisent XDR pour décrire le format de leurs données.

Le standard XDR fait l'hypothèse que les octets sont portables, un octet étant défini comme 8 bits de données. Un appareil matériel donné devrait coder l'octet sur les divers supports de telle façon que les autres matériels puissent décoder les octets sans perte de signification. Par exemple, le standard Ethernet* suggère que les octets soient codés en style "petit boutien" [2], c'est à dire avec le bit de moindre poids en premier.

2. Tailles de bloc de base

La représentation de tout élément exige 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 sur 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 assez (de 0 à 3) d'octets résiduels à zéro, r, pour rendre le compte d'octets total multiple de 4.

On inclut la notation familière par boîtes graphiques pour l'illustration et la comparaison. Dans la plupart des illustrations, chaque boîte (délimitée par un signe plus aux quatre coins et des barres verticales et des traits d'union) représente un octet. Les points de suspension (...) entre les boîtes montrent zéro, un ou plusieurs octets supplémentaires lorsque nécessaire.

```
+--------+--------+...+---------+--------+...+--------+
|octet 0 |octet 1 |...|octet n-1|    0   |...|    0   |   Bloc
+--------+--------+...+---------+--------+...+--------+
|<-----------n octets---------->|<------r octets----->|
|<-------------n+r (où (n+r) mod 4 = 0)>------------->|
```

3. Types de données XDR

Chacun des paragraphes qui suit décrit un type de données défini dans la norme XDR, montre comment il est déclaré dans le langage, et comporte un graphique pour illustrer son codage.

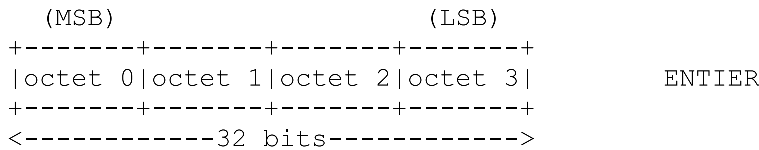
Pour chaque type de données dans le langage, on montre le paradigme général de déclaration. Noter que les crochets angulaires (< et >) notent des séquences de données de longueur variable, et les crochets ([et]) notent des séquences de données de longueur fixe. "n", "m" et "r" notent des entiers. La spécification complète du langage et des définitions plus formelles de termes comme "identifiant" et "déclaration", se trouve à la section 5 "Spécification du langage XDR".

Pour certains types de données, des exemples plus spécifiques sont inclus. Un exemple plus développé de description de données figure à la section 6 "Exemple de description de données XDR".

3.1 Entier

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

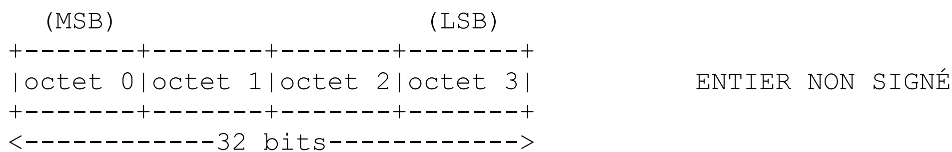
identifiant d'entier ;



3.2 Entier non signé

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

identifiant d'entier non signé ;



3.3 Énumération

Les énumérations ont la même représentation que les entiers signés. Les énumérations sont pratiques 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 autre entier que ceux qui ont reçu une allocation dans la déclaration enum.

3.4 Booléen

les booléens sont assez importants et surviennent assez fréquemment pour assurer 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 ;

3.5 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 poids fort et de moindre poids sont respectivement 0 et 7. Leurs déclarations sont :

identifiant hyper ; identifiant hyper non signé ;

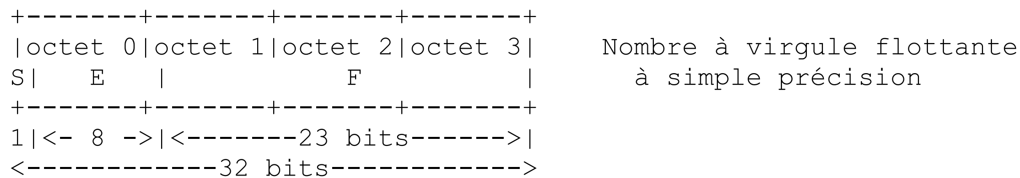
```
 (MSB)                                                       (LSB)
+-------+-------+-------+-------+-------+-------+-------+-------+
|octet 0|octet 1|octet 2|octet 3|octet 4|octet 5|octet 6|octet 7| Hyper entier
+-------+-------+-------+-------+-------+-------+-------+-------+ Hyper entier non
signé
<----------------------------64 bits---------------------------->
```

