Unicode : traiter toutes les écritures du monde

Stéphane Bortzmeyer
AFNIC
<bortzmeyer@nic.fr>

$Id: unicode.db,v 1.15 2003/10/07 08:11:10 bortzmeyer Exp $
Remerciements a :
Patrick Andries - Plein de bonnes suggestions en français : vocabulaire Unicode, notamment
Marianne Roger - Relecture attentive
Sophie-Charlotte Barrière - Relecture attentive

Résumé :

Au contraire d'ASCII et même de Latin-1, Unicode permet de représenter la quasi-totalité des écritures utilisées dans le monde. Une autre particularité d'Unicode est qu'il est conçu selon un modèle en couches, avec séparation du répertoire de caractères et de leur représentation en bits. Cet exposé présentera le jeu de caractères Unicode et les encodages comme UTF-8 (très utilisé dans les protocoles Internet).

1. Textes et informatique : une introduction

Depuis belle lurette, on représente les textes sous forme de 0 et de 1 dans la mémoire des ordinateurs. Cette représentation nécessite une formalisation à laquelle les écritures (et ceux qui les connaissent et les étudient) n'étaient pas toujours préparées. Le domaine est donc assez miné, et le vocabulaire complexe ([1]).

Quelques rappels sont donc nécessaires.

1.1. Langue et écriture

Ce sont deux choses différentes : le turc est passé de l'alphabet arabe à l'alphabet latin au début du XX^{ème} siècle. À la Bibliothèque Nationale, on peut admirer une Bible en espagnol composée avec des caractères arabes. En effet, quant une partie de l'Espagne était occupée par des Arabes, l'espagnol s'écrivait avec l'alphabet des dirigeants.

Certaines langues s'écrivent couramment avec deux alphabets comme le serbo-croate, qui utilise l'alphabet latin en Croatie et l'alphabet cyrillique en Serbie.

On verra qu'Unicode ne s'occupe que des écritures, pas des langues.

De même, à propos des noms de domaines Unicode dans le DNS ([18]), il est inapproprié de parler de "noms de domaines multilingues".

1.2. Le concept de caractère

La norme Unicode utilise ce terme mais il est trompeur : il peut faire penser à des écritures alphabétiques, comme les alphabets grecs ou arabes, alors qu'il existe des écritures non-alphabétiques, notamment en Asie (écriture idéographique en Chine et idéo-syllabique au Japon).

Certains préfèrent donc éviter complètement le terme de caractère.

1.3. La variété du monde

Chaque fois que l'on croit avoir tout normalisé, on découvre une écriture qui comporte des exceptions aux règles que l'on croyait complètes.

Pensons aux écritures de droite à gauche, aux écritures en lacet (un coup à gauche, un coup à droite), aux écritures à casses différentes (majuscules/minuscules, qui sont très déroutantes pour ceux qui ont appris avec un alphabet qui n'a pas ce concept), etc.

1.4. La forme

On le verra, Unicode gère des caractères abstraits, pas des formes (glyphes), en raison de la grande variété de formes pour une même lettre (et les concepteurs de polices de caractères en inventent régulièrement des nouvelles).

La norme Unicode présente des glyphes, mais à des fins d'illustration uniquement. Unicode n'est **pas** une police de caractères.

2. Un modèle pour la représentation de caractères

Unicode est défini suivant un modèle en couches ([4]). Les autres normes ne faisaient typiquement pas de distinction entre, par exemple, le jeu de caractères et la représentation physique. On va présenter les couches, en partant de la plus haute (la plus éloignée de la machine).

2.1. Jeu de caractères abstraits (*Abstract Character Repertoire*)

La couche la plus élevée est la définition du jeu de caractères. Par exemple, Latin-1 a un jeu de 256 caractères et Unicode normalise actuellement près de 100 000 caractères. En outre, Unicode leur donne des noms[1].

La table suivante est donc le cœur d'Unicode.

Tableau 1. Une partie d'Unicode

Lettre majuscule latine a	A
Lettre majuscule latine c cédille	Ç
Lettre minuscule grecque lambda	λ
Lettre hébraïque alef	ℵ

1. Les noms utilisés ici sont les noms officiels français extraits de l'ISO 10646 (F) (http://pages.infinit.net/hapax/ListeDesNoms.txt).

2.2. Jeu de caractères codés (*Coded Character Set*)

Ici, on ajoute à la table précédente un index numérique. Notons bien qu'il ne s'agit pas d'une représentation en mémoire, juste d'un nombre.

