Ecole Nationale Supérieure des Télécommunications

Etude de l'intégration de l'interpréteur Java Kaffe avec le navigateur Web mMosaic

Mémoire de fin d'études
Juillet 1997
Olivier PILLON
pillon@mafjet.fr

Table des matières

1. PRÉSENTATION 3

1.1. KAFFE 3
1.2. MMOSAIC 3
1.3. BESOINS ET OBJECTIFS 4
1.4. QUELQUES ADRESSES UTILES 4

2. IDENTIFICATION DES DIFFICULTÉS POTENTIELLES 5

2.1. VARIABLES GLOBALES ET/OU STATIQUES 5
2.1.1. Rappels de langage C 6
2.1.2. Clashs de noms entre Kaffe et mMosaic 6
2.1.3. Partage des variables globales et/ou statiques par plusieurs instances de Kaffe 7
2.2. SIGNAUX / EXCEPTIONS 8
2.3. EVÉNEMENTS X-WINDOW 8
2.3.1. Rappels X-Window 8
2.3.2. Redirection des événements entre mMosaic et Kaffe 9
2.4. THREADS 9
2.5. FONCTIONNALITÉS DEVANT ÊTRE RAJOUTÉES ÀKAFFE ET MMOSAIC 10
2.6. DIFFICULTÉS INDUITES 10

3. DESCRIPTION DU FONCTIONNEMENT DE KAFFE 11

3.1. SYSTÈME GRAPHIQUE 11
3.1.1. Principe général 11
*3.1.2. Structures de données principales ..... **Erreur! Signet non défini.***
3.1.3. Initialisation 12
3.1.4. Gestion des fenêtres 12
3.1.5. Boucle d'évènements 12
3.2. GESTIONNAIRE DE THREADS 13
3.2.1. Structures de données principales 13
3.2.2. Initialisation 14
3.2.3. Fonctionnement 14
3.3. GARBAGE COLLECTOR 15
3.3.1. Principe général 15
3.3.2. Structures de données principales 17
3.3.3. Initialisation 18
3.3.4. Fonctionnement 18
3.4. GESTIONNAIRE DE CLASSES 19
3.4.1. Principe général 19
3.4.2. Structures de données principales 20
3.4.3. Fonctionnement 21

4. SOLUTIONS PROPOSÉES 22

5. CONCLUSION 23

1. Présentation

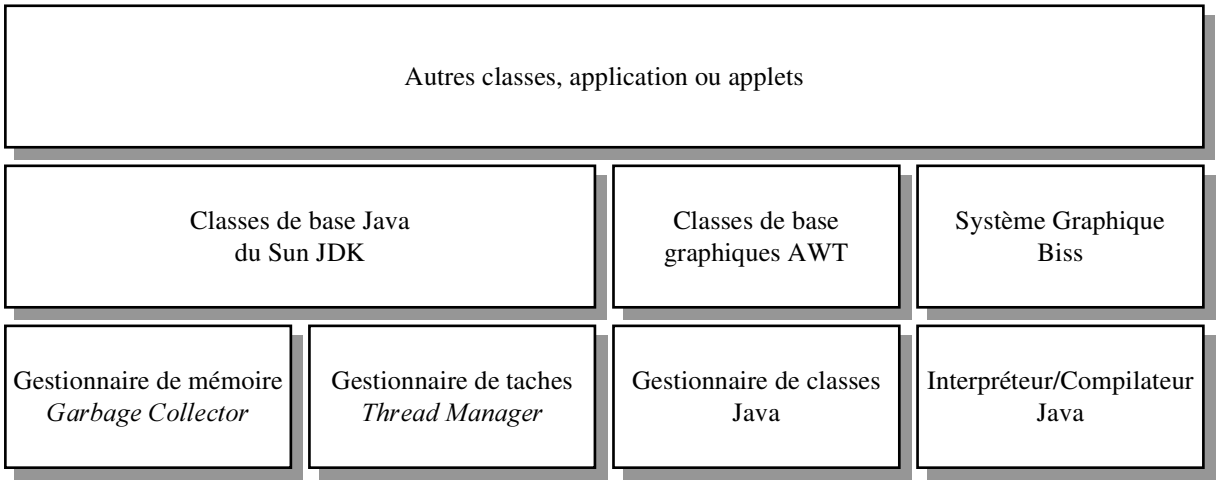
1.1. Kaffe

Kaffe est un projet domaine public de machine virtuelle exécutant du code Java. Ce projet, lancé et dirigé activement par le britannique Tim Wilkinson, comprend un interpréteur/compilateur Java. Kaffe est compatible avec les standards définis par Sun. Kaffe est disponible sur quasiment tous les environnements (Unix, Windows, Macintosh, ...) et tous les processeurs (Intel, Sparc, Alpha, PowerPC, 68K, Mips). Les fonctionnalités et les commandes de Kaffe sont très proches de celles du JDK de Sun.

L'architecture de Kaffe se décompose en:

- Un gestionnaire de mémoire de type *Garbage Collector*. Conformément à la définition du langage Java, la mémoire est allouée explicitement par le programmeur, comme en langage C, mais le système se charge de la désallouer automatiquement lorsqu'elle n'est plus utilisée.
- Un système multithread non préemptif indépendant du système d'exploitation sous-jacent.
- Un gestionnaire de classes Java.
- Un interpréteur/JIT Java.