3.6 Virgule flottante

La norme définit le type de données à virgule flottante "float" (32 bits soit 4 octets). Le codage utilisé est le standard IEEE pour les nombres normalisés à virgule flottante à simple précision [3]. Les trois champs suivants décrivent le nombre à virgule flottante à simple précision :
S : Signe du nombre. Les valeurs 0 et 1 représentent respectivement le positif et le négatif. Un bit.
E : Exposant du nombre, en base 2. 8 bits sont consacrés à ce champ. Le biais de l'exposant est 127.
F : Partie fractionnaire de la mantisse du nombre, en base 2. 23 bits sont consacrés 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 de float ;



```
+-------+-------+-------+-------+
|octet 0|octet 1|octet 2|octet 3|    Nombre à virgule flottante
S|   E   |           F          |      à simple précision
+-------+-------+-------+-------+
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 à simple précision sont 0 et 31. Les décalages des bits de début (et de poids fort) de S, E, et F sont respectivement de 0, 1, et 9. Noter que ces nombres se réfèrent aux positions mathématiques des bits, et NON à leur position physique réelle (qui peut varier selon le support).

On devrait consulter les spécifications de l'IEEE pour ce qui concerne le codage des nombres zéro signé, infini signé (débordement) et dénormalisés (dépassement de capacité négatif) [3]. Selon les spécifications de l'IEEE, le "NaN" (pas un nombre , *not a number*) dépend du système et ne devrait pas être utilisé en externe.

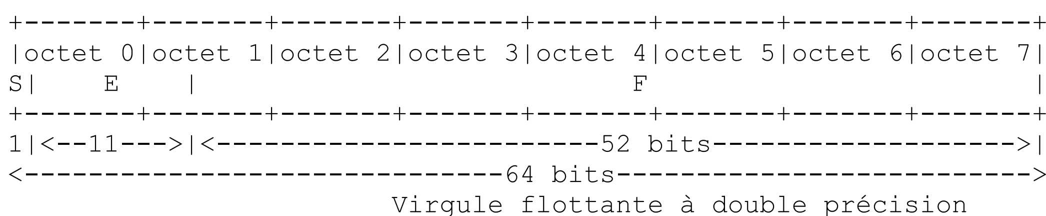
3.7 Virgule flottante à double précision

La norme définit le codage pour le type de données à virgule flottante à double précision "double" (64 bits ou 8 octets). Le codage utilisé est la norme IEEE pour les nombres normalisés à virgule flottante à double précision [3]. La norme code les trois champs suivants, qui décrivent les nombres à virgule flottante à double précision :
S : Signe du nombre. Les valeurs 0 et 1 représentent respectivement le positif et le négatif. Un bit.
E : Exposant du nombre, en base 2. 11 bits sont dédiés à ce champ. L'exposant est biaisé de 1023.
F : Partie fractionnaire de la mantisse du nombre, base 2. 52 bits sont alloués à ce champ.

Le nombre à virgule flottante est donc décrit par :

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

Il est déclaré comme suit :
identifiant double ;



```
+-------+-------+-------+-------+-------+-------+-------+-------+
|octet 0|octet 1|octet 2|octet 3|octet 4|octet 5|octet 6|octet 7|
S|    E    |                       F                            |
+-------+-------+-------+-------+-------+-------+-------+-------+
1|<--11--->|<-----------------------52 bits------------------->|
<------------------------------64 bits------------------------->
                       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 de bit de début (et bit de poids fort) de S, E , et F sont respectivement de 0, 1, et 12. Noter que ces nombres se réfèrent aux positions mathématiques des bits, et NON à leur situation physique réelle (qui varie selon le support).

On devrait consulter les spécifications de l'IEEE sur le codage des nombres zéro signé, infini signé (débordement), et dénormalisés (dépassement de capacité négatif) [3]. Selon les spécifications de l'IEEE, le "NaN" (pas un nombre , *not a number*) dépend du système et ne devrait pas être utilisé en externe.