Tableau 2. Une partie d'Unicode

U+0041	Lettre majuscule latine a	A
U+00C7	Lettre majuscule latine c cédille	Ç
U+03BB	Lettre minuscule grecque lambda	λ
U+0500	Lettre hébraïque alef	ℵ

En Unicode, cet index numérique se nomme le point de code (*code point*). Il est noté U+xxxx où xxxx est la valeur de l'index en hexadécimal. Répétons que c'est juste un index, pas une représentation en bits.

2.3. Forme codée en mémoire (*Character Encoding Form*) ou encodage

Cette fois, nous arrivons à une représentation en mémoire : cette couche spécifie quelles unités de stockage (*code units*), octets ou bien mots de 16 - seizets - ou de 32 bits, vont représenter un caractère ou plus exactement un point de code (*code point*).

Une des particularités d'Unicode est qu'il existe plusieurs encodages (*Character Encoding Form*) possibles (exemples plus loin). Comme ce modèle en couches est inhabituel pour les jeux de caractères, il y a fréquemment confusion entre Unicode et tel ou tel encodage. Par exemple, le paramètre "charset" utilisé dans MIME est mal nommé puisque ses valeurs sont des encodages, pas des jeux de caractères.

Du fait de ce modèle en couches successives, des phrases comme "Unicode est un jeu de caractères sur 16 bits" ne sont pas seulement fausses, elles sont absurdes.

2.4. Mécanisme de sérialisation des caractères (*Character Encoding Scheme*)

Cette couche s'occupe de sérialiser les unités de stockage (*code units*) définies par la couche du dessus. C'est ici que se traite l'opposition entre gros boutiens et petits boutiens[2]. Ce mécanisme définit donc ce que l'IETF (http://www.ietf.org/) appelle un *on-the-wire format* (format des données vues par un observateur qui regarde ce qui passe sur le câble).

C'est également ici qu'on spécifie la marque de boutianité (BOM, pour *Byte Order Mark*) qui permet d'indiquer en début de fichier s'il est en gros boutien ou en petit boutien. Dans le monde Internet, on l'utilise rarement, en préférant un marquage explicite (charset=UTF-16BE en MIME, par exemple, pour indiquer un flot de données gros boutien, avec BE pour *Big Endian*).

2.5. Surcodage de transfert (*Transfer Encoding Syntax*)

Ici, interviennent **optionnellement** les mécanismes de compression ou de chiffrement, que nous n'étudierons pas.

Il peut aussi y avoir un surencodage comme pour le LDAP qui spécifie que les chaînes Unicode doivent être encodées en UTF-8 et surencodées en Base-64.

2. Les gros boutiens sont les processeurs - la grande majorité sauf les i386 - qui mettent l'octet de plus fort poids en premier. Quant un flot de données multi-octets passe d'un processeur gros boutien à un petit boutien via le réseau, il faut penser à inverser les octets.

3. Les différents jeux

3.1. US-ASCII

Il ne comprend que l'alphabet latin, et sans caractères composés (127 caractères en tout).

Les caractères sont encodés sur 7 bits (en général inclus dans un octet dont le bit de poids fort est à zéro).

Pour écrire, il ne convient donc qu'aux anglophones[3] et aux communications entre ordinateurs (VRFY, HELO, POST, GET, etc).

3.2. ASCII accentué[4]

Les lecteurs les plus âgés se souviennent des imprimantes 7 bits : incapables de recevoir les octets dont le bit de poids fort était à un, elles redéfinissaient US-ASCII pour que les motifs correspondant à des caractères comme { ou } impriment les caractères composés utilisés en français.

C'était supportable pour les textes mais peu pratique pour les programmeurs C ou Perl. Rapidement, tous les systèmes sont devenus *8-bits clean* et ce jeu de caractères a donc disparu.

3.3. ISO-8859-X

Ce terme désigne un ensemble de normes ISO[5]. La plus connue en France est ISO-8859-1, dite Latin-1 (**man iso_8859_1**). Le répertoire comprend 255 caractères. L'encodage est sur 8 bits (un octet). C'est pour remplacer US-ASCII et "ASCII accentué" par ISO-8859-X qu'il a fallu se battre pour déployer des systèmes *8-bits clean* dans les années 80-90.

Contrairement à une idée reçue, Latin-1 ne permet pas d'encoder tous les caractères utilisés en français. La phrase suivante (tirée de [6]), qui inclut tous ces caractères, utilise des caractères extérieurs à Latin-1 (la ligature œ n'existe pas du tout en Latin-1 et le ÿ n'y existe pas en majuscule).

