Rapport de troisième Soutenance

SD\n – ALIAS

*<**A** **L**exeme **I**s **A** **S**ound>*

Julie THÉODOSE-SOREL Mathieu BASCLE
Nicolas TANDÉ Yohan BAINIER

11 mai 2005

Table des matières

1 Introduction

Section Disparue à la ligne au rapport pour cette troisième soutenance, aKa S-III.
L'objectif de ce rapport est de vous présenter l'avancée de notre travail avant la terrible soutenance finale.

La mission à accomplir pour cette soutenance était la suivante : améliorer le *Parser* pour qu'il puisse détecter les nombres pour ensuite les traduire en suites de phonèmes, tâche réalisée par l'Officier BAINIER et la recrue BASCLE. Il s'agissait surtout d'améliorer encore et toujours le *Speaker* pour pouvoir enfin restituer les sons associés aux phonèmes. Cette tâche ardue a été confiée aux recrues THÉODOSE SOREL et TANDÉ. Il est à noter que notre projet ne contient plus qu'un seul fichier exécutable et que toutes ses fonctionnalités finales sont disponibles à partir de l'*Interface*.

Cette mission a été la plus rude de toutes celles auxquelles nous avons participé. En effet, il a fallut faire face aux terribles *Contrôle de mathématiques* et *Contrôles d'algorithmique*, sans oublier l'ultime ennemi auquel nous avons du faire face, les *Vacances De Pâques*. Heureusement, l'équipe n'a pas faillit à sa tache, et c'est ainsi que nous pouvons présenter devant vos yeux ébahis, **ALIAS**, le *Text To Speech* que tout le monde s'arrache !

2 Autotools

Lors de la préparation des précédentes soutenances, nous avions un problème majeur dû à la portabilité des codes sources : certaines parties du code étaient incompatibles entre PC et PowerPC. Ainsi, nous avions des conflits de version sur *CVS* car certaines sources demandent l'usage de fonctions spécifiques à **Linux** (notamment *ALSA*[1] dans la partie audio du projet), fonctions non implémentées sous **MacOS X**.
Nous avons donc procédé à l'amélioration des fichiers *configure.in* et *Makefile.am* en prévention du portage sur les deux plateformes choisies.

2.1 configure.in

L'amélioration apportée à ce fichier réside dans le choix des options possibles à passer au script *configure* et surtout, au fait qu'il créée des constantes très pratique pour la compilation conditionnelle.

Il est maintenant possible de passer un chemin préfixe où chercher *ALSA*, *Ogg* et *Vorbis*. La macro `AC_ARG_WITH` permet à *configure* de prendre en compte ce genre d'option :

```
$ ./configure --with-ogg-prefix=/sw
```

Dans ce cas, la variable `$withval`, créée par `AC_ARG_WITH` contiendra "/sw". Il est alors possible d'effectuer des tests de détections de la bibliothèque *Ogg* à partir de ce préfixe... Lorsqu'une bibliothèque est trouvée, une constante est définie dans le fichier *config.h* mais aussi de manière globale afin que sa portée atteigne aussi les *Makefile*'s. La macro `AC_DEFINE` permet de définir une constante dans *config.h*, et `AM_CONDITIONAL` permet de définir une constante (booléenne) pour les *Makefile.am*'s. Par exemple, si *ALSA* est détectée, on définit `HAVE_ALSA` via `AC_DEFINE` mais aussi par `AM_CONDITIONAL`.

[1] Advanced Linux Architecture Sound

2.2 Makefile.am

Voici tout d'abord un exemple pour illustrer l'explication précédente :

```
if HAVE_ALSA
libaudio_la_SOURCES       += alsa_support.c
LIBS                      += @ALSA_LIBS@
CFLAGS                    += @ALSA_CFLAGS@
endif
```

Cet exemple est un extrait du *Makefile.am* qui sert à compiler le module audio de notre projet. On remarque que si *ALSA* est présente, un fichier source sera rajouté à la bibliothèque partagée (générée par *libtool*), et les macros `LIBS` et `CFLAGS` du compilateur seront modifiées en conséquences.

2.3 Code source portable

Comme nous venons de le voir, certaines constantes ont été définies ou non lors du *configure*. De ce fait, nous sommes maintenant capables de supprimer ou d'ajouter du code selon l'architecture ou les bibliothèques présentes.
`__APPLE__` nous permet de savoir si nous sommes sur un **MacOS**. Voici un autre exmple extrait de *decode.c* :

```
#ifdef HAVE_ALSA
# include <alsa/asoundlib.h>
# include "alsa_support.h"
#endif
```

Ceci permet d'inclure les fichiers d'en-têtes nécessaires au support *ALSA* seulement si la bibliothèque a été détectée avant la compilation.

