Groupe de travail Réseau
Request for Comments : 3492
Catégorie : En cours de normalisation
Traduction Claude Brière de L'Isle

A. Costello
Univ. of California, Berkeley

mars 2003

Punycode : codage Bootstring de Unicode pour les noms de domaine internationalisés dans les applications (IDNA)

Statut de ce mémoire
Le présent document spécifie un protocole Internet en cours de normalisation pour la communauté de l'Internet, et appelle à des discussions et des suggestions pour son amélioration. Prière de se reporter à l'édition actuelle du STD 1 "Normes des protocoles officiels de l'Internet" pour connaître l'état de normalisation et le statut de ce protocole. La distribution du présent mémoire n'est soumise à aucune restriction.
(La présente traduction incorpore les errata 265 et 3026).

Résumé
Punycode est une syntaxe simple et efficace de codage de transfert conçue pour être utilisée avec les noms de domaine internationalisés dans les applications (IDNA, *Internationalized Domain Names in Applications*). Elle transforme de façon univoque et réversible une chaîne Unicode en une chaîne ASCII. Les caractères ASCII dans la chaîne Unicode sont représentés littéralement, et les caractères non ASCII sont représentés par des caractères ASCII qui sont permis dans les étiquettes de nom d'hôte (lettres, chiffres, et traits d'union). Le présent document définit un algorithme général appelé Bootstring qui permet qu'une chaîne de codets de base représente de façon univoque toute chaîne de codets tirée d'un ensemble plus large. Punycode est une instance de Bootstring qui utilise des valeurs de paramètre particulières spécifiées par le présent document, appropriées pour les IDNA.

Table des Matières

1. Introduction....2
1.1 Caractéristiques....2
1.2 Interaction des parties de protocole....2
2. Terminologie....2
3. Description de Bootstring....3
3.1 Ségrégation des codets de base....3
3.2 Insertion de codage non trié....3
3.3 Entiers généralisés de longueur variable....3
3.4 Adaptation du biais....4
4. Paramètres Bootstring....5
5. Valeurs des paramètres pour Punycode....5
6. Algorithmes Bootstring....6
6.1 Fonction d'adaptation de biais....6
6.2 Procédure de décodage....6
6.3 Procédure de codage....7
6.4 Traitement des débordements....8
7. Exemples de Punycode....8
7.1 Exemples de chaînes....8
7.2 Traces de décodage....10
7.3 Traces de codage....12
8. Considérations sur la sécurité....13
9. Références....13
A. Annotation en casse mixte....13
B. Déclinatoire de responsabilité et licence....13
C. Mise en œuvre d'échantillon de Punycode....14

1. Introduction

La [RFC3490] décrit une architecture pour la prise en charge des noms de domaines internationalisés. Les étiquettes qui contiennent des caractères non ASCII peuvent être représentées par des étiquettes ACE, qui commencent par un préfixe ACE particulier et ne contiennent que des caractères ASCII. Le reste de l'étiquette après le préfixe est un codage Punycode d'une chaîne Unicode qui satisfait à certaines contraintes. Pour les détails du préfixe et des contraintes, voir les [RFC3490] et [RFC3491].

Punycode est une instance d'un algorithme plus général appelé Bootstring, qui permet des chaînes composées d'un petit ensemble de codets de "base" pour représenter de façon univoque toute chaîne de codets tirée d'un ensemble plus large. Punycode est Bootstring avec des valeurs de paramètres particulières appropriées pour l'IDNA.

1.1 Caractéristiques

Bootstring a été conçu pour avoir les caractéristiques suivantes :

* Complétude : chaque chaîne étendue (séquence de codets arbitraire) peut être représentée par une chaîne de base (séquence de codets de base). Des restrictions sur les chaînes admises, et sur leur longueur, peuvent être imposées par les couches supérieures.
* Unicité : il y a au moins une chaîne de base qui représente une certaine chaîne étendue.
* Réversibilité : toute chaîne étendue transposée en une chaîne de base peut être récupérée à partir de cette chaîne de base.
* Codage efficace : le ratio de la longueur de la chaîne de base sur celle de la chaîne étendue est faible. Ceci est important dans le contexte des noms de domaines parce que la [RFC1034] restreint la longueur d'une étiquette de domaine à 63 caractères.
* Simplicité : les algorithmes de codage et de décodage sont raisonnablement simples à mettre en œuvre. Les objectifs d'efficacité et de simplicité sont contradictoires ; Bootstring vise à un bon équilibre entre les deux.
* Lisibilité : les codets de base qui apparaissent dans les chaînes étendues sont représentés par eux-mêmes dans la chaîne de base (bien que le principal objet soit d'améliorer l'efficacité, non la lisibilité).

Punycode peut aussi prendre en charge une caractéristique supplémentaire qui n'est pas utilisée par les opérations ToASCII et ToUnicode de la [RFC3490]. Lorsque des chaînes étendues ont leur casse changée avant le codage, la chaîne de base peut utiliser une casse mixte pour dire comment convertir la chaîne changée en une chaîne à casse mixte. Voir l'Appendice "Annotation en casse mixte".

1.2 Interaction des parties de protocole

Punycode est utilisé par le protocole IDNA [RFC3490] pour convertir les étiquettes de domaines en ASCII ; il n'est conçu pour aucun autre objet. Il est explicitement non conçu pour traiter des chaînes arbitraires de texte libre.

2. Terminologie

Les mots clés "DOIT", "NE DOIT PAS", "EXIGE", "DEVRA", "NE DEVRA PAS", "DEVRAIT", "NE DEVRAIT PAS", "RECOMMANDE", "PEUT", et "FACULTATIF" en majuscules dans ce document sont à interpréter comme décrit dans le BCP 14, [RFC2119].

Un codet *(code point)* est une valeur entière associée à un caractère dans un jeu de caractères codé.

Comme dans la norme [UNICODE], les codets Unicode sont notés par "U+" suivi par quatre à six chiffres hexadécimaux, tandis qu'une gamme de codets est notée par deux nombres hexadécimaux séparés par "..", sans préfixe.

Les opérateurs div et mod effectuent la division d'entiers ; (x div y) est le quotient de x divisé par y, éliminant le reste, et (x mod y) est le reste, de sorte que (x div y) * y + (x mod y) == x. Bootstring n'utilise ces opérateurs qu'avec des opérandes non négatifs, de sorte que le quotient et le reste sont toujours non négatifs.

La déclaration de coupure sort de la boucle la plus interne (comme dans le langage C).

Un débordement est une tentative de calcul d'une valeur qui excède la valeur maximum d'une variable d'entier.

3. Description de Bootstring

Bootstring représente une séquence arbitraire de codets (la chaîne "étendue") comme une séquence de codets de base (la "chaîne de base"). La présente section décrit cette représentation. La Section 6 "Algorithmes Bootstring" présente les algorithmes en pseudo code. Les paragraphes 7.1 "Traces de décodage" et 7.2 "Traces de codage" tracent les algorithme pour des exemples d'entrées.

Les paragraphes qui suivent décrivent les quatre techniques utilisées dans Bootstring. La "ségrégation de codet de base" est un codage très simple et efficace pour les codets de base qui surviennent dans la chaîne étendue : ils sont simplement copiés d'un seul coup. Le "codage d'insertion non triée" code les codets qui ne sont pas de base comme des deltas, et traite les codets en ordre numérique plutôt que dans l'ordre d'apparition, ce qui résulte normalement en de plus petits deltas. Les deltas sont représentés comme des "entiers généralisés de longueur variable", qui utilisent les codets de base pour représenter des entiers non négatifs. Les paramètres de cette représentation d'entiers sont ajustés de façon dynamique en utilisant une "adaptation de biais" pour améliorer l'efficacité lorsque des deltas consécutifs ont des ordres de grandeur similaires.

3.1 Ségrégation des codets de base

Tous les codets de base qui apparaissent dans la chaîne étendue sont représentés littéralement au début de la chaîne de base, dans leur ordre d'origine, suivis par un délimiteur si (et seulement si) le nombre de codets de base n'est pas zéro. Le délimiteur est un codet de base particulier, qui n'apparaît jamais dans le reste de la chaîne de base. Le décodeur peut donc trouver la fin de la portion littérale (si il en est une) en examinant où est le dernier délimiteur.