Dès Noël où un zéphyr haï me vêt de glaçons würmiens, je dîne d'exquis rôtis de bœuf au kir à l'aÿ d'âge mûr & cætera !

Latin-1 sera petit à petit remplacé par Latin-15, quasi-identique à l'exception du caractère € pour l'euro.

Pour l'écriture grecque, on utilise ISO-8859-7. La partie haute (bit de poids fort à 1) sert pour l'alphabet grec et la partie basse pour l'alphabet latin. Cela permet de programmer en C en mettant des commentaires en grec. Ou bien d'écrire un texte qui mêle le grec et l'anglais (mais pas le grec et l'allemand).

3.4. Pourquoi Unicode ?

Alors, après tant de jeux de caractères, largement implémentés et déployés, pourquoi Unicode ? Parce qu'il est le seul à permettre des textes multi-écritures (français et grec, par exemple) et le seul qui permet de traiter toutes les écritures (certaines écritures "rares" n'ont pas de jeu de caractères en dehors d'Unicode).

Voilà pourquoi je souhaite le succès d'Unicode.

---

3. Et encore puisque certains mots anglais en comportent comme *résumé* (qu'il ne faut pas confondre avec *resume*).
4. **ISO 646 (http://alis.isoc.org/glossaire/iso646.10646.fr.htm)**
5. Un serveur Web (http://babel.alis.com/codage/iso8859/jeuxiso.fr.htm) les présentent fort bien.

4. Unicode au complet

4.1. Les index

Les 128 premiers caractères d'Unicode ont un index qui correspond aux caractères d'US-ASCII, pour des raisons pratiques évidentes[6]. Ainsi, le petit "d" a pour *code point* U+0064[7], comme en US-ASCII.

Les 128 suivants ont le même index qu'en Latin-1, par exemple U+OOFE pour le þ, le *thorn* scandinave.

Rappelons qu'il ne s'agit que d'une identité des index, pas des représentations, un fichier Latin-1 ne sera pas un fichier Unicode légal, aucun encodage d'Unicode ne correspond à Latin-1.

4.2. Les combinaisons

Unicode gère les combinaisons de caractères. Certains caractères peuvent en effet être définis comme la combinaison de deux ou plus caractères. Par exemple le ç (c cédille) peut être vu comme la combinaison de la lettre c et de la cédille combinatoire.

C'est une des raisons pour lesquelles il n'y a pas d'équivalence entre les caractères au sens Unicode du terme et les caractères perçus par l'utilisateur. Si un fichier Unicode contient les deux points de code U+0063 (c) et U+0327 (cédille combinatoire[8]), l'utilisateur qui l'affiche ne verra probablement qu'un seul caractère, ç.

En outre, à des fins de compatibilité avec les anciens jeux de caractères, certaines combinaisons sont précomposées dans Unicode. La combinaison citée au paragraphe précédent aurait pu être remplacée par le seul U+00E7 (ç).

L'existence de caractères combinants est une des raisons pour lesquelles il ne faut **jamais** comparer deux chaines de caractères Unicode sans leur avoir fait subir une normalisation.

4.3. Les encodages

Cas unique parmi les jeux de caractères, Unicode permet plusieurs encodages. La conception d'un bon encodage n'est pas triviale et chaque encodage a des propriétés distinctes :

- Place occupée en mémoire. Ce point est en général celui qui suscite les plus grandes passions. Bien à tort, à l'époque où, sur une page Web, le texte, même encodé avec l'encodage le plus gaspilleur, ne représente qu'une petite partie de la taille totale, vue l'abondance des images fixes, de la vidéo, etc.
- Compatibilité avec les anciennes applications. Par exemple, la libc s'attend à ce que les chaines soient terminées par un octet nul. Un encodage qui produirait des octets nuls lui poserait des problèmes.
- Performances pour des opérations comme la sélection du N^{ième} caractère. Un encodage où tous les caractères sont représentés avec la même taille est avantagé.

4.3.1. UCS-2

Cet encodage tient sur 16 bits et ne permet donc pas de représenter tout Unicode. Les 65 535 premiers caractères d'Unicode sont nommés le plan multilingue de base (BMP, *Basic Multilingual Plane*) et comportent presque toutes les écritures

6. Contrairement à beaucoup d'autres normes, Unicode a beaucoup soigné le côté pratique.
7. Ceci est la notation habituelle des points de code, en hexadécimal.
8. Il existe aussi une cédille non-combinatoire, U+00B8.

couramment utilisées, seuls les idéogrammes chinois manquant largement. UCS-2 n'est guère utilisé, en raison de cette limite.