3.8 Données opaques de longueur fixe

Parfois, des données non interprétées de longueur fixe doivent être passés d'une machine à l'autre. Ces données sont dites "opaques" et sont déclarées comme suit :

identifiant[n] opaque ;

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

```
    0        1     ...
+--------+--------+...+--------+--------+...+--------+
|octet 0 |octet 1 |...|octet n-1|    0   |...|    0   |
+--------+--------+...+--------+--------+...+--------+
|<-----------n octets---------->|<------r octets----->|
|<--------------n+r (où (n+r) mod 4 = 0)------------->|
                            Opaque de longueur fixe
```

3.9 Données opaques de longueur variable

La norme traite aussi des données opaques de longueur variable (comptée) définies comme une séquence de n octets arbitraires (numérotés de 0 à n-1) comme étant 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 (le compte) de la séquence. Si n n'est pas un multiple de quatre, les n octets sont alors suivis par assez (de 0 à 3) octets résiduels à zéro, r, pour rendre le compte total d'octets multiple de quatre. Les données opaques de longueur variable sont déclarées de la façon suivante :

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

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, il est supposé être (2**32) - 1, la longueur maximum. La constante m va normalement se trouver dans une spécification de protocole. Par exemple, un protocole de gestion de fichier peut déclarer que la taille maximum de transfert de données est de 8192 octets, comme suit :

opaque filedata<8192>;

```
   0     1     2     3     4     5   ...
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|      longueur n       |octt0|octt1|...| n-1 |  0  |...|  0  |
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|<-------4 octets------>|<------n octets----->|<---r octets-->|
                        |<-----n+r (où (n+r) mod 4 = 0)------>|
                                     Opaque de longueur variable
```

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

3.10 Chaîne

La norme définit une chaîne de n octets ASCII (numérotés de 0 à n-1) comme étant le nombre n codé comme un entier non signé (comme décrit plus haut) 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 résiduels à zéro, r, pour rendre le compte total des octets multiple de quatre. Les chaînes d'octets comptées sont déclarées comme suit :

objet chaîne<m>;
ou
objet chaîne<>;

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

string filename<255>;

```
   0     1     2     3     4     5   ...
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|      longueur n       |octt0|octt1|...| n-1 |  0  |...|  0  |
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|<-------4 octets------>|<------n octets----->|<---r octets-->|
                        |<--- -n+r (où (n+r) mod 4 = 0)-  --->|
                                                         Chaîne
```

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

3.11 Dispositif de longueur fixée

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

identifiant de nom de type[n] ;

Les dispositifs de longueur fixée 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, ils peuvent avoir des tailles différentes. Par exemple, dans un dispositif de longueur fixée de chaînes, tous les éléments sont du type "chaîne", bien que chaque élément diffère par sa longueur.

```
+---+---+---+---+---+---+---+---+...+---+---+---+---+
|   élément 0   |   élément 1   |...|  élément n-1  |
+---+---+---+---+---+---+---+---+...+---+---+---+---+
|<--------------------n éléments------------------->|
```

Dispositif de longueur fixée

3.12 Dispositif de longueur variable

Les dispositifs comptés donnent la capacité de coder des dispositifs de longueur variable d'éléments homogènes. Le dispositif est codé comme un compte d'éléments n (un entier non signé) suivi par le codage de chacun des éléments du dispositif, en commençant par l'élément 0 et en progressant jusqu'à l'élément n-1. La déclaration des dispositifs de longueur variable a la forme suivante :

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

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

```
  0  1  2  3
+--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
|     n     | élément 0 | élément 1 |...|élément n-1|
+--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
|<-4 octets>|<--------------n éléments------------->|
```

Dispositif compté

C'est une erreur de coder une valeur de n plus grande que le maximum décrit dans la spécification.

3.13 Structure

Les structures sont déclarées comme suit :

```
struct {
  déclaration-de composant-A;
  déclaration-de composant-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 être de différentes tailles.

```
+-------------+-------------+...
| composant A | composant B |...             Structure
+-------------+-------------+...
```

3.14 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 "int", soit "unsigned int", 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-value-A:
   arm-declaration-A;
   cas discriminant-value-B:
   arm-declaration-B;
   ...
   default: default-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 alors pas prendre des valeurs de discriminant non spécifiées. 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.