Les *Autotools* constituent un outils formidable et très pratique pour le portage de code mais ils restent néanmoins relativement difficiles à utiliser.

3 Interface

Je sais, il avait été dit et redit, que l'interface était finie, complètement finie, lors de la dernière soutenance. On peut constater qu'il n'en est rien, et qu'en l'occurrence de nouvelles modifications ont été apportées. En effet, notre application dispose dorénavant d'une barre de menus (*Widget* reconnu sous le nom de *QMenuBar*). Ce *Widget* est un descendant de la *QDialog*, qui rappelons le, forme la fenêtre principale de notre projet.

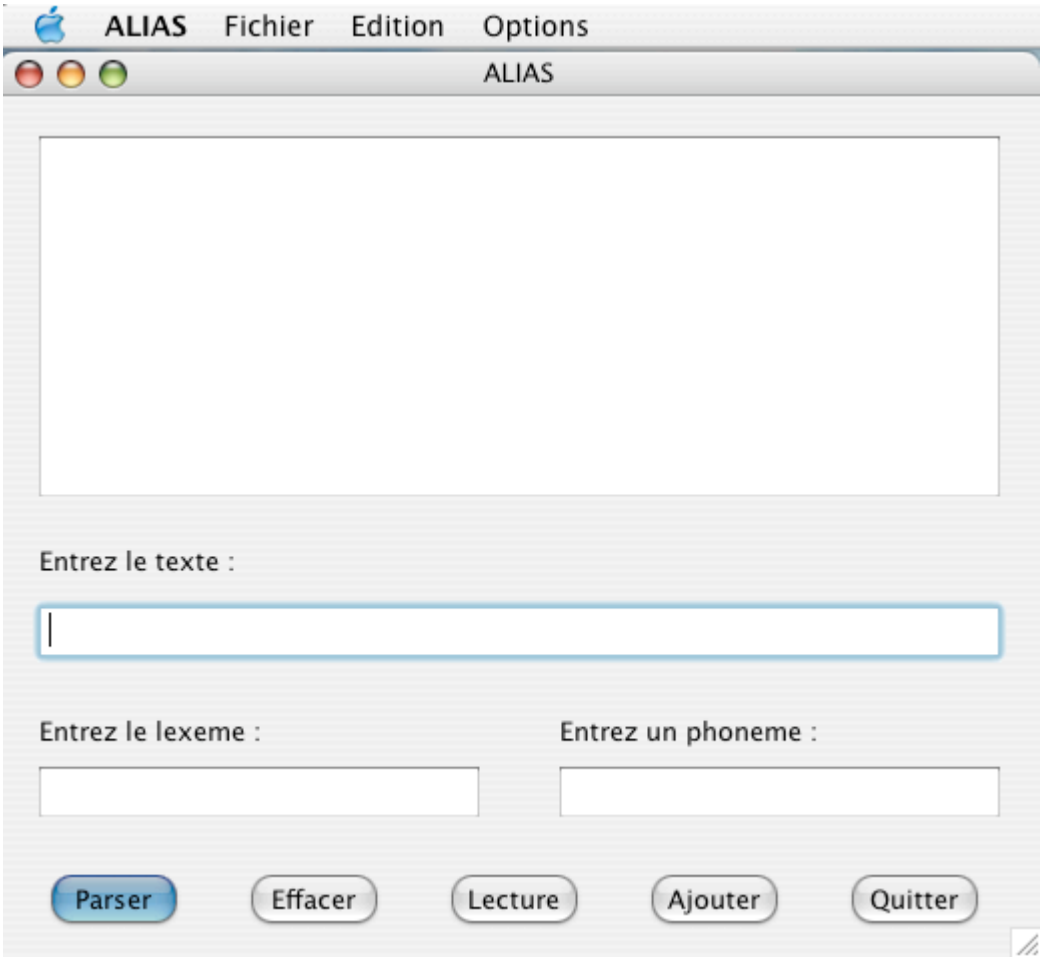

FIG. 1 – La nouvelle interface, avec la barre de menus, sous **MacOS X**

Pourquoi ce besoin impératif ? Tout d'abord, un besoin d'avoir une application un peu plus aboutie. De fait, les utilisateurs ont une plus grande

préférence pour des applications soignées, surtout faciles à manipuler. Par ailleurs, il se trouve qu'une barre de menus confère un aspect plus professionnel au projet. Et pour finir elle nous permet d'utiliser des boîtes de dialogue.