4.3.2. UCS-4 ou UTF-32

Cet encodage tient sur 32 bits et est le seul où l'index est identique à la représentation. Si Unicode dépasse un jour les quatre milliards de caractères[9], il aura les mêmes problèmes qu'UCS-2, mais cela semble peu vraisemblable. UTF-32/UCS-4 n'est guère utilisé mais, lorsque vous compilez Python (http://www.python.org/), l'option **--enable-unicode=ucs4** indique à Python qu'il devra utiliser UCS-4 en interne pour les chaines Unicode.

4.3.3. UTF-16

Comme UCS-2, c'est un encodage 16-bits. Mais il dispose d'un mécanisme d'échappement, les seizets d'indirection (*surrogates*) qui lui permet de représenter tout Unicode. Java utilise cet encodage en interne[10].

4.3.4. UTF-8, l'encodage de l'Internet ?

UTF-8 ([16]) est peut-être le plus répandu. En effet, il est l'encodage par défaut en XML (tout processeur XML doit traiter au moins UTF-8 et UTF-16) et les normes Internet qui utilisent Unicode imposent la plupart du temps UTF-8.

UTF-8 est le plus compatible avec les anciennes applications comme celles qui utilisent la libc. Il ne produit pas d'octets nuls, un caractère ASCII a la même représentation en UTF-8 et en ASCII (ce qui veut dire que tout fichier ASCII est un fichier UTF-8), les octets d'ASCII seront forcément une représentation d'un caractère ASCII (ce qui permet d'utiliser un grep non modifié sur un fichier UTF-8, tant qu'on ne cherche que des caractères US-ASCII).

Par exemple, on peut utiliser UTF-8 avec les noyaux Unix existants, si on veut des noms de fichier en Unicode[11].

En revanche, comme un caractère Unicode a un nombre d'octets variable en UTF-8, des opérations comme l'extraction du $N^{ième}$ caractère d'une chaîne sont inefficaces.

Perl utilise UTF-8 comme représentation interne des chaines Unicode.

4.3.5. Trouver l'encodage

Tout ceci laisse entière la question de trouver l'encodage d'un fichier que l'on vient de recevoir ou bien d'un flot de bits qui vient du réseau. Il existe certaines heuristiques (UTF-8 a une syntaxe telle que, si un ensemble d'octets est de l'UTF-8 valable, il est très probablement réellement de l'UTF-8, le hasard seul ne peut pas produire de l'UTF-8 légal) mais aucune solution générale. Il faut donc marquer ses fichiers. Dans le courrier, on utilise l'option "charset" du champ Content-Type de MIME, pour le Web, une option équivalente peut être utilisée en HTTP, pour les fichiers en local, ni Unix, ni MS-Windows ne prévoient de méthode. Lorsqu'on décide de passer à Unicode, il vaut mieux changer tous ses fichiers d'un coup.

---

9. En fait, beaucoup moins, car un autre encodage, UTF-16, impose d'autres contraintes.
10. Il semble que Java ne le mette pas en œuvre correctement et qu'il s'agisse plutôt d'UCS-2 (http://www.ingrid.org/java/i18n/utf-16/) (sans les seizets d'indirection). Un tel problème existe avec d'autres systèmes : l'UTF-8 d'Oracle n'est pas complètement correct, par exemple.
11. En pratique, le noyau ne faisant pas de normalisation, on aura sans doute des ennuis.

5. Diverses questions

5.1. Normalisation

Si on compare deux chaînes de caractère Unicode en comparant leurs octets, on est sûr de se tromper. En effet, elles n'utilisent pas forcément le même encodage et la comparaison physique des bits n'a donc pas de sens.

Or, les applications ont tout le temps besoin de comparer des textes, pour vérifier l'unicité (noms de domaines, noms de fichiers) ou pour des signatures cryptographiques, par exemple.

Si on est d'avantage soigneux et qu'on convertit tout en entrée, de façon à n'utiliser qu'un seul encodage en interne, il reste d'autres problèmes : les caractères combinants et le fait que certains caractères, quoique différents pour la norme Unicode (*code point* différents) sont équivalents dans un contexte donné. Ainsi, la plupart des germanophones considèrent que *straße* et *strasse* sont équivalents. Et "pi au carré" ? Est-ce que "pi2" est équivalent à "pi^2" ? Le second peut être mieux représenté avec un balisage extérieur (en LaTeX ou en DocBook). Enfin, les ligatures comme fi ou ff, peuvent être trompeuses.