Les classes Java de base proviennent du JDK de Sun.
Les classes graphiques proposées sont celles de la librairie graphique Biss-AWT, et remplacent celles de Sun.



1.2. mMosaic

mMosaic est un navigateur Web du domaine public dérivé du NCSA XMosaic. De nombreuses extensions ont déjà été développées, dont le support de nouveaux marqueurs HTML et des fonctionnalités de multicast étendues.

1.3. Besoins et objectifs

L'objectif de mMosaic étant d'être un navigateur de référence dans le domaine public, mMosaic se doit de supporter les applets Java. Les modes de licence du JDK de Sun étant incompatibles avec un statut de domaine public, le choix d'un interpréteur Java s'est orienté vers le projet Kaffe. Le but de ce travail est d'étudier l'intégration de Kaffe dans mMosaic. Kaffe n'ayant pas été conçu pour fonctionner à l'intérieur d'un navigateur, cette intégration ne sera pas sans soulever de nombreux problèmes.

La démarche proposée est la suivante:

1. Identifier et répertorier les difficultés potentielles *a priori.*
2. Rechercher des solutions potentielles pour chacun de ces obstacles.
3. Analyser l'architecture et le fonctionnement de Kaffe.
4. En déduire les solutions à sélectionner.
5. Réaliser une maquette pour valider les solutions retenues.

1.4. Quelques adresses utiles

A propos de Kaffe:

Serveur Web:	*www.kaffe.org*
Mailing List:	*kaffe@kaffe.org*

A propos de mMosaic:

Serveur Web:	*www-sig.enst.fr/~dauphin/mMosaic*

2. Identification des difficultés potentielles

L'intégration de Kaffe dans mMosaic peut être envisagée de deux manières :

1. mMosaic est un premier processus, et chaque applet Java est un processus distinct.
2. Chaque applet Java est un thread dans le processus mMosaic.

Le choix entre ces deux options est lourd de conséquences. C'est un choix stratégique. Les avantages et inconvénients principaux de chacune de ces options sont :

- L'option 1 demande beaucoup moins de modifications sur Kaffe que l'option 2.
- L'option 2 est plus performante et moins gourmande en ressources (machine, système, mémoire,...). Cet avantage se fera d'autant plus sentir que le nombre d'applets Java concomitantes sera grand. Mais cette option est beaucoup plus compliquée à mettre en oeuvre, car ni Kaffe ni mMosaic n'ont été conçus dans cette optique.

L'ambition de mMosaic de devenir un navigateur de référence semble pousser vers la solution offrant le meilleur résultat final, quel que soit le travail à fournir. Le choix portera donc sur l'option 2.

L'objectif retenu est de réaliser une intégration complète de Kaffe et mMosaic dans un même processus.

2.1. Variables globales et/ou statiques

Le code source de Kaffe (langage C) contient un grand nombre de variables globales, statiques ou non statiques. Certaines fonctions contiennent aussi des variables statiques. Le code de mMosaic comprend aussi des variables globales.

De cette situation découlent deux difficultés :

- Des clashs de noms éventuels entre variables globales non statiques de Kaffe et mMosaic.
- Le partage des mêmes variables par toutes les instances de Kaffe d'un même navigateur mMosaic.

2.1.1. Rappels de langage C

- Variables globales :
  Une variable définie en dehors du corps d'une fonction est une variable globale. Une variable globale peut être accédée depuis n'importe quelle partie du code grâce au mot clef 'extern'. La durée de vie d'une variable globale est celle du programme. Les variables globales sont résolues au moment du link.

  Ex :

```
/* main.c */
int             processId ;     /* instanciation d'une variable globale dans main.c */
.........
/* toto.c */
extern int      processId ;     /* reference à une variable globale dans toto.c */
```

- Variables globales statiques :
  Une variable globale statique est identique à une variable globale sauf que sa visibilité est réduite au fichier source dans lequel elle est instanciée.

  Ex :

```
/* main.c */
static int      nargs ;   /* instanciation d'une variable globale statique dans main.c */
```

- Variables locales statiques :
  Une variable locale statique est locale à une fonction donnée, mais persiste entre appels de fonction successifs. C'est un moyen de stocker de façon permanente des données à l'usage exclusif d'une seule fonction.

  Ex :

```
int compteur()
{
        static int      n = 0;    /* instanciation d'une variable locale statique */
        n++ ;
        return n ;
}
```

2.1.2. Clashs de noms entre Kaffe et mMosaic

Les variables globales sont résolues au moment du link. Une variable globale n'est identifiée que par son nom. Par conséquent, le link échouera si deux variables globales ont le même nom. Cette situation est susceptible de se produire entre Kaffe et mMosaic.

La solution consiste à renommer les variables fautives dans mMosaic. De simples 'Replace All' devraient suffire.

2.1.3. Partage des variables globales et/ou statiques par plusieurs instances de Kaffe

Une page Web comprend fréquemment plusieurs applets Java. mMosaic devra instancier un Kaffe pour chacune de ces applets. Ces différentes