```
  0   1   2   3
+---+---+---+---+---+---+---+---+
|  discriminant | bras impliqué |          UNION DISCRILINÉE
+---+---+---+---+---+---+---+---+
|<---4 octets-->|
```

3.15 Vide

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

Les vides sont illustrés comme suit :

```
     ++
     ||                                                  VIDE
     ++
   --><-- 0 octet
```

3.16 Constante

La déclaration des données pour une constante suit cette forme :

```
const name-identifier = n;
```

"const" est utilisé pour définir un nom symbolique pour une constante ; il ne déclare aucune donnée. 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;
```

3.17 Typedef

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

```
typedef declaration;
```

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

```
typedef œuf boîte_à_œuf [DOUZAINE];
```

Les variables déclarées en utilisant le nom du nouveau type ont le même type qu'aurait le nom de nouveau type dans le typedef, si il avait été considéré comme une variable. Par exemple, les deux déclarations suivantes sont équivalentes pour déclarer la variable "œufs_frais" :

```
boîte_à_œuf  œufs_frais;
œuf     œufs_frais[DOUZAINE];
```

Lorsque un 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, un typedef de la forme suivante :

```
typedef <<struct, union, ou enum definition>> identifiant;
```

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

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

enum bool {                 /* solution préférée */
  FAUX = 0,
  VRAI = 1
};
```

La raison pour laquelle cette syntaxe est préférée est qu'on n'a pas à attendre jusqu'à la fin de la déclaration pour savoir le nom du nouveau type.

3.18 Optional-data

Optional-data *(données facultatives)* est une sorte d'union qui se produit si fréquemment qu'on lui donne une syntaxe qui lui est particulière pour la déclarer. On la déclare comme suit :

```
type-name *identifiant;
```

C’est équivalent à l’union suivante :

```
union switch (option booléen) {
cas VRAI:
  élément type-name;
cas FAUX:
  vide;
} identifiant;
```

C’est aussi équivalent à la déclaration de dispositif de longueur variable suivante, car l’"option" booléenne peut être interprétée comme étant la longueur du dispositif :

```
type-name identifiant<1>;
```

Optional-data n’est pas très intéressant par lui-même, mais est très utile pour décrire des structures de données récurrentes telles que des listes liées et des arborescences. Par exemple, ce qui suit définit un type "stringlist" qui code des listes de chaînes de longueur arbitraire :

```
struct *stringlist {
  string item<>;
  stringlist next;
};
```

Cela aurait pu être déclaré de façon équivalente comme l’union suivante :

```
union stringlist switch (bool opted) {
      cas VRAI:
        struct {
          string item<>;
          stringlist next;
        } element;
      cas FAUX:   vide;
      };
```

ou comme un dispositif de longueur variable :