Le problème est encore plus épineux en chinois où il existe plusieurs représentations du même caractère, depuis une réforme de la graphie.

Si on veut comparer deux chaînes de caractères Unicode, par exemple pour mettre en œuvre un système de fichiers où les noms Unicode sont admis, il faut donc normaliser les chaînes avant comparaison.

Unicode prévoit plusieurs mécanismes de normalisation, que l'on peut utiliser pour bâtir sa normalisation à soi (le cadre de normalisation d'Unicode, pour l'Internet, est défini dans [17][12]).

Pour plus de détails, on consultera [3].

Pour voir la normalisation en action, on peut utiliser l'excellent programme charlint (http://www.w3.org/International/charlint/) du World-Wide Web Consortium. Par exemple, on a ci-dessous un programme Perl qui produit des données sous une forme non canonique et un autre programme Perl qui affiche simplement les caractères Unicode reçus :

```
% perl produce-data.pl |  perl consume-data.pl

LATIN SMALL LETTER C (Basic Latin)
COMBINING CEDILLA (Combining Diacritical Marks)
LATIN SMALL LETTER E (Basic Latin)
COMBINING ACUTE ACCENT (Combining Diacritical Marks)
```

Et, en insérant charlint dans le tube :

```
% perl produce-data.pl | perl charlint.pl | perl consume-data.pl

LATIN SMALL LETTER C WITH CEDILLA (Latin-1 Supplement)
LATIN SMALL LETTER E WITH ACUTE (Latin-1 Supplement)
```

Les mots ayant un contenu sémantique (même les identificateurs dans les programmes), la normalisation peut susciter des débats vigoureux.

Quiz. Par exemple, pour les noms de domaines, considérez-vous les noms suivants comme équivalents ?

1. café.fr

12. Un exemple d'utilisation figure en [19].

2. CAFÉ.FR
3. cafe.fr
4. cafê.fr (notez la faute)

5.2. Casse

Mettre en œuvre un système de nommage indépendant de la casse est trivial en ASCII : on ajoute ou on retire 32 à l'encodage. Cela ne marche pas en Unicode, où les tables sont bien plus complexes. Même la longueur des chaînes change selon la casse (le majuscule de *weiß* est *WEISS*).

Le consortium Unicode fournit une table de correspondance de casse (*case folding*) qu'on peut utiliser dans ses programmes. Il ne peut pas exister une table qui convienne à tous car, dans certains cas, le changement de casse peut avoir un résultat qui dépend du profil (*locale*), par exemple de la langue de l'utilisateur.

5.3. Tri

Le tri pose des problèmes analogues à la normalisation. En Unicode, on ne peut pas, contrairement à US-ASCII, trier sur la représentation binaire : il faut introduire plus de sémantique et consulter des tables.

Par exemple, en Suède et en Finlande, le Ä apparait après le Z dans les tris, comme l'annuaire du téléphone, ce qui déroute l'étranger de passage qui cherche son correspondant...

5.4. Outils Unicode[13]

- yudit (http://www.yudit.org/), un éditeur Unicode
- Depuis la version 21, Emacs commence à avoir une prise en charge correcte d'Unicode
- charlint (http://www.w3.org/International/Charlint/), un outil de normalisation
- recode (http://www.gnu.org/directory/GNU/recode.html), un convertisseur entre jeux de caractères (**recode `Latin-1..UTF-8 mon-fichier.txt`**)
- iconv, un autre convertisseur, dans la GNU libc

5.5. Prise en charge d'Unicode dans les programmes

Que veut dire "supporter Unicode" ? Unicode étant désormais un argument de vente, on voit de nombreux programmes proclamer qu'ils "supportent Unicode". Or, il existe plusieurs niveaux de prise en charge :

1. Au minimum, un programme doit pouvoir réaliser des entrées/sorties en Unicode. En entrée, il devrait accepter au moins les encodages UTF-8 et UTF-16, en sortie également. Entre les deux, il ne doit **pas** modifier les caractères qu'il ne comprend pas (aucun programme ne peut connaitre la sémantique de tous les caractères Unicode).
2. À un niveau de support plus élevé, il devrait pouvoir :
   a. S'il fait des tris, ne pas trier selon la représentation binaire, mais selon l'algorithme de tri d'Unicode.
   b. S'il permet des recherches avec des expressions rationnelles, utiliser les expressions Unicode.