Les *QFileDialog* sont des *Widgets* descendants de la *Qdialog*. Ce *Widget* possède différentes propriétés, *slots*, *signals* tels que *GetOpenFilename*, qui nous permettent d'ouvrir une boîte de dialogue afin d'ouvrir un fichier, ou bien de l'enregistrer. Des filtres peuvent être aussi prédéfinis, pour que l'utilisateur final puisse employer à loisir un fichier sous tel ou tel format. Ainsi, pour l'enregistrement des sons, nous avons définis un filtre autorisant uniquement l'enregistrement de fichiers de type *.wav*, *.raw* ou *.ogg*.

Les *QFileDialog* sont appelées à partir de *Popupmenu*. Un *Popupmenu* est le *Widget* rend possible le fait de regrouper des exécutables, des fichiers ou des dossiers dans un menu disponible à partir d'un raccourci. Les *Popupmenu* font partis de la classe des *QPopupmenu* qui héritent des *Qdialog*.

Comme on peut le voir deux nouveaux *Widgets* sont apparus. Les premiers sont des *Qlabel* qui ont pour mission de nous permettre d'afficher du texte directement dans notre application. Ces *Widgets* sont à l'opposé des *QTextedit*, qui, rappelons le, nous permettent de saisir du texte une fois notre application lancée.

Les *QTextedit* sont là pour permettre à l'utilisateur d'insérer de nouveaux mots dans le dictionnaire depuis l'interface. Dans le premier *QTextedit*, l'utilisateur rentre le mot (ou *lexème*) et dans l'autre sa version phonétique. Un clic sur le bouton *Ajouter* et voilà notre nouveau mot ajouté directement dans le dictionnaire !

Nouveauté dans notre application : il est désormais possible d'ouvrir un fichier de type texte, de le sauvegarder pour une utilisation future.

4 Des chiffres et des lettres

Lors de la précédente soutenance, nous avons présenté un *dictionnaire* capable de gérer pour nous les mots déclenchant de trop nombreuses exceptions. Ainsi, nous avons pu montrer en exemple la décomposition en *phonèmes* du mot *cation* et du chiffre 2. Mais, les combinaisons des **10 chiffres de base** étant infinies et le *dictionnaire* étant par définition un ensemble fini, nous avons dû rechercher et écrire un nouvel algorithme dédié au traitement des nombres.

Cet algorithme nécessite l'usage de quelques fonctions préliminaires dont la détermination de la longueur du nombre —1234 a une longueur de 4— qui permet en même temps de savoir si le pointeur (de la chaîne de caractères) pointe sur un nombre, car dans le cas contraire, notre fonction retourne 0, ce qui a pour effet de prévenir le *parser* principal que c'est à lui d'agir.

La fonction principale `parse_number` utilise la fonction `is_number` pour déterminer si le prochain caractère est le début d'une suite de chiffres. `is_number` parcourt la chaîne tant que le caractère courant est un chiffre (donc un code ASCII[2] compris entre celui de '0' et celui de '9') puis retourne la longueur du nombre en terme de chiffres. Cette longueur est très importante pour choisir quel coefficient doit être attribué au paquet que nous allons traiter. Par exemple, le nombre 626921023 se découpe par paquets de trois nombres : soient 626, 921, et 023, auxquels on affecte les coefficients respectifs `MILLION`, `MILLE`, et `NO`. Ce sont des macro-constantes représentant les *phonèmes* des mots million et mille ; `NO` est juste une redéfinition de `NULL`...

Finallement, la phase de traduction s'effectue via deux fonctions et plus de cent macro-constantes représentant la suite de phonèmes associée, entre autres, aux nombres de 0 à 100.

[2] American Standard Code for Information Interchange

4.1 Traduction d'un paquet

Cette tâche est une phase essentielle à la décomposition d'un nombre en sa suite de *phonèmes* associés. La fonction `process_number` va effectuer une série d'opérations mathématiques sur la *longueur* du nombre afin de connaître l'aspect du paquet en cours de traitement. Voici un petit exemple avec le nombre (paquet) 627 provenant de 627921023 :

- `len`[3] est divisible par 3 (c'est-à-dire, que le reste de la division entière de `len` par 3 est égal à 0), donc 6 doit être considéré comme l'unité de centaine, ainsi, nous devrons enfiler les symboles de `SIX` suivis de ceux de `CENT` à ceci près que 6 génère une exception face à un coefficient car nous ne disons pas [s i s s B] mais [s i s B]. Le premier chiffre venant d'être traité, `len` est décrémentée.