```
struct stringlist<1> {
  string item<>;
  stringlist next;
};
```

Ces deux déclarations obscurcissent l’intention du type stringlist, de sorte que la déclaration optional-data est préférée à ces deux là. Le type optional-data a aussi une corrélation étroite avec la façon dont sont représentées les données récurrentes dans des langages de haut niveau tels que le Pascal ou le langage C par l’utilisation de pointeurs. En fait, la syntaxe est la même que celle du langage C pour les pointeurs.

3.19 Directions pour des améliorations futures

La norme XDR manque de représentations pour les champs binaires et les bitmaps, car la norme se fonde sur les octets. Il manque aussi une représentation des décimaux empaquetés (ou codés en binaire).

L’intention de la norme XDR n’était pas de décrire toutes les sortes de données qu’on peut envoyer ou vouloir envoyer de machine à machine. Elle décrit plutôt les types de données les plus couramment utilisés par les langages de haut niveau tels que le Pascal ou le langage C afin que les applications écrites dans ces langages soient capables de communiquer facilement sur un certain support.

On pourrait imaginer des extensions à XDR qui lui feraient décrire presque tous les protocoles existants, comme TCP. Le minimum nécessaire pour cela est la prise en charge des tailles de bloc et des ordres des octets différents. Le XDR discuté ici pourrait alors être considéré comme le membre gros boutien à quatre octets d’une plus grande famille XDR.

4. Discussion

(1) Pourquoi utiliser un langage pour décrire les données ? Qu'est ce qui ne va pas avec les diagrammes ?
Il y a de nombreux avantages à utiliser un langage de description de données tel que XDR plutôt que d'utiliser des diagrammes. Les langages sont plus formels que les diagrammes et conduisent à moins d'ambiguïtés dans les descriptions des données. Les langages sont aussi plus faciles à comprendre et permettent de regarder d'autres aspects que les détails de bas niveau du codage des bits. Aussi, il y a 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 de XDR une tâche plus aisée. Finalement, la spécification du langage lui-même est une chaîne ASCII qui peut être passée d'une machine à l'autre pour effectuer l'interprétation des données au vol.

(2) Pourquoi y a-t-il seulement un ordre des octets pour une unité XDR ?
La prise en charge de deux ordres des octets exige un protocole de niveau supérieur pour déterminer dans quel ordre d'octets sont codées les données. Comme XDR n'est pas un protocole, cela ne peut pas être fait. L'avantage est cependant que les données en format XDR peuvent être écrites sur une bande magnétique, par exemple, 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 données dans XDR est il l'ordre gros boutien plutôt que le petit boutien ? N'est-ce pas injuste pour les machines petites-boutiennes telles que le VAX(r), qui doivent convertir d'une forme à l'autre ?
Oui, c'est injuste, mais avoir un seul ordre des octets signifie qu'on doit être injuste avec quelqu'un. De nombreuses architectures, comme celle du Motorola 68000* et de l'IBM 370*, prennent en charge l'ordre gros-boutien des octets.

(4) Pourquoi l'unité XDR fait elle quatre octets ?
Il y a eu un compromis sur la taille de l'unité XDR. Le choix d'une petite taille comme deux rend petites les données à coder, mais cause des problèmes d'alignement pour les machines qui ne sont pas alignées sur ces limites. Une grande taille comme huit signifierait que les données seraient alignées sur virtuellement toutes les machines, mais causerait une grosse inflation des données codées. On a choisi quatre comme compromis. Quatre est assez gros pour prendre en charge efficacement la plupart des architectures, sauf de rares machines comme le Cray* aligné sur huit octets. Quatre est aussi assez petit pour garder les données codées à une taille raisonnable.

(5) Pourquoi les données de longueur variable doivent elles être bourrées avec des zéros ?
Il est souhaitable que les mêmes données se codent de la même façon sur toutes les machines, afin que les données codées ou leurs sommes de contrôle puissent être comparées de façon significative. Forcer les octets de bourrage à être à zéro l'assure.

(6) Pourquoi n'y a-t-il pas de frappe explicite des données ?
La frappe des données a un coût relativement élevé par rapport au faible avantage qu'elle peut apporter. Un coût est l'expansion des données due aux champs de type insérés. Un autre est le coût supplémentaire de l'interprétation de ces champs de type et de l'action qui s'ensuit. Et la plupart des protocoles savent déjà quel type attendre, de sorte que la frappe des données ne fournit que des informations redondantes. Cependant, on peut encore bénéficier de la frappe des données en utilisant XDR. Une façon de le faire est de coder deux choses : d'abord une chaîne qui est la description de données XDR des données codées, et ensuite les données codées elles-mêmes. Une autre façon est d'allouer une valeur à tous les types en XDR, puis de définir un type universel qui prend cette valeur comme discriminant et pour chaque valeur, de décrire le type de données correspondant.

5. Spécification du langage XDR

5.1 Conventions de notation

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

(1) Les caractères '|', '(', ')', '[', ']', '"', et '*' sont des caractères spéciaux.
(2) Les symboles terminaux sont des chaînes de tous caractères 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 inclus entre des crochets.
(6) Les éléments sont groupés en les incluant entre des parenthèses.
(7) Une '*' suivant un élément signifie 0 ou plusieurs occurrences de cet élément.

Par exemple, considérons le schéma suivant :

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

Un nombre infini de chaînes correspondent à ce schéma. Quelques unes 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 pluvieux"

5.2 Notes lexicales

(1) Les commentaires commencent par '/*' et se terminent par '*/'.
(2) Les espaces servent à séparer les éléments et sont autrement ignorées.
(3) Un identifiant est une lettre suivie par une séquence facultative de lettres, chiffres ou barre de soulignement ('_'). La casse des identifiants n'est pas ignorée.
(4) Une constante est une séquence de un ou plusieurs chiffres décimaux, facultativement précédés du signe moins ('-').

5.3 Informations sur la syntaxe

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

valeur :
    constante
  | identifiant

spécificateur-de-type :
    [ "non-signé" ] "entier"
  | [ "non-signé" ] "hyperentier"
  | "flottant"
  | "double"
  | "booléen"
  | enum-type-spec
  | struct-type-spec
  | union-type-spec   | identifiant

enum-type-spec:
  "enum" corps-d'énum

corps-d'énum
  "{"
    ( identifiant "=" valeur )
    ( "," identifiant "=" valeur )*
  "}"

struct-type-spec:
  "struct" corps-de-structure

corps-de-structure :
  "{"
    ( déclaration ";" )
    ( déclaration ";" )*
  "}"

union-type-spec:
```

```
  "union" corps-d'union

corps-d'union :
  "switch" "(" déclaration ")" "{"
    ( "cas" valeur ":" déclaration ";" )
    ( "cas" valeur ":" déclaration ";" )*
    [ "défaut" ":" déclaration ";" ]
  "}"

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

type-def:
    "typedef" déclaration ";"
  | "enum" identifiant corps-d'énum ";"
  | "struct" identifiant corps-de-structure ";"
  | "union" identifiant corps-d'union ";"

définition:
    type-def
  | constant-def

spécification:
    définition *
```