13. Cet article a été réalisé avec Emacs en DocBook/XML sur une machine Debian. Il a été traité avec jade, JadeTeX et xsltproc.

5.6. Support d’Unicode dans vos programmes

Unicode nous oblige à reconsidérer pas mal de présupposés. Par exemple, une fonction aussi inoffensive que `strlen` doit renvoyer quoi ?

1. Le nombre de graphèmes (ce que l’utilisateur appelerait "caractères") ?
2. Le nombre de *code points* (ce que l’informaticien appelerait "caractères") ?
3. Le nombre d’octets (ce que ferait un programme naïf[14]) ?

Ainsi, la chaîne "U+0063 U+0327" représente un seul graphème (ç), mais nécessite deux points de code (si on normalisait, on pourrait la réduire à un seul) et un nombre d’octets qui dépend de l’encodage.

La distinction entre ces différents sens (qui sont tous valables, à leur façon), se retrouve dans la plupart des langages de programmation, qui permettent de différencier entre eux. Voici un exemple en Perl :

```
# "character semantics" vs. "byte semantics" en Perl
# Dans le premier cas, on travaille au niveau du jeu de caractères
# Unicode.
# Dans le deuxième, on ne connait que des octets, sans leur signification.
print $ustring->length(), "\n"; # Affiche le
                  # nombre de code points

$utf8 = $ustring->utf8;
print length($utf8), "\n"; # Affiche le nombre
                  # d’octets
```

5.7. Langages de programmation

5.7.1. Règles générales

Chaque langage utilise un encodage différent en interne : attention aux entrées/sorties binaires !

Unicode soulève des problèmes de sécurité nouveaux :

1. Vu le nombre de caractères présents, tester la présence de caractères interdits a encore moins de sens qu’en US-ASCII. Il faut au contraire tester que les données ne comportent que des caractères autorisés.
2. Si vous validez des données, faites-le **après** normalisation.
3. Il y a peu de chances que vous écriviez un décodeur UTF-8 mais, si c’est le cas, attention, il existe plusieurs pièges permettant potentiellement de faire passer des caractères illégaux.
4. On pourra consulter *Security Risks of Unicode* (http://www.counterpane.com/crypto-gram-0007.html#9), bien qu’il soit excessivement hostile, et la discussion qui a suivi (http://www.counterpane.com/crypto-gram-0008.html#9).

14. "Naïf" au sens de "écrit par un programmeur qui ne connait pas Unicode et qui est resté au paradigme un caractère == un octet".

5.7.2. Perl

Perl a un bon support d'Unicode. La version 5.8 ou supérieure est recommandée pour Unicode. **man perlunicode** sur un système où Perl est installé vous donnera plein d'informations. Une bonne liste de diffusion est perl-unicode@perl.org.

Le classique livre de Christiansen et Torkington, "Perl Cookbook", aurait été refait dans sa deuxième édition pour exposer toutes les arcanes de la prise en charge d'Unicode par Perl.

5.7.3. Python

Python a un bon support d'Unicode, qui s'est encore amélioré dans la version 2.3. [9] est un bon point de départ.

5.7.4. Java

Java n'utilise qu'Unicode depuis le début.

5.7.5. C

Pas de vrai support Unicode. Il existe des types de données suffisamment grands pour stocker les caractères Unicode comme wchar_t mais sans sémantique associée. L'entrée *Character Set Handling* de la documentation de la GNU libc contient des indications à ce sujet.

6. Études de cas

6.1. Noms de fichiers en Unicode

Pour une première étude de cas, imaginons qu'on veuille permettre à un système Unix d'avoir des noms de fichier en Unicode, comme le fait Plan9. On suppose que le noyau est *8-bits clean*[15]. Le problème peut sembler simple mais il ne l'est pas. Il faut en effet définir des règles de normalisation pour éviter d'avoir deux fichiers ayant le même nom affiché mais deux chaînes de points de code différentes. Et il va falloir Unicodiser certaines applications.

Pour normaliser, on peut s'appuyer sur stringprep et définir un profil adapté aux noms de fichier. Le noyau sera alors chargé de mettre les noms de fichier sous leur forme canonique[16].

On va ensuite Unicodiser les applications. Prenons l'exemple de ls. Cette application va devoir afficher des noms en Unicode. Il ne faut évidemment pas mettre un moteur de rendu Unicode complet dans chaque application : il faudra développer une bibliothèque que ls pourra appeler.

ls doit également trier les fichiers par leur nom : il faudra donc ajouter à ls l'algorithme de tri d'Unicode.