- la gestion des dizaines n'induit pas de difficultés supplémentaires puisque celles-ci sont déjà présentes, nous le rappelons, sous la forme de macro-constantes. De plus, un tableau de chaînes global contient toutes ces constantes dans leur ordre croissant ; de ce fait, il est facile d'accéder aux symboles du nombre des dizaines :
  ```
  gl_numbers[10*(numb[k]-'0')+numb[k+1]-'0']
  ```
  où `gl_numbers` est le tableau de chaînes, `numb`, la chaîne pointant sur une représentation du nombre et `k`, la position dans cette chaîne. `len` est décrémentée de 2. Il est utile pour la suite de préciser que `len` est passée par référence à la fonction `process_number`.

- il ne reste plus qu'à enfiler les *phonèmes* du coefficient (s'il y en a un). La manière de déterminer s'il s'agit de `MILLIARD`, `MILLION`, `MILLE`, ou `NO` relève de la fonction principale d'analyse du nombre (`parse_number`).

4.2 Diviser pour régner

Il faut établir une règle de base (un cas d'arrêt) à notre algorithme de traitement des nombres. Cette règle se charge de traduire les nombres de 0 à 999999999999. Grâce à cette règle nous pouvons ensuite gérer l'infini[4] (évidemment limité par la représentation des entiers `long` —et même `long long`— en mémoire). Dans les deux cas, l'algorithme boucle tant que `len` est strictement positive.

[3] len = la longueur du nombre à parser

[4] De nos jours l'infini a une limite assez lointaine ©Krisboul

4.2.1 Domaine fini : moins de 12 chiffres

Plusieurs cas se présentent à nous :

- `len` est strictement supérieure à 9 et inférieure à 12 : il s'agit d'un paquet appartenant aux milliards.
- `len` est strictement supérieure à 6 et inférieure à 9 : il s'agit d'un paquet appartenant aux millions.
- `len` est strictement supérieure à 3 et inférieure à 6 : il s'agit d'un paquet appartenant aux milliers.
- `len` est inférieure à 3 : il s'agit d'un paquet non coefficienté (d'où l'usage de `NO`).

4.2.2 Domaine infini : plus de 12 chiffres

Le plus dur est de déterminer dans quel domaine (`MILLIARD`, `MILLION`, etc.) le prochain paquet va se trouver. Encore une fois, une série (un peu plus importante) de tests est nécessaire :

- `len` est congrue à 3 modulo 9 : il s'agit d'un paquet appartenant aux milliards.
- `len` est divisible par 9 : il s'agit d'un paquet appartenant aux millions.
- `len` est congrue à 6 modulo 9 : il s'agit d'un paquet appartenant aux milliers.

Par ailleurs, on effectue deux autres tests similaires à chaque expression conditionnelle, avec les changements de variables `len1` $=$ `len` $+ 1$ et `len2` $=$ `len` $+ 2$.

Lorsque le nombre a été converti dans sa suite de *phonèmes*, on déplace (incrémente) l'index (de la chaîne principale) `i` de la longueur `len` afin de permettre au *parser* de continuer sa tâche…

5 Partie audio

5.1 Inclusion du son dans l'interface

Convertir du texte en *phonèmes* peut être très intéressant pour apprendre le français, mais s'il peut être lu, cela lui confère une utilité. Nous avons donc réalisé pour cette soutenance la liaison entre les deux parties les plus visibles du projet, à savoir l'interface et la sortie audio.

5.1.1 Correspondance phonème et nom de fichier

Le *parser* utilise une file de *phonèmes* en sortie, et nous utilisons pour la sortie audio, une file de noms de fichiers. Il a donc été nécessaire d'utiliser une fonction de conversion qui associe une liste de *phonèmes* à une liste de fichiers. Pour l'instant il s'agit d'un tableau à double entrée, mais il est prévu d'utiliser un fichier de configuration pour une plus grande souplesse pour l'utilisateur.

5.1.2 Inclusion du son dans l'interface

Le bouton "lecture" et les possibilités de sauvegardes ont étés reliés à la partie audio, et, la sortie audio est initialisée avec le lancement du programme. Comme cette liaison a été prévue depuis le début du développement, elle s'est effectuée sans difficulté majeure, et nous pouvons maintenant entendre les sorties du *parser*, pourvu qu'elles n'utilisent que des *phonèmes* implémentés (enregistrés).