3.2 Insertion de codage non trié

Le reste de la chaîne de base (après le dernier délimiteur (si il y en a un) représente une séquence de deltas entiers non négatifs comme entiers de longueur variable généralisée, décrits au paragraphe 3.3. La signification des deltas est mieux comprise en termes de décodeur.

Le décodeur construit la chaîne étendue de façon incrémentaire. Au début, la chaîne étendue est une copie de la portion littérale de la chaîne de base (à l'exclusion du dernier délimiteur). Le décodeur insère des codets qui ne sont pas de base, un pour chaque delta, dans la chaîne étendue, jusqu'à arriver à la chaîne décodée finale.

Au cœur de ce processus est un automate à états avec deux variables d'état : un indice i et un compteur n. L'indice i se réfère à une position dans la chaîne étendue ; elle va de 0 (première position) à la longueur actuelle de la chaîne étendue (qui se réfère à une position potentielle au-delà de la fin actuelle). Si l'état en cours est <n,i>, le prochain état est <n,i+1> si i est inférieur à la longueur de la chaîne étendue, ou <n+1,0> si i est égal à la longueur de la chaîne étendue. En d'autres termes, chaque changement d'état cause l'incrément de i, revenant à zéro si nécessaire, et n compte le nombre de retours à zéro.

Noter que l'état avance toujours de façon monotone (il n'y a aucun moyen pour que le décodeur retourne à un état antérieur). À chaque état, une insertion est soit effectuée soit non effectuée. Au plus une insertion est effectuée dans un certain état. Une insertion insère la valeur de n à la position i dans la chaîne étendue. Les deltas sont un codage au fil de cette séquence d'événements : ils sont les longueurs des séquences d'états de non insertion précédant les états d'insertion. Donc, pour chaque delta, le décodeur effectue des changements d'état, puis une insertion, et ensuite un changement d'état de plus. (Une mise en œuvre n'a pas besoin d'effectuer individuellement chaque changement d'état, mais peut à la place utiliser la division et le calcul du reste pour calculer directement le prochain état d'insertion.) Il y a une erreur si le codet inséré est un codet de base (parce que les codets de base sont supposés être séparés comme décrit au paragraphe 3.1).

La principale tâche du codeur est de déduire la séquence de deltas qui va causer la construction par le décodeur de la chaîne désirée. Il peut le faire en examinant de façon répétée la chaîne étendue à la recherche du prochain codet que le décodeur aurait besoin d'insérer, et en comptant le nombre de changements d'état que le décodeur aurait besoin d'effectuer, en se souvenant du fait que la chaîne étendue du décodeur va inclure seulement les codets qui ont déjà été insérés. Le paragraphe 6.3 "Procédure de codage" donne un algorithme précis.

3.3 Entiers généralisés de longueur variable

Dans une représentation d'entiers conventionnelle, la base est le nombre de symboles distincts pour les chiffres, dont les valeurs sont de 0 à base-1. Soit chiffre_0 qui note le chiffre de moindre poids, chiffre_1 celui de moindre poids suivant, et ainsi de suite. La valeur représentée est la somme sur j de chiffre_j * w(j), où w(j) = base^j est le poids (facteur d'échelle) pour la

position j. Par exemple, dans l'entier 437 en base 8, les chiffres sont 7, 3, et 4, et les poids sont 1, 8, et 64, de sorte que la valeur est 7 + 3*8 + 4*64 = 287. Cette représentation présente deux inconvénients : d'abord, il y a plusieurs codages de chaque valeur (parce qu'il peut y avoir des zéros supplémentaires dans les positions les plus significatives) ce qui est un inconvénient lorsque des codages uniques sont nécessaires. Ensuite, l'entier n'est pas auto délimitant, de sorte que si plusieurs entiers sont enchaînés, les frontières entre eux sont perdues.

La représentation de longueur variable généralisée résout ces deux problèmes. Les valeurs des chiffres sont toujours de 0 à base-1, mais maintenant l'entier est auto délimité au moyen des seuils t(j), dont chacun est dans la gamme de 0 à base-1. Exactement un chiffre, le plus significatif, satisfait digit_j < t(j). Donc, si plusieurs entiers sont enchaînés, il est facile de les séparer, en commençant par le premier si ils sont petits boutiens (chiffre de moindre poids en premier) ou en commençant par le dernier si ils sont gros boutiens (chiffre de poids fort en premier). Comme précédemment, la valeur est la somme sur j de chiffre_j * w(j), mais les poids sont différents :

w(0) = 1
w(j) = w(j-1) * (base - t(j-1)) pour j > 0

Par exemple, considérons la séquence petite boutienne de base 8 des chiffres 734251... Supposons que les seuils sont 2, 3, 5, 5, 5, 5... Cela implique que les poids sont 1, 1*(8-2) = 6, 6*(8-3) = 30, 30*(8-5) = 90, 90*(8-5) = 270, et ainsi de suite. 7 n'est pas moins que 2, et 3 n'est pas moins que 3, mais 4 est moins que 5, de sorte que 4 est le dernier chiffre. La valeur de 734 est 7*1 + 3*6 + 4*30 = 145. Le prochain entier est 251, avec la valeur 2*1 + 5*6 + 1*30 = 62. Décoder cette représentation est très similaire au décodage d'un entier conventionnel : on commence par la valeur courante de N = 0 et un poids w = 1. On va chercher le prochain chiffre d et on augmente N de d * w. Si d est inférieur au seuil (t) actuel, on arrête, autrement, on augmente w d'un facteur de (base - t), on met à jour t pour la position suivante, et on répète.

Coder cette représentation est similaire au codage d'un entier conventionnel : si N < t, on sort un chiffre pour N et on arrête, autrement on sort le chiffre pour t + ((N - t) mod (base - t)), puis on remplace N par (N - t) div (base - t), on met à jour t pour la position suivante, et on répète.

Pour tout ensemble particulier de valeurs de t(j), il y a exactement une représentation de longueur variable généralisée de chaque valeur entière non négative.

Bootstring utilise l'ordre petit boutien de sorte que les deltas puissent être séparés en commençant par le premier. Les valeurs t(j) sont définies en termes de base constante, tmin, et tmax, et une variable d'état appelée biais :

t(j) = base * (j + 1) - biais, dans la gamme de tmin à tmax

Les limites signifient que si la formule donne une valeur inférieure à tmin ou supérieure à tmax, alors, t(j) = tmin ou tmax, respectivement. (Dans le pseudocode de la Section 6 "Algorithmes Bootstring", l'expression base * (j + 1) est notée par k pour des raisons de performances.) Ces valeurs t(j) font que la représentation favorise les entiers dans une gamme particulière déterminée par le biais.

3.4 Adaptation du biais

Après le codage ou décodage de chaque delta, le biais est réglé pour le prochain delta comme suit :

1. Le delta est adapté afin d'éviter un débordement dans l'étape suivante :

   soit delta = delta div 2

Mais lorsque c'est le tout premier delta, le diviseur n'est pas 2, mais plutôt une constante appelée damp *(frein)*. Cela compense le fait que le second delta est usuellement beaucoup plus petit que le premier.

2. Le delta est augmenté pour compenser le fait que le prochain delta sera inséré dans une chaîne plus longue :

   soit delta = delta + (delta div numpoints)

numpoints est le nombre total de codets codés/décodés jusqu'alors (incluant celui qui correspond à ce delta lui-même, et incluant les codets de base).

3. Le delta est divisé de façon répétée jusqu'à ce qu'il entre dans un seuil, pour prédire le nombre minimum de chiffres nécessaire pour représenter le prochain delta :

lorsque delta > ((base - tmin) * tmax) div 2
alors delta = delta div (base - tmin)

4. Le biais est réglé :

soit biais = (base * le nombre de divisions effectuées à l'étape 3) + (((base - tmin + 1) * delta) div (delta + skew))

Le motif de cette procédure est que le delta actuel donne une indication de la taille probable du prochain delta, et donc t(j) est réglé à tmax pour les chiffres de poids fort commençant par celui dont on s'attend à ce qu'il soit le dernier, à tmin pour le chiffre de moindre poids jusqu'à celui dont on s'attend à ce qu'il soit l'antépénultième, et quelque part entre tmin et tmax pour le chiffre dont on s'attend à ce qu'il soit l'avant dernier (trouvant un équilibre entre l'espoir que le dernier chiffre attendu ne soit pas nécessaire et le danger qu'il soit insuffisant).