5.4 Notes de syntaxe

(1) Les mots clés suivants ne peuvent pas être utilisés comme identifiants : "bool", "cas", "const", "défaut", "double", "enum", "flottant", "hyperentier", "opaque", "chaîne", "struct", "switch", "typedef", "union", "non-signé" et "vide".

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

(3) Les identifiants de constante et de type dans le domaine d'une spécification sont dans le même espace de nom et doivent être déclarés de façon univoque au sein de ce domaine.

(4) De façon similaire, les noms de variables doivent être uniques au sein des portées de structure et des déclaration d'union. Les structures incorporées et les déclarations d'union créent de nouvelles portées.

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

6. Exemple de description de données XDR

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

```
      const MAXUSERNAME = 32;          /* longueur maximale d'un nom d'utilisateur */
      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 sorte de fichier : */

   union filetype switch (filekind kind) {
   cas TEXT:
     vide;                                   /* pas d'informations supplémentaires */
   cas DATA:
     string creator<MAXNAMELEN>;             /* créateur de données  */
   cas EXEC:
     string interpretor<MAXNAMELEN>;         /* interpréteur de programme */
   };

/* Un fichier complet :  */
   struct file {
     string filename<MAXNAMELEN>;            /* nom de fichier   */
     filetype type;                          /* informations sur le fichier */
     string owner<MAXUSERNAME>;              /* propriétaire du fichier  */
     opaque data<MAXFILELEN>;                /* données de fichier  */
   };
```

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

Décalage	Octets hex	ASCII	Commentaires
0	00 00 00 09	....	-- longueur du nom de fichier = 9
4	73 69 6c 6c	sill	-- caractères du nom de fichier
8	79 70 72 6f	ypro	-- ... autres caractères ...
12	67 00 00 00	g...	-- ... et 3 octets à zéro de bourrage
16	00 00 00 02	....	-- le type du fichier est EXEC = 2
20	00 00 00 04	....	-- longueur de l'interpréteur = 4
24	6c 69 73 70	lisp	-- caractères de l'interpréteur
28	00 00 00 04	....	-- longueur du propriétaire = 4
32	6a 6f 68 6e	john	-- caractères du propriétaire
36	00 00 00 06	....	-- longueur du fichier de données = 6
40	28 71 75 69	(qui	-- octets des données du fichier ...
44	74 29 00 00	t)..	-- ... et deux octets de bourrage à zéro

7. Références

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

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

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

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

8. Marques commerciales et propriétaires

SUN WORKSTATION	Sun Microsystems, Inc.
VAX	Digital Equipment Corporation
IBM-PC	International Business Machines Corporation
Cray	Cray Research
NFS	Sun Microsystems, Inc.
Ethernet	Xerox Corporation.
Motorola 68000	Motorola, Inc.
IBM 370	International Business Machines Corporation