5.2 Utilisation des Autotools

Le Code a été intégralement revu, dans le but de faciliter le portage. Ainsi, nous utilisons les autotools pour utiliser ou non, les sorties *ALSA* et *OSS*. Ce travail a été difficile, car les deux bibliothèques n'utilisent pas les mêmes unités pour leur sorties. Ainsi *ALSA* compte en *frames*, tandis que *OSS* compte en *octets*. Nous avons aussi rajouté une sortie nulle (ou sortie *virtuelle*), permettant d'utiliser le projet sur des configurations sans sorties sonores, ou de permettre de lancer le projet sur des architectures non supportées. Il permet aussi d'accélerer le traitement en vue de conversions en masse, car nous ne devons pas attendre la carte son dans ce cas là.

5.3 Exportation vers les différents formats

Notre *Text-to-Speech* a de nombreuses utilités, mais une fonctionnalité souvent recherchée par les utilisateurs est la possibilité de sauvegarder la sortie, dans le but de la réutiliser. C'est pourquoi nous avons implémenté, les sorties en format *PCM*, *WAVE*, et *Ogg/Vorbis*.

5.3.1 Le format *PCM* brut : le *RAW*

Le format *PCM* (*Pulse Code Modulation*), est le format que nous utilisons depuis le début en interne. Il peut donc nous être utile de l'exporter à des fins de débogage. De plus, comme tous les programmes travaillant le son utilisent ce format en interne, ou presque, il peut être intéressant pour l'utilisateur final de disposer de fichiers en *RAW* pour les inclure dans des programmes simplistes. Nos fichiers *RAW* utilisent le format interne à notre programme, à savoir des données 16 bits signées, en stéréo entrelacée, et à une fréquene de 44.1KHz, ce qui est de loin le format le plus courant de nos jours.

5.3.2 Le format Microsoft *WAVE* (ou *WAV*)

Cependant, le format PCM n'est pas directement lisible, car il faut que l'utilisateur connaisse le format des données (par exemple, sur combien de bits sont stockés les informations), pour pouvoir les lire. Ainsi, nous avons choisi d'exporter le son dans un format très standard depuis son utilisation dans *Microsoft Windows*, à savoir le format *WAVE*. Le format *WAVE* est supporté par la majorité des systèmes d'exploitations et systèmes traitant le son que cela soit pour le lire ou bien pour le modifier. Les fichiers *WAVE* sont conformes aux spécifications *RIFF* (*Resource Interchange File Format*). Mais que sont les spécifications *RIFF* ? Il s'agit d'ajouter à des fichiers multimédia des en-têtes, ou *CHUNK*, permettant de délimiter les différentes parties d'un fichier, comme par exemple, plusieurs bandes sons pour un film, ou des informations sur les droits d'auteurs pour de la musique, ou même des index de pistes (*tracks*), permettant de changer de chanson dans un album. Dans notre cas, il s'agit d'inclure les informations sur le format des données *PCM* dans le fichier lui même, et ainsi, permettre sa lecture sur n'importe quel lecteur.
À la création du fichier *WAVE*, nous connaissons quasiment tous les paramètres du fichier, à savoir, qu'il n'y a qu'une seule piste au format *PCM*, qu'il s'agit de données stéréo, avec une fréquence de 44.1KHz, et que la profondeur est de 16 bits. Il ne nous reste plus qu'à calculer le nombre

d'octets par seconde, et le nombre d'octets par échantillonage utilisés, et, mise à part la durée totale du fichier, nous pouvons donc écrire dans le fichier tous les paramètres. Ensuite, nous écrivons les données *PCM* qui sont heureusement dans le même format que l'impose la norme *WAVE*, c'est-à-dire des données signées et entrelacées. Enfin, quand le fichier est terminé, nous n'avons plus qu'à remplir la taille dans les en-têtes.

5.3.3 Le format *Ogg/Vorbis*

Le format *WAVE* est un format standard, cependant, il est non compressé, il peut donc être intéressant d'utiliser un format qui l'est. Comme nous imposons déjà à l'utilisateur de posséder les bibliothèques *Ogg/Vorbis*, il y a de fortes chances qu'il possède le programme de compression *oggenc* qui fait partie du même projet. Nous utilisons ce programme externe à notre projet pour compresser les sorties en *WAVE*. Nous permettons à l'utilisateur de choisir son programme de compression, selon la configuration de sa machine, et les éventuelles licences qu'il pourrait acquérir. Comme implémenter des algorithmes de compression n'est pas le but premier de notre projet, ce n'est pas bien gênant. Si le programme est absent de la machine de l'utilisateur, cette fonctionnalitée est tout simplement ignorée.