4. Paramètres Bootstring

Étant donné un ensemble de codets de base, l'un d'eux doit être désigné comme délimiteur. La base ne peut pas être supérieure au nombre distinguable de codets de base restants. Les valeurs de chiffres dans la gamme 0 à base-1 doivent être associées aux codets de base non délimiteur distincts. Dans certains cas, plusieurs codets doivent avoir la même valeur de chiffre ; par exemple, les versions majuscule et minuscule de la même lettre doivent être équivalentes si les chaînes de base sont insensibles à la casse.

La valeur initiale de n ne peut pas être supérieure au codet non de base minimum qui pourrait apparaître dans les chaînes étendues.

Les cinq paramètres restants (tmin, tmax, skew, damp, et la valeur initiale de biais) doivent satisfaire les contraintes suivantes :
0 ≤ tmin ≤ tmax <= base-1
skew ≥ 1
damp ≥ 2
initial_bias mod base ≤ base - tmin

Pourvu que les contraintes soient satisfaites, ces cinq paramètres affectent l'efficacité mais pas la correction. Ils sont mieux choisis de façon empirique.

Si on désire prendre en charge des annotations de casse mixte (voir l'Appendice A) on s'assurera que les codets correspondants à 0 à tmax-1 ont tous les formes majuscules et minuscules.

5. Valeurs des paramètres pour Punycode

Punycode utilise les valeurs de paramètres Bootstring suivantes :
base = 36
tmin = 1
tmax = 26
skew = 38
damp = 700
initial_bias = 72
initial_n -= 128 = 0x80

Bien que la seule restriction que Punycode impose aux entiers d'entrée soit qu'ils ne soient pas négatifs, ces paramètres sont spécialement conçus pour bien fonctionner avec les codets Unicode [UNICODE], qui sont entiers dans la gamme 0 à 10FFFF (mais pas D800 à DFFF, qui sont réservés pour l'utilisation du codage UTF-16 de Unicode). Les codets de base sont les codets ASCII [RFC0020] (0 à 7F), dont U+002D (-) est le délimiteur, et certains des autres ont les valeurs de chiffres suivantes :

codets	**valeurs numériques**
41 à 5A (A-Z)	= 0 à 25, respectivement
61 à 7A (a-z)	= 0 à 25, respectivement
30 à 39 (0-9)	= 26 à 35, respectivement

L'utilisation du signe trait d'union (moins) comme délimiteur implique que la chaîne codée ne peut se terminer par un trait d'union que si la chaîne Unicode consiste entièrement en codets de base, mais IDNA interdit que de telles chaînes soient codées. La chaîne codée peut commencer par un trait d'union, mais IDNA ajoute devant un préfixe. Donc, IDNA utilisant

Punycode se conforme à la règle de la RFC0952 qui veut que les étiquettes de nom d'hôte ne commencent ni ne finissent par un trait d'union [RFC0952].

Un décodeur DOIT reconnaître les lettres dans les deux formes majuscules et minuscules (y compris les mélanges des deux formes). Un codeur DEVRAIT produire des formes seulement majuscules ou seulement minuscules, sauf si il utilise des annotation de casse mixte (voir l'Appendice A).

On peut présumer que la plupart des utilisateurs ne vont pas écrire ou taper manuellement les chaînes codées (par opposition à en faire un copié collé) mais ceux qui le font auront besoin d'être conscients de l'ambiguïté visuelle potentielle des ensembles de caractères suivants :

G 6
I l 1
O 0
S 5
U V
Z 2

De telles ambiguïtés sont usuellement résolues par le contexte, mais dans une chaîne codée en Punycode, aucun contexte n'est apparent pour l'homme.

6. Algorithmes Bootstring

Certaines parties du pseudocode peuvent être omises si les paramètres satisfont certaines conditions (pour lesquelles Punycode est qualifié). Ces parties sont entre des accolades { }, et les notes qui suivent immédiatement le pseudocode expliquent les conditions dans lesquelles elles peuvent être omises.

Formellement, les codets sont des entiers, et donc le pseudocode suppose que des opérations arithmétiques peuvent être effectuées directement sur les codets. Dans certains langages de programmation, une conversion explicite entre codets et entiers peut être nécessaire.

6.1 Fonction d'adaptation de biais

```
fonction adapt(delta,numpoints,premièrefois) :
  si premièrefois alors soit delta = delta div damp
  autrement soit delta = delta div 2
  soit delta = delta + (delta div numpoints)
  soit k = 0
  tandis que delta > ((base - tmin) * tmax) div 2 faire début
    soit delta = delta div (base - tmin)
    soit k = k + base
  fin
  retourner k + (((base - tmin + 1) * delta) div (delta + skew))
```

Il importe peu que les modifications à delta et k à l'intérieur de adapt() affectent des variables du même nom dans les procédures de codage/décodage, parce qu'après l'invocation de adapt() l'invocateur ne lit plus ces variables avant de les réécrire.

6.2 Procédure de décodage

```
soit n = initial_n
soit i = 0
soit biais = initial bias
soit résultat = une chaîne vide indexée à partir de 0 consomme tous les codets avant le dernier délimiteur (si il y en a un)
  et les copie en sortie, échec sur tout codet non de base
si plus de zéro codets ont été consommés, alors en consommer un de plus (qui sera le dernier délimiteur)
tandis que l'entrée n'est pas épuisée, faire début
  soit vieuxi= i
  soit w = 1
```

```
  pour k = base à infini dans les étapes de base faire début
    consommer un codet, ou échec si il n'y en a pas à consommer
    soit chiffre = la valeur numérique du codet, échec si il n'en a pas
    soit i = i + chiffre * w, échec sur débordement
    soit t = tmin si k <= biais {+ tmin}, ou
           tmax si k >= biais + tmax, ou k - biais autrement
    si chiffre < t alors coupure
    soit w = w * (base - t), échec sur débordement
  fin
  soit biais = adapt(i - vieuxi, longueur(résultat) + 1, vérifier si vieuxi est 0 ?)
  soit n = n + i div (longueur(résultat) + 1), échec sur débordement
  soit i = i mod (longueur(résultat) + 1)
  {si n est un codet de base alors échec}
  insérer n dans résultat à la position i
  incrémenter i
fin
```

Toute la déclaration entre les accolades (vérifier si n est un codet de base) peut être omise si initial_n excède tous les codets de base (ce qui est vrai pour Punycode) parce que n n'est jamais inférieur à initial_n.

Dans l'allocation de t, où t est fixé dans la gamme de tmin à tmax, "+ tmin" peut toujours être omis. Cela rend le calcul des limites incorrect lorsque biais < k < biais + tmin, mais cela ne peut pas arriver à cause de la façon dont le biais est calculé et à cause des contraintes sur les paramètres.

Comme l'état du décodeur ne peut avancer que de façon monotone, il y a seulement une représentation d'un delta, il n'y a donc qu'une seule chaîne codée qui puisse représenter une certaine séquence d'entiers. Les seules conditions d'erreur sont les codets invalides, les fins d'entrée inattendues, les débordements, et les codets de base codés en utilisant les deltas au lieu d'apparaître littéralement. Si le décodeur échoue sur ces erreurs comme montré ci-dessus, alors il ne peut pas produire le même résultat pour deux entrées distinctes. Sans cette propriété, il aurait été nécessaire de recoder le résultat et de vérifier qu'il correspond à l'entrée afin de garantir l'unicité du codage.

6.3 Procédure de codage

```
soit n = initial_n
soit delta = 0
soit biais = initial bias
soit h = b = le nombre de codets de base dans l'entrée
les copier au résultat dans l'ordre, suivis par un délimiteur si b > 0
{si l'entrée contient un codet non de base < n alors échec}
tandis que h < longueur(entrée) faire début
  soit m = le codet minimum {non de base} ≥ n dans l'entrée
  soit delta = delta + (m - n) * (h + 1), échec sur débordement
  soit n = m
  pour chaque codet c dans l'entrée (dans l'ordre) faire début
    si c < n {ou c est de base} alors incrémenter delta, échec sur débordement
    si c == n alors début
      soit q = delta
      pour k = base à infini dans les étapes de base faire début
        soit t = tmin si k ≤ biais {+ tmin}, ou
               tmax si k ≤ biais + tmax, ou k - biais autrement
        si q < t alors coupure
        entrer le codet pour chiffre t + ((q - t) mod (base - t))
        soit q = (q - t) div (base - t)
      fin
      entrer le codet pour chiffre q
      soit biais = adapt(delta, h + 1, vérifier que h égale b ?)
      soit delta = 0
      incrémenter h
    fin
  fin
  incrémenter delta et n
```

fin

Toute la déclaration enclose entre des accolades (vérifier si l'entrée contient un codet non de base inférieur à n) peut être omis si tous les codets inférieurs à initial_n sont des codets de base (ce qui est vrai pour Punycode si les codets sont non signés).

Les conditions "non-basic" et "ou c est de base" incluses entre des accolades peuvent être omises si initial_n excède tous les codets de base (ce qui est vrai pour Punycode) parce que le codet vérifié n'est jamais inférieur à initial_n.

Dans l'allocation de t, où t est fixé à la gamme de tmin à tmax, "+ tmin" peut toujours être omis. Cela rend le calcul des limites incorrect lorsque biais < k < biais + tmin, mais cela ne peut pas arriver à cause de la façon dont le biais est calculé et à cause des contraintes sur les paramètres.

Les vérifications de débordement sont nécessaires pour éviter de produire un résultat invalide lorsque l'entrée contient de très grandes valeurs ou est très longue.

L'incrément de delta à la fin de la boucle externe ne peut pas déborder parce que delta < longueur(entrée) avant l'incrément, et longueur(entrée) est déjà supposé être représentable. L'incrément de n pourrait déborder, mais seulement si h == longueur(entrée) auquel cas la procédure est finie de toutes façons.

6.4 Traitement des débordements

Pour IDNA, des entiers non signés de 26 bits sont suffisants pour traiter toutes les étiquettes IDNA valides sans débordement, parce que toute chaîne qui a besoin d'un delta de 27 bits devrait excéder soit la limite de codet (0 à 10FFFF) soit la limite de longueur d'étiquette (63 caractères). Cependant, le traitement de débordement est nécessaire parce que les entrées ne sont pas nécessairement des étiquettes IDNA valides.

Si le langage de programmation ne fournit pas de détection de débordement, la technique suivante peut être utilisée. Supposons que A, B, et C soient des entiers non négatifs représentables et que C soit différent de zéro. Alors A + B déborde si et seulement si B > maxint - A, et A + (B * C) déborde si et seulement si B > (maxint - A) div C, où maxint est la valeur maximum d'une variable entière. Se référer à l'Appendice C "Exemple de mise en œuvre de Punycode" pour des démonstrations de cette technique en langage C.

Les algorithmes de décodage et de codage montrés aux paragraphes 6.2 et 6.3 traitent le débordement en le détectant chaque fois qu'il se produit. Une autre approche est d'appliquer des limites aux entrées qui empêchent le débordement de se produire. Par exemple, si le codeur devait vérifier qu'aucun codet d'entrée n'excède M et si la longueur d'entrée n'excède pas L, alors aucun delta ne pourra jamais dépasser (M - initial_n) * (L + 1) et donc aucune débordement ne pourra survenir si des variables entières étaient capables de représenter des valeurs de cette taille. Cette approche préventive imposerait plus de restrictions sur les entrées que ne le fait l'approche de détection, mais pourrait être considérée comme plus simple dans certains langages de programmation.

En théorie, le décodeur pourrait utiliser une approche analogue, en limitant le nombre de chiffres dans un entier de longueur variable (c'est-à-dire, en limitant le nombre d'itérations dans la boucle la plus interne). Cependant, le nombre de chiffres qui suffisent à représenter un certain delta peut parfois représenter des deltas beaucoup plus grands (à cause de l'adaptation) et donc cette approche exigerait probablement des entiers plus grands que 32 bits.

Une autre approche pour le décodeur est de permettre au débordement de se produire, mais de vérifier la chaîne de résultat finale en la recodant et en la comparant à l'entrée du décodeur. Si et seulement si ils ne correspondent pas (en utilisant une comparaison ASCII insensible à la casse) un débordement s'est produit. Cette approche de détection à retardement n'imposerait pas plus de restrictions à l'entrée que l'approche de la détection immédiate, et pourrait être considérée comme plus simple dans certains langages de programmation.

En fait, si le décodeur n'est utilisé qu'à l'intérieur de l'opération ToUnicode d'IDNA [RFC3490], alors il n'y a pas besoin de vérification du tout pour le débordement, parce que ToUnicode effectue un recodage et une comparaison de niveau supérieur, et une discordance a les mêmes conséquences que si le décodeur Punycode avait échoué.

7. Exemples de Punycode

7.1 Exemples de chaînes

Dans les codages Punycode ci-dessous, le préfixe ACE n'est pas montré. Des barres obliques inverses montrent où les coupures de lignes ont été insérées dans les chaînes trop longues pour une ligne.

Les premiers exemples sont tous la traduction de la phrase "Pourquoi ne parlent ils pas un <langage>?" (emprunté à la page "provincial" de Michael Kaplan avec sa permission [PROVINCIAL]). Les coupures de mots et la ponctuation ont été retirées, comme c'est souvent fait dans les noms de domaines.

(A) Arabe (Égyptien) :
u+0644 u+064A u+0647 u+0645 u+0627 u+0628 u+062A u+0643 u+0644
u+0645 u+0648 u+0634 u+0639 u+0631 u+0628 u+064A u+061F
Punycode: egbpdaj6bu4bxfgehfvwxn

(B) Chinois (simplifié) :
u+4ED6 u+4EEC u+4E3A u+4EC0 u+4E48 u+4E0D u+8BF4 u+4E2D u+6587
Punycode: ihqwcrb4cv8a8dqg056pqjye

(C) Chinois (traditionnel) :
u+4ED6 u+5011 u+7232 u+4EC0 u+9EBD u+4E0D u+8AAA u+4E2D u+6587
Punycode: ihqwctvzc91f659drss3x8bo0yb

(D) Tchèque : Pro<ccaron>prost<ecaron>nemluv<iacute><ccaron>esky
U+0050 u+0072 u+006F u+010D u+0070 u+0072 u+006F u+0073 u+0074
u+011B u+006E u+0065 u+006D u+006C u+0075 u+0076 u+00ED u+010D
u+0065 u+0073 u+006B u+0079
Punycode: Proprostnemluvesky-uyb24dma41a

(E) Hébreu :
u+05DC u+05DE u+05D4 u+05D4 u+05DD u+05E4 u+05E9 u+05D5 u+05D8
u+05DC u+05D0 u+05DE u+05D3 u+05D1 u+05E8 u+05D9 u+05DD u+05E2
u+05D1 u+05E8 u+05D9 u+05EA
Punycode: 4dbcagdahymbxekheh6e0a7fei0b

(F) Hindi (Devanagari) :
u+092F u+0939 u+0932 u+094B u+0917 u+0939 u+093F u+0928 u+094D
u+0926 u+0940 u+0915 u+094D u+092F u+094B u+0902 u+0928 u+0939
u+0940 u+0902 u+092C u+094B u+0932 u+0938 u+0915 u+0924 u+0947
u+0939 u+0948 u+0902
Punycode: i1baa7eci9glrd9b2ae1bj0hfcgg6iyaf8o0a1dig0cd

(G) Japonais (kanji et hiragana) :
u+306A u+305C u+307F u+3093 u+306A u+65E5 u+672C u+8A9E u+3092
u+8A71 u+3057 u+3066 u+304F u+308C u+306A u+3044 u+306E u+304B
Punycode: n8jok5ay5dzabd5bym9f0cm5685rrjetr6pdxa

(H) Coréen (syllabes Hangul) :
u+C138 u+ACC4 u+C758 u+BAA8 u+B4E0 u+C0AC u+B78C u+B4E4 u+C774
u+D55C u+AD6D u+C5B4 u+B97C u+C774 u+D574 u+D55C u+B2E4 u+BA74
u+C5BC u+B9C8 u+B098 u+C88B u+C744 u+AE4C
Punycode: 989aomsvi5e83db1d2a355cv1e0vak1dwrv93d5xbh15a0dt30a5jpsd879ccm6fea98c

(I) Russe (Cyrillique) :
U+043F u+043E u+0447 u+0435 u+043C u+0443 u+0436 u+0435 u+043E
u+043D u+0438 u+043D u+0435 u+0433 u+043E u+0432 u+043E u+0440
u+044F u+0442 u+043F u+043E u+0440 u+0443 u+0441 u+0441 u+043A
u+0438
Punycode: b1abfaaepdrnnbgefbadotcwatmq2g4l

(J) Espagnol : Porqu<eacute>nopuedensimplementehablarenEspa<ntilde>ol

U+0050 u+006F u+0072 u+0071 u+0075 u+00E9 u+006E u+006F u+0070
u+0075 u+0065 u+0064 u+0065 u+006E u+0073 u+0069 u+006D u+0070
u+006C u+0065 u+006D u+0065 u+006E u+0074 u+0065 u+0068 u+0061
u+0062 u+006C u+0061 u+0072 u+0065 u+006E U+0045 u+0073 u+0070
u+0061 u+00F1 u+006F u+006C
Punycode: PorqunopuedensimplementehablarenEspaol-fmd56a

(K) Vietnamien :
T<adotbelow>isaoh<odotbelow>kh<ocirc>ngth<ecirchookabove>ch\
<ihookabove>n<oacute>iti<ecircacute>ngVi<ecircdotbelow>t
U+0054 u+1EA1 u+0069 u+0073 u+0061 u+006F u+0068 u+1ECD u+006B
u+0068 u+00F4 u+006E u+0067 u+0074 u+0068 u+1EC3 u+0063 u+0068
u+1EC9 u+006E u+00F3 u+0069 u+0074 u+0069 u+1EBF u+006E u+0067
U+0056 u+0069 u+1EC7 u+0074
Punycode: TisaohkhngthchnitingVit-kjcr8268qyxafd2f1b9g

Les exemples suivants sont tous des noms de chanteurs japonais, des titres de chansons, et de programmes de télévision, simplement parce qu'il se trouve que l'auteur les a sous la main (mais le japonais est utile pour fournir des exemples de texte sur une colonne, sur deux colonnes, du texte idéographique, et leurs divers mélanges).

(L) 3<nen>B<gumi><kinpachi><sensei>
u+0033 u+5E74 U+0042 u+7D44 u+91D1 u+516B u+5148 u+751F
Punycode: 3B-ww4c5e180e575a65lsy2b

(M) <amuro><namie>-with-SUPER-MONKEYS
u+5B89 u+5BA4 u+5948 u+7F8E u+6075 u+002D u+0077 u+0069 u+0074
u+0068 u+002D U+0053 U+0055 U+0050 U+0045 U+0052 u+002D U+004D
U+004F U+004E U+004B U+0045 U+0059 U+0053
Punycode: -with-SUPER-MONKEYS-pc58ag80a8qai00g7n9n

(N) Hello-Another-Way-<sorezore><no><basho>
U+0048 u+0065 u+006C u+006C u+006F u+002D U+0041 u+006E u+006F
u+0074 u+0068 u+0065 u+0072 u+002D U+0057 u+0061 u+0079 u+002D
u+305D u+308C u+305E u+308C u+306E u+5834 u+6240
Punycode: Hello-Another-Way--fc4qua05auwb3674vfr0b

(O) <hitotsu><yane><no><shita>2
u+3072 u+3068 u+3064 u+5C4B u+6839 u+306E u+4E0B u+0032
Punycode: 2-u9tlzr9756bt3uc0v

(P) Maji<de>Koi<suru>5<byou><mae>
U+004D u+0061 u+006A u+0069 u+3067 U+004B u+006F u+0069 u+3059
u+308B u+0035 u+79D2 u+524D
Punycode: MajiKoi5-783gue6qz075azm5e

(Q) <pafii>de<runba>
u+30D1 u+30D5 u+30A3 u+30FC u+0064 u+0065 u+30EB u+30F3 u+30D0
Punycode: de-jg4avhby1noc0d

(R) <sono><supiido><de>
u+305D u+306E u+30B9 u+30D4 u+30FC u+30C9 u+3067
Punycode: d9juau41awczczp

Le dernier exemple est une chaîne ASCII qui rompt avec les règles existantes pour les étiquettes de nom d'hôte. (Ce n'est pas un exemple réaliste pour IDNA, parce que IDNA ne code jamais des étiquettes de pur ASCII.)

(S) -> $1.00 <-
u+002D u+003E u+0020 u+0024 u+0031 u+002E u+0030 u+0030 u+0020
u+003C u+002D
Punycode: -> $1.00 <--

7.2 Traces de décodage

Dans les traces qui suivent, l'évolution de l'état du décodeur est montrée comme une séquence de valeurs hexadécimales, représentant les codets dans la chaîne étendue. Un astérisque apparaît juste après le codet inséré le plus récent, indiquant à la fois n (la valeur précédant l'astérisque) et i (la position de la valeur juste après l'astérisque). Les autres valeurs numériques sont décimales.

Trace de décodage de l'exemple B du paragraphe 7.1

```
n est 128, i est 0, le biais est de 72
  l'entrée est "ihqwcrb4cv8a8dqg056pqjye"
   il n'y a pas de délimiteur, de sorte que la chaîne étendue commence vide
   delta "ihq" se décode en 19853
   le biais devient 21
   4E0D *
   delta "wc" se décode en 64
   le biais devient 20
   4E0D 4E2D *
   delta "rb" se décode en 37
   le biais devient 13
   4E3A * 4E0D 4E2D
   delta "4c" se décode en 56
   le biais devient 17
   4E3A 4E48 * 4E0D 4E2D
   delta "v8a" se décode en 599
   le biais devient 32
   4E3A 4EC0 * 4E48 4E0D 4E2D
   delta "8d" se décode en 130
   le biais devient 23
   4ED6 * 4E3A 4EC0 4E48 4E0D 4E2D
   delta "qg" se décode en 154
   le biais devient 25
   4ED6 4EEC * 4E3A 4EC0 4E48 4E0D 4E2D
   delta "056p" se décode en 46301
   le biais devient 84
   4ED6 4EEC 4E3A 4EC0 4E48 4E0D 4E2D 6587 *
   delta "qjye" se décode en 88531
   le biais devient 90
   4ED6 4EEC 4E3A 4EC0 4E48 4E0D 8BF4 * 4E2D 6587
```

Trace de décodage de l'exemple L du paragraphe 7.1 :

```
n est 128, i est 0, le biais est de 72
   l'entrée est "3B-ww4c5e180e575a65lsy2b"
   la portion littérale est "3B-", de sorte que la chaîne étendue commence par :
   0033 0042
   delta "ww4c" se décode en 62042
   le biais devient 27
   0033 0042 5148 *
   delta "5e" se décode en 139
   le biais devient 24
   0033 0042 516B * 5148
   delta "180e" se décode en 16683
   le biais devient 67
   0033 5E74 * 0042 516B 5148
   delta "575a" se décode en 34821
   le biais devient 82
   0033 5E74 0042 516B 5148 751F *
   delta "65l" se décode en 14592
   le biais devient 67
   0033 5E74 0042 7D44 * 516B 5148 751F
   delta "sy2b" se décode en 42088
```

le biais devient 84
0033 5E74 0042 7D44 91D1 * 516B 5148 751F

7.3 Traces de codage

Dans les traces suivantes, les valeurs des codets sont hexadécimales, tandis que les autres valeurs numériques sont décimales.

Trace de codage de l'exemple B du paragraphe 7.1 :

le biais est de 72
l'entrée est : 4ED6 4EEC 4E3A 4EC0 4E48 4E0D 8BF4 4E2D 6587
il n'y a pas de codet de base, et donc pas de portion littérale
le prochain codet à insérer est 4E0D
le delta nécessaire est 19853, qui est codé par "ihq"
le biais devient 21
le prochain codet à insérer est 4E2D
le delta nécessaire est 64, qui est codé par "wc"
le biais devient 20
le prochain codet à insérer est 4E3A
le delta nécessaire est 37, qui est codé par "rb"
le biais devient 13
le prochain codet à insérer est 4E48
le delta nécessaire est 56, qui est codé par "4c"
le biais devient 17
le prochain codet à insérer est 4EC0
le delta nécessaire est 599, qui est codé par "v8a"
le biais devient 32
le prochain codet à insérer est 4ED6
le delta nécessaire est 130, qui est codé par "8d"
le biais devient 23
le prochain codet à insérer est 4EEC
le delta nécessaire est 154, qui est codé par "qg"
le biais devient 25 le prochain codet à insérer est 6587
le delta nécessaire est 46301, qui est codé par "056p"
le biais devient 84
le prochain codet à insérer est 8BF4
le delta nécessaire est 88531, qui est codé par "qjye"
le biais devient 90
le résultat est "ihqwcrb4cv8a8dqg056pqjye"

Trace de codage de l'exemple L du paragraphe 7.1 :

le biais est de 72
l'entrée est : 0033 5E74 0042 7D44 91D1 516B 5148 751F
les codets de base (0033, 0042) sont copiés à la portion littérale : "3B-"
le prochain codet à insérer est 5148
le delta nécessaire est 62042, qui est codé par "ww4c"
le biais devient 27
le prochain codet à insérer est 516B
le delta nécessaire est 139, qui est codé par "5e"
le biais devient 24
le prochain codet à insérer est 5E74
le delta nécessaire est 16683, qui est codé par "180e"
le biais devient 67
le prochain codet à insérer est 751F
le delta nécessaire est 34821, qui est codé par "575a"
le biais devient 82
le prochain codet à insérer est 7D44
le delta nécessaire est 14592, qui est codé par "65l"
le biais devient 67
le prochain codet à insérer est 91D1

le delta nécessaire est 42088, qui est codé par "sy2b"
le biais devient 84
le résultat est "3B-ww4c5e180e575a65lsy2b"

8. Considérations sur la sécurité

Les utilisateurs s'attendent à ce que chaque nom de domaine du DNS soit contrôlé par une seule autorité. Si une chaîne Unicode destinée à être utilisée comme étiquette de domaine pouvait se transposer en plusieurs étiquettes ACE, un nom de domaine internationalisé pourrait alors se transposer en plusieurs noms de domaine ASCII, chacun contrôlé par une autorité différente, dont certains pourraient être des mystifications qui capturent les demandes de service destinées à d'autres. Donc, Punycode est conçu de telle sorte que chaque chaîne Unicode ait un codage univoque.

Cependant, il peut quand même y avoir des représentations Unicode multiples du "même" texte, pour diverses définitions de "même". Ce problème est traité dans une certaine mesure par la norme Unicode sous le thème de la canonisation, et ce travail est renforcé pour les noms de domaines par Nameprep [RFC3491].

9. Références

[RFC0020] V. Cerf, "Format ASCII pour les échanges sur les réseaux", octobre 1969.

[RFC0952] K. Harrenstien, M. Stahl, E. Feinler, "Spécification du tableau des hôtes de l'Internet du DOD", octobre 1985.

[RFC1034] P. Mockapetris, "Noms de domaines - Concepts et facilités", STD 13, novembre 1987.

[RFC2119] S. Bradner, "Mots clés à utiliser dans les RFC pour indiquer les niveaux d'exigence", BCP 14, mars 1997.

[RFC3490] P. Faltstrom et autres, "Internationalisation des noms de domaine dans les applications (IDNA)", mars 2003. (*Remplacée par les* RFC5890 et 5891, *D.S.*)

[RFC3491] P. Hoffman et M. Blanchet, "Nameprep : Profil Stringprep pour les noms de domaine internationalisés (IDN)", mars 2003.

[PROVINCIAL] Kaplan, M., "The 'anyone can be provincial!' page", http://www.trigeminal.com/samples/provincial.html .

[UNICODE] The Unicode Consortium, "The Unicode Standard", http://www.unicode.org/unicode/standard/standard.html .

A. Annotation en casse mixte

Pour utiliser Punycode à représenter des chaînes insensibles à la casse, les couches supérieures doivent normaliser la casse avant le codage Punycode. La chaîne codée peut utiliser une casse mixte en annotation pour dire comment convertir la chaîne normalisée en une chaîne à casse mixte pour les besoins de l'affichage. Noter cependant qu'une annotation en casse mixte n'est pas utilisée par les opérations ToASCII et ToUnicode spécifiées dans la [RFC3490], et donc que les mises en œuvre de IDNA peuvent oublier le présent appendice.

Les codets de base peuvent utiliser directement une casse mixte, parce que le décodeur les copie textuellement, laissant en minuscule les codets en minuscules, et en majuscule les codets en majuscules. Chaque codet non de base est représenté par un delta, qui est représenté par une séquence de codets de base, dont le dernier fournit l'annotation. Si il est en majuscules, il suggère de transposer le codet non de base en majuscules (si possible) ; si il est en minuscules, il suggère de transposer le codet non de base en minuscules (si possible).

Ces annotations n'altèrent pas les codets retournés par les décodeurs ; les annotations sont retournées séparément, pour que l'appelant les utilise ou les ignore. Les codeurs peuvent accepter les annotations en plus des codets, mais les annotations n'altèrent pas le résultat, sauf à influencer la forme majuscules/minuscules des lettres ASCII.

Les codeurs et décodeurs Punycode n'ont pas besoin de prendre en charge ces annotations, et les couches supérieures n'ont pas besoin de les utiliser.

B. Déclinatoire de responsabilité et licence

En ce qui concerne le présent document dans son entier ou ses parties (y compris le pseudocode de code C) l'auteur ne donne aucune garantie et ne peut être tenu pour responsable d'aucun dommage résultant de son utilisation. L'auteur accorde une permission irrévocable à quiconque de l'utiliser, le modifier, et le distribuer de toutes façons qui ne diminuent pas les droits d'utilisation, modification et distribution d'autrui, pour autant que les travaux de redistribution dérivés ne contiennent pas d'informations fallacieuses sur l'auteur ou la version. Les travaux dérivés n'ont pas besoin de faire l'objet d'une licence dans des termes similaires.

C. Mise en œuvre d'échantillon de Punycode

```
/*
punycode.c d'après la RFC 3492
http://www.nicemice.net/idn/
Adam M. Costello
http://www.nicemice.net/amc/

Ceci est du code C ANSI (C89) qui met en œuvre Punycode (RFC3492).

*/

/************************************************************/
/* Interface publique (irait normalement dans son propre fichier .h) : */

#include <limits.h>

enum punycode_status {
  punycode_success,
  punycode_bad_input,           /* L'entrée est invalide.*/
  punycode_big_output,          /* Le résultat va excéder l'espace fourni. */
  punycode_overflow             /* L'entrée a besoin de plus grands entiers pour continuer.  */
};

#if UINT_MAX >= (1 << 26) - 1
typedef unsigned int punycode_uint;
#else
typedef unsigned long punycode_uint;
#endif

enum punycode_status punycode_encode(
  punycode_uint input_length,
  const punycode_uint input[],
  const unsigned char case_flags[],
  punycode_uint *output_length,
  char output[] );

/* punycode_encode() convertit Unicode en Punycode. L'entrée est représentée comme un dispositif de codets Unicode (pas
d'unités de code ; les paires de substitution ne sont pas permises) et le résultat va être représenté par un dispositif de codets
ASCII. La chaîne résultante *n'est pas* terminée par un caractère nul ; elle ne va contenir des zéros que si et seulement si
l'entrée contient des zéros. (Bien sûr l'invocateur peut laisser de la place pour une terminaison et en ajouter une si nécessaire.)
La longueur d'entrée est le nombre de codets dans l'entrée. La longueur de résultat est un argument entrée/sortie : l'invocateur
y entre le nombre maximum de codets qu'il peut recevoir, et sur un retour réussi, il va contenir le nombre de codets réellement
sortis. Le dispositif case_flags (fanions de casse) détient les valeurs booléennes de longueur d'entrée, où non zéro suggère que
les caractères Unicode correspondants soient forcés en majuscules après décodage (si possible), et zéro suggère qu'ils soient
forcés en minuscules (si possible). Les codets ASCII sont codés littéralement, excepté que les lettres ASCII sont forcées en
majuscules ou minuscules selon les fanions correspondants. Si case_flags est un pointeur nul, alors les lettres ASCII sont
laissées comme elles sont, et les autres codets sont traités comme si leur fanion de casse était à zéro. La valeur de retour peut
être toute valeur de punycode_status (état Punycode) définie ci-dessus sauf punycode_bad_input ; si ce n'est pas
punycode_success, alors result_size (taille de résultat) et result (résultat) peuvent contenir des aberrations. */

enum punycode_status punycode_decode(
```

```
  punycode_uint input_length,
  const char input[],
  punycode_uint *output_length,
  punycode_uint output[],
  unsigned char case_flags[] );
```

/* punycode_decode() convertit Punycode en Unicode. L'entrée est représentée comme un dispositif de codets ASCII, et le résultat sera représenté par un dispositifs de codets Unicode. Le paramètre input_length est le nombre de codets dans l'entrée. Le paramètre output_length est un argument entrée/sortie : l'invocateur y entre le nombre maximum de codets qu'il peut recevoir, et pour un retour réussi il va contenir le nombre réel de codets de résultat. Le dispositif case_flags doit avoir de la place pour au moins les valeurs de longueur de résultat, ou il peut être un pointeur nul si les informations de casse ne sont pas nécessaires. Un fanion non zéro suggère que le caractère Unicode correspondant soit forcé en majuscules par l'appelant (si possible) tandis que zéro suggère qu'il soit forcé en minuscules (si possible). Les codets ASCII sont déjà sortis dans la casse appropriée, mais leurs fanions seront réglé de façon appropriée pour que l'application des fanions soit sans dommage. La valeur de retour peut être toute valeur de punycode_status définie ci-dessus ; si ce n'est pas punycode_success, alors result_length, résult, et case_flags peuvent contenir des aberrations. En cas de réussite, le décodeur n'aura jamais besoin d'écrire une longueur de résultat plus grande que la longueur d'entrée, à cause de la façon dont le codage est défini. */

```
/**********************************************************/
/* Mise en œuvre (va normalement aller dans son propre fichier .c) : */

#include <string.h>

/*** Paramètres Bootstring pour Punycode ***/

enum { base = 36, tmin = 1, tmax = 26, skew = 38, damp = 700, initial_bias = 72, initial_n = 0x80, delimiter = 0x2D };

/* Vérifications sur basic(cp) pour voir si cp est un codet de base : */
#define basic(cp) ((punycode_uint)(cp) < 0x80)

/* Vérifications sur delim(cp) pour voir si cp est un délimiteur : */
#define delim(cp) ((cp) == delimiter)

/* decode_digit(cp) retourne la valeur numérique d'un codet de base (à utiliser pour représenter des entiers) dans la gamme 0 à
base-1, ou base si cp ne représente pas une valeur. */

static punycode_uint decode_digit(punycode_uint cp)
{
 return  cp - 48 < 10 ? cp - 22 :  cp - 65 < 26 ? cp - 65 :  cp - 97 < 26 ? cp - 97 :  base;
}

/* encode_digit(d,flag) retourne le codet de base dont la valeur (lorsque utilisé pour représenter des entiers) est d, qui doit être
dans la gamme de 0 à base-1. La forme minuscules est utilisée sauf si le fanion n'est pas zéro, auquel cas la forme majuscule
est utilisée. Le comportement est indéfini si le fanion n'est pas zéro et le chiffre d n'a pas de forme majuscule. */

static char encode_digit(punycode_uint d, int flag)
{
 return d + 22 + 75 * (d < 26) - ((flag != 0) << 5);
                                        /* 0 à 25 se transpose en ASCII a..z ou A..Z */
                                        /* 26 à 35 se transpose en ASCII 0..9 */
}

/* flagged(bcp) vérifie si un codet de base a un fanion (majuscule). Le comportement est indéfini si bcp n'est pas un codet de
base. */

#define flagged(bcp) ((punycode_uint)(bcp) - 65 < 26)

/* encode_basic(bcp,flag) force un codet de base à être en minuscules si le fanion est zéro, en majuscules si le fanion est un, et
retourne le codet résultant. Le codet est inchangé si il est sans casse. Le comportement est indéfini si bcp n'est pas un codet de
base. */

static char encode_basic(punycode_uint bcp, int flag)
{
```

```
  bcp -= (bcp - 97 < 26) << 5;
  return bcp + ((!flag && (bcp - 65 < 26)) << 5);
}

/*** Constantes spécifiques de plateforme ***/

/* maxint est la valeur maximum d'une variable punycode_uint : */
static const punycode_uint maxint = -1;
/* Comme maxint est non signé, -1 devient la valeur maximum. */

/*** Fonction d'adaptation de biais ***/

static punycode_uint adapt( punycode_uint delta, punycode_uint numpoints, int firsttime )
{
  punycode_uint k;

  delta = firsttime ? delta / damp : delta >> 1;
                                        /* delta >> 1 est une façon plus rapide de faire le delta / 2 */
  delta += delta / numpoints;

  for (k = 0;  delta > ((base - tmin) * tmax) / 2;  k += base) {
    delta /= base - tmin;
  }

  return k + (base - tmin + 1) * delta / (delta + skew);
}

/*** Fonction principale de codage ***/

enum punycode_status punycode_encode(
  punycode_uint input_length,
  const punycode_uint input[],
  const unsigned char case_flags[],
  punycode_uint *output_length,
  char output[] )
{
  punycode_uint n, delta, h, b, out, max_out, bias, j, m, q, k, t;

/* Initialise l'état : */

  n = initial_n;
  delta = out = 0;
  max_out = *output_length;
  bias = initial_bias;

/* Traite les codets de base : */

  for (j = 0;  j < input_length;  ++j) {
    if (basic(input[j])) {
      if (max_out - out < 2) return punycode_big_output;
      output[out++] =
        case_flags ?  encode_basic(input[j], case_flags[j]) : input[j];
    }
/* autrement si (input[j] < n) retourne punycode_bad_input ; (pas nécessaire pour Punycode avec des codets non signés) */
  }

  h = b = out;

/* h est le nombre de codets traités, b est le nombre de codets de base, et out est le nombre de caractères sortis. */

  if (b > 0) output[out++] = delimiter;
```

```
/* Principale boucle de codage : */

  while (h < input_length) {
                                    /* Tous les codets non de base < n ont déjà été traités. Trouver le prochain plus grand : */

    for (m = maxint, j = 0;  j < input_length;  ++j) {
                                    /* si (basic(input[j])) continuer ; (pas nécessaire pour Punycode) */
      if (input[j] >= n && input[j] < m) m = input[j];
    }

/* Augmenter assez delta pour avancer l'état <n,i> du décodeur à <m,0>, mais attention au débordement : */

    if (m - n > (maxint - delta) / (h + 1)) return punycode_overflow;
    delta += (m - n) * (h + 1);
    n = m;

    for (j = 0;  j < input_length;  ++j) {
                                    /* Punycode n'a pas besoin de vérifier si input[j] est de base: */
      if (input[j] < n /* || basic(input[j]) */ ) {
        if (++delta == 0) return punycode_overflow;
      }

      if (input[j] == n) {
                                    /* Représente delta comme un entier généralisé de longueur variable : */

        for (q = delta, k = base;  ;  k += base) {
          if (out >= max_out) return punycode_big_output;

          t = k <= bias /* + tmin */ ? tmin :              /* +tmin n'est pas nécessaire */
              k >= bias + tmax ? tmax : k - bias;
          if (q < t) break;
          output[out++] = encode_digit(t + (q - t) % (base - t), 0);
          q = (q - t) / (base - t);
        }

        output[out++] = encode_digit(q, case_flags && case_flags[j]);
        bias = adapt(delta, h + 1, h == b);
        delta = 0;
        ++h;
      }
    }

    ++delta, ++n;
  }

  *output_length = out;
  return punycode_success;
}

/*** Fonction principale de décodage ***/

enum punycode_status punycode_decode(
  punycode_uint input_length,
  const char input[],
  punycode_uint *output_length,
  punycode_uint output[],
  unsigned char case_flags[] )
{
  punycode_uint n, out, i, max_out, bias, b, j, in, oldi, w, k, digit, t;

/* Initialiser l'état : */

  n = initial_n;
```

```
  out = i = 0;
  max_out = *output_length;
  bias = initial_bias;
```

/* Traite les codets de base : soit b le nombre de codets en entrée avant le dernier délimiteur, ou 0 si il n't en a pas, alors copier les b premiers codets en sortie. */

```
  for (b = j = 0;  j < input_length;  ++j) if (delim(input[j])) b = j;
  if (b > max_out) return punycode_big_output;

  for (j = 0;  j < b;  ++j) {
    if (case_flags) case_flags[out] = flagged(input[j]);
    if (!basic(input[j])) return punycode_bad_input;
   output[out++] = input[j];
  }
```

/* Boucle principale de décodage : commencer juste après le dernier délimiteur si des codets de base ont été copiés ; autrement, commencer au début. */

```
  for (in = b > 0 ? b + 1 : 0;  in < input_length;  ++out) {
```

/* in est l'indice du prochain caractère à consommer, et out est le nombre de codets dans le dispositif de sortie. */
/* Décoder un entier généralisé de longueur variable en delta, qui sera ajouté à i. La vérification de débordement est plus facile si on augmente i au passage, et qu'on soustrait à la fin sa valeur de début pour obtenir le delta. */

```
    for (oldi = i, w = 1, k = base;  ;  k += base) {
      if (in >= input_length) return punycode_bad_input;
      digit = decode_digit(input[in++]);
      if (digit >= base) return punycode_bad_input;
      if (digit > (maxint - i) / w) return punycode_overflow;
      i += digit * w;
      t = k <= bias /* + tmin */ ? tmin :                    /* +tmin n'est pas nécessaire */
          k >= bias + tmax ? tmax : k - bias;
      if (digit < t) break;
      if (w > maxint / (base - t)) return punycode_overflow;
      w *= (base - t);
    }

    bias = adapt(i - oldi, out + 1, oldi == 0);
```

/* i était supposé revenir à partir de out+1 à 0, incrémentant n à chaque fois, de sorte qu'on va corriger cela maintenant : */

```
    if (i / (out + 1) > maxint - n) return punycode_overflow;
    n += i / (out + 1);
    i %= (out + 1);
```

/* Insérer n à la position i du résultat : pas nécessaire pour Punycode : si (decode_digit(n) ≤ base) retourner punycode_invalid_input ; */

```
    if (out >= max_out) return punycode_big_output;
    if (case_flags) {
      memmove(case_flags + i + 1, case_flags + i, out - i);
                                        /* La casse du dernier caractère détermine le fanion majuscules : */
      case_flags[i] = flagged(input[in - 1]);
    }

    memmove(output + i + 1, output + i, (out - i) * sizeof *output);
    output[i++] = n;
  }

  *output_length = out;
  return punycode_success;
}
```

```
/**************************************************************/
/* Enveloppe d'essai (irait normalement dans un fichier .c séparé) : */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Pour l'essai, on va juste fixer des limites de temps de compilation plutôt que d'utiliser malloc(), et établir une option de temps de compilation plutôt que d'utiliser une option de ligne de commande. */

enum {
  unicode_max_length = 256,
  ace_max_length = 256
};

static void usage(char **argv)
{
  fprintf(stderr,
    "\n"
    "%s -e lit les codets et écrit une chaîne Punycode.\n"
    "%s -d lit une chaîne Punycode et écrit des codets.\n"
    "\n"
"Entrée et résultat sont du texte en clair dans le jeu de caractères natif. Les codets sont dans la forme u+hex séparés par des espaces. Bien que la spécification permette aux chaînes Punycode de contenir tous les caractères du répertoire ASCI, ce code d'essai ne prend en charge que les caractères imprimables, et exige que la chaîne Punycode soit suivie par une nouvelle ligne.\n"
"La casse du u dans u+hex est le fanion force-to-uppercase.\n"
    , argv[0], argv[0]);
  exit(EXIT_FAILURE);
}

static void fail(const char *msg)

{
  fputs(msg,stderr);
  exit(EXIT_FAILURE);
}

static const char too_big[] =
"L'entrée ou le résultat est trop grand, recompiler avec des limites plus grandes.\n";
static const char invalid_input[] = "invalid input\n";
static const char overflow[] = "arithmetic overflow\n";
static const char io_error[] = "I/O error\n";

/* La chaîne suivante est utilisée pour convertir des caractères imprimables d'ASCII en jeu de caractères natif :  */

static const char print_ascii[] =
  "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"
  "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"
  " !\"#$%&'()*+,-./"
  "0123456789:;<=>?"
  "@ABCDEFGHIJKLMNO"
  "PQRSTUVWXYZ[\\]^_"
  "`abcdefghijklmno"
  "pqrstuvwxyz{|}~\n";

int main(int argc, char **argv)
{
  enum punycode_status status;
  int r;
  unsigned int input_length, output_length, j;
```

```
  unsigned char case_flags[unicode_max_length];

  if (argc != 2) usage(argv);
  if (argv[1][0] != '-') usage(argv);
  if (argv[1][2] != 0) usage(argv);

  if (argv[1][1] == 'e') {
    punycode_uint input[unicode_max_length];
    unsigned long codept;
    char output[ace_max_length+1], uplus[3];
    int c;

/* Lire les codets d'entrée : */

    input_length = 0;

    for (;;) {
      r = scanf("%2s%lx", uplus, &codept);
      if (ferror(stdin)) fail(io_error);

      if (r == EOF || r == 0) break;

      if (r != 2 || uplus[1] != '+' || codept > (punycode_uint)-1) {
        fail(invalid_input);
      }

      if (input_length == unicode_max_length) fail(too_big);

      if (uplus[0] == 'u') case_flags[input_length] = 0;
      else if (uplus[0] == 'U') case_flags[input_length] = 1;
      else fail(invalid_input);

      input[input_length++] = codept;
    }

/* Coder : */

    output_length = ace_max_length;
    status = punycode_encode(input_length, input, case_flags, &output_length, output);
    if (status == punycode_bad_input) fail(invalid_input);
    if (status == punycode_big_output) fail(too_big);
    if (status == punycode_overflow) fail(overflow);
    assert(status == punycode_success);

/* Convertit en jeu de caractères natif et résultat : */

    for (j = 0;  j < output_length;  ++j) {
      c = output[j];
      assert(c >= 0 && c <= 127);
      si (print_ascii[c] == 0) fail(invalid_input);
      output[j] = print_ascii[c];
    }

    output[j] = 0;
    r = puts(output);
    if (r == EOF) fail(io_error);
    return EXIT_SUCCESS;
  }

  if (argv[1][1] == 'd') {
    char input[ace_max_length+2], *p, *pp;
    punycode_uint output[unicode_max_length];
```

```
/* Lit la chaîne d'entrée Punycode et la convertit en ASCII : */

   fgets(input, ace_max_length+2, stdin);
   if (ferror(stdin)) fail(io_error);

   if (feof(stdin)) fail(invalid_input);
   input_length = strlen(input) - 1;
   if (input[input_length] != '\n') fail(too_big);
   input[input_length] = 0;

   for (p = input;  *p != 0;  ++p) {
     pp = strchr(print_ascii, *p);
     if (pp == 0) fail(invalid_input);
     *p = pp - print_ascii;
   }

/* Décodage : */

   output_length = unicode_max_length;
   status = punycode_decode(input_length, input, &output_length, output, case_flags);
   if (status == punycode_bad_input) fail(invalid_input);
   if (status == punycode_big_output) fail(too_big);
   if (status == punycode_overflow) fail(overflow);
   assert(status == punycode_success);

/* Sort le résultat : */

   for (j = 0;  j < output_length;  ++j) {
     r = printf("%s+%04lX\n",
               case_flags[j] ? "U" : "u",
               (unsigned long) résultat[j] );
     si (r < 0) fail(io_error);
   }

   return EXIT_SUCCESS;
 }

 usage(argv);
 return EXIT_SUCCESS;                    /* non atteint, mais calme les avertissements du compilateur. */
}
```

Adresse de l'auteur

Adam M. Costello
University of California, Berkeley, USA
http://www.nicemice.net/amc/

Déclaration complète de droits de reproduction

Remerciement
Le financement de la fonction d'édition des RFC est actuellement fourni par l'Internet Society.