---

15. Un encodage d'Unicode sur 7 bits, UTF-7, permet de gérer l'Unicode sur des systèmes archaïques mais il est très peu utilisé.
16. Il est clair qu'un tel *patch*, avec la taille des tables Unicode et les problèmes de sécurité potentiels, ne va pas susciter l'enthousiasme de beaucoup de développeurs Linux ou NetBSD...

6.2. Noms de domaine en Unicode (IDN)

Notre autre étude de cas portera sur un exemple réel. On le sait, un nom comme `www.maçonnerie-générale.fr` ne serait pas légal. Ce qu'on sait moins, c'est que le problème ne vient pas entièrement du DNS, qui est *8-bits clean* depuis toujours mais des règles de nommage des machines, beaucoup plus strictes, et qui n'acceptent que le US-ASCII[17].

Les noms de domaines internationalisés (IDN), visent à permettre l'utilisation de noms en Unicode. IDN permet donc des noms multi-écritures[18].

Pour cela, la norme, [18] spécifie :

1. Un principe : le DNS lui-même n'est pas modifié. Aucun besoin de changer les serveurs DNS ou les couches basses des résolveurs. Tout le travail doit être fait dans les applications, d'où le nom officiel de cette solution, IDNA (*Internationalized Domain Names in Applications*).
2. Une normalisation, dite nameprep, décrite en [19], lui-même un profil de stringprep, décrit en [17].

   Cette normalisation, commune à tous, permet de comparer des noms de domaine afin de savoir si `pi2.fr` est égal à `pi`^{2}`.fr`. Toute comparaison d'IDN doit donc se faire après le passage par nameprep, qui transforme un IDN en un autre IDN, opération qui est irréversible puisque plusieurs IDN ont la même forme normalisée.

3. Un encodage, qui est spécifique à IDN [19], punycode. Après le passage par punycode, l'IDN est exprimé en US-ASCII et peut passer par les bibliothèques existantes. Cet encodage est réversible.

Le déroulement complet de l'algorithme est le suivant (cet algorithme est effectué par l'application, ou plutôt par la bibliothèque IDN utilisée, pas par l'utilisateur).

1. Vous partez d'une chaîne Unicode, disons CAFÉ.fr.
2. Vous la découpez en labels, ici CAFÉ et fr.
3. Vous passez les labels dans nameprep, ce qui donne ici café et fr[20].
4. Vous passez les labels dans punycode[21], ce qui donne ici xn--caf-dma et fr[22]. Ce sont ces labels *punycodés* qui seront mis dans le fichier de zone du serveur DNS.

Un algorithme analogue permet de traduire l'ACE (*ASCII Compatible Encoding*) en un IDN Unicode.

Une application IDN, ici ping peut alors fonctionner. ping étant un outil de déboguage, on a laissé voir l'encodage punycode, l'ACE, ce que ne ferait pas une application normale.

```
% ping CAFÉ.eureg.org
PING CAFÉ.eureg.org (xn--caf-dma.eureg.org) (192.134.7.252) from 192.134.4.68 : ...
64 bytes from café.eureg.org: icmp_seq=1 ttl=254 time=0.775 ms
64 bytes from café.eureg.org: icmp_seq=2 ttl=254 time=0.359 ms
64 bytes from café.eureg.org: icmp_seq=3 ttl=254 time=0.342 ms

--- CAFÉ.eureg.org (xn--caf-dma.eureg.org.) ping statistics ---
3 packets transmitted, 3 received, 0% loss, time 2021ms
rtt min/avg/max/mdev = 0.342/0.492/0.775/0.200 ms
```

17. Plus exactement dans un sous-ensemble d'US-ASCII, LDH, qui ne comporte que les lettres, les chiffres et le tiret.
18. Et pas multi-lingues, comme on le lit souvent.
19. IDN est donc une des rares normes IETF à ne pas utiliser UTF-8.
20. En alphabet latin, nameprep se contente en général de tout mettre en minuscules mais dans d'autres écritures, sa tâche est plus complexe.
21. Le choix du préfixe xn--, qui permet de reconnaitre un IDN encodé, sans ambiguïté, n'a pas été sans mal...
22. Les labels déjà en US-ASCII ne sont pas affectés.

Il existe déjà plusieurs applications IDN comme Mozilla ou mutt. De nombreuses bibliothèques en logiciel libre (comme GNU libidn (http://josefsson.org/libidn/)) existent pour les programmeurs qui veulent IDNiser leurs applications.

Plusieurs registres ont commencé à enregistrer des IDN. Les délais pour déployer cette technologie ne sont pas tous techniques mais viennent aussi de la difficulté d'adapter les règles de droit au nom ou bien les règles d'arbitrage en cas de conflit.

7. Questions non techniques

7.1. Les organismes

Le consortium Unicode (unicode) est le principal organisme pour le développement d'Unicode. Il édite la norme, les rapports techniques (en général, très lisibles, comme les RFC et contrairement aux normes ITU ou ISO), les fichiers de données.

La licence est assez restrictive mais au moins tout est disponible en ligne.

Il existe également un comité ISO, le 10646, qui ne semble pas avoir d'autres activités que de mettre un tampon ISO sur les documents Unicode.[23]

7.2. Déploiement d'Unicode

Comme pour d'autres technologies, la seule supériorité technique ne suffira pas à faire passer tout le monde à Unicode dans la journée. Le processus de transition promet de ne pas être instantané.

Plusieurs acteurs vont en effet devoir agir :

- Les auteurs de norme vont devoir intégrer Unicode : pour les normes IETF ([15] et [21]) ou W3C ([11]), c'est déjà largement réalisé.
- Les programmeurs vont devoir écrire des programmes Unicode. C'est beaucoup plus simple dans un langage de haut niveau comme Python mais cela va nécessiter des changements.

  Il faudra aussi se débarasser d'habitudes mentales très ancrées comme l'égalité un caractère = un octet.

- Les administrateurs système vont devoir donner la préférence aux programmes qui prennent déjà en charge Unicode. Si Unicode est une option de compilation, ils vont devoir penser à l'activer.
- Et les utilisateurs vont devoir houspiller tout le monde pour avoir Unicode :-)

Une transition très longue est donc à prévoir, pendant laquelle il faudra accepter la coexistence d'Unicode avec d'autres jeux de caractères.

23. Autrefois, Unicode et UCS - le nom officiel de la norme ISO - étaient deux normes différentes. Ce sont aujourd'hui la même.

Bibliographie

[1] Unicode consortium, *Unicode Glossary*, ?.

http://www.unicode.org/glossary/

[2] Unicode consortium, *UTR #10 - Unicode Collation Algorithm*, 2002.

http://www.unicode.org/reports/tr10/

[3] Unicode consortium, *UTR #15 - Unicode Normalization Forms*, 2003.

http://www.unicode.org/reports/tr15/

[4] Unicode consortium, *UTR #17 - Character encoding model*, 2000.

http://www.unicode.org/reports/tr17/

[5] Unicode consortium, *UTR #18 - Unicode Regular Expression Guidelines*, 2002.

http://www.unicode.org/reports/tr18/

[6] Jacques André, *ISO Latin-1, norme de codage des caractères européens?*, 1996.

http://www.gutenberg.eu.org/pub/GUTenberg/publicationsPDF/25-andre.pdf

[7] Rob Pike, Ken Thompson, *Hello World*, , 1993.

http://www.cs.bell-labs.com/sys/doc/utf.html

[8] Richard Gillam, *Unicode demystified*, 2003, Addison-Wesley.

[9] Marc-André Lemburg, *Python and Unicode*, 2002.

http://www.egenix.com/files/python/Unicode-EPC2002-Talk.pdf

[10] Hapax, *Unicode 3.1 et ISO 10646 en français*, ?.

http://pages.infinit.net/hapax/

[11] World-Wide Web consortium, *Character Model for the World Wide Web*, 2003.

http://www.w3.org/TR/charmod/

[12] P. Mockapetris, *RFC 1035: Domain names - implementation and specification*, 1987.

[13] R. Braden, *RFC 1123: Requirements for Internet Hosts - Application and Support*, 1989.

[14] N. Freed, N.S. Borenstein, *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, 1996.

[15] H.T. Alvestrand, *RFC 2277: IETF Policy on Character Sets and Languages*, 1998.

[16] F. Yergeau, *RFC 2279: UTF-8, a transformation format of ISO 10646*, 1998.

[17] P. Hoffman, M. Blanchet, *RFC 3454: Preparation of Internationalized Strings ("stringprep")*, 2002.

[18] P. Faltstrom, P. Hoffman, A. Costello, *RFC 3490: Internationalizing Domain Names in Applications (IDNA)*, 2003.

[19] P. Hoffman, M. Blanchet, *RFC 3491: Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)*, 2003.

[20] A. Costello, *RFC 3492: Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)*, 2003.

[21] P. Hoffman, *RFC 3536: Terminology Used in Internationalization in the IETF*, 2003.