6 Site web

À la demande de notre client préféré, le logo de notre hébergeur[5] a été réduit. Voici une capture d'écran de notre site. La page de caractéristiques où l'on peut trouver nos rapports de soutenance (y compris celui-ci) a été mise à jour.

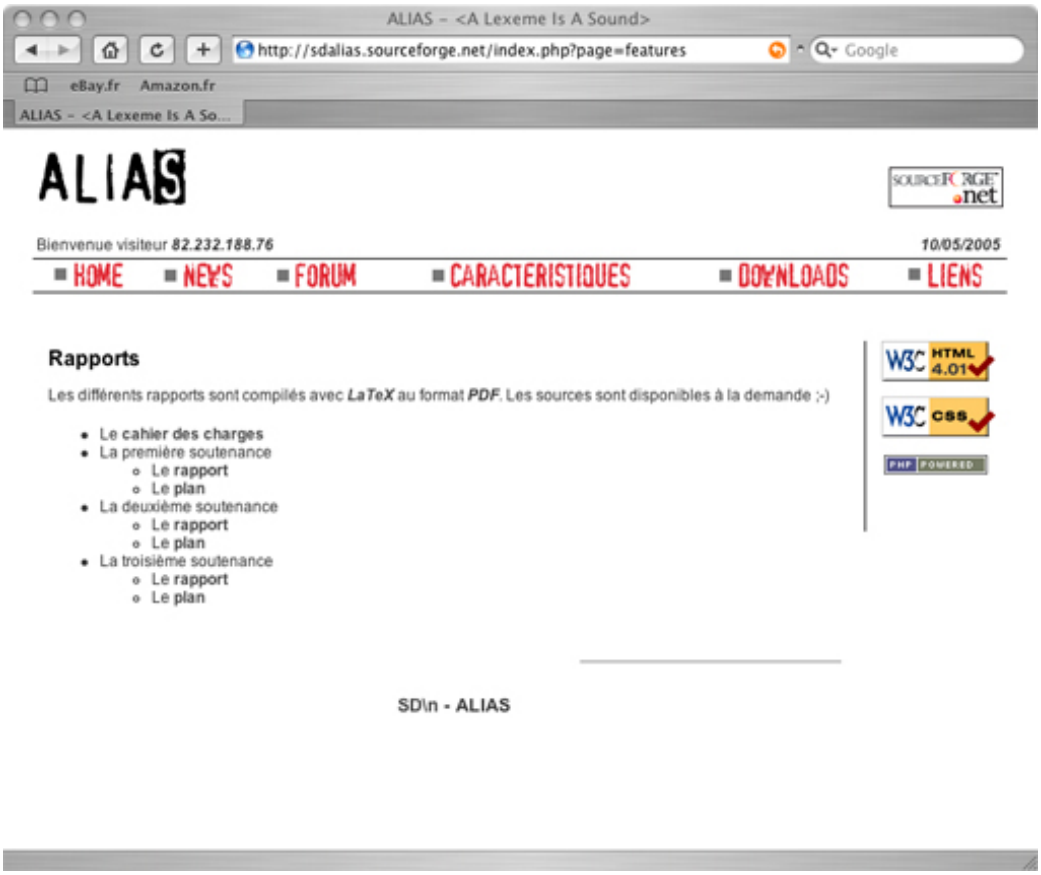

FIG. 2 – La page "CARACTÉRISTIQUES" du site

[5]http ://sourceforge.net

7 Conclusion

Troisième partie du projet terminée. Mission S-III terminée. Tous les objectifs que nous nous étions fixés pour cette soutenance ont été réalisé avec succès par les membres de l'équipe.
Enfin la dernière ligne droite. Il ne nous reste plus que six semaines pour accomplir notre ultime mission, nom de code S-IV. Sur notre route, la plus dangereuse organisation maléfique nous barrera la route, je veux bien sûr parler de *la semaine de partiels* avec son lot d'ennemis tous les plus redoutables les uns que les autres. Mais nous serons braves, nous ne baisserons pas la tête devant les dangers qui nous menace, et c'est ainsi qu'en cette semaine du 20 juin 2005, le projet **ALIAS** verra enfin le jour dans sa première version entièrement fonctionnelle.

Troisième rapport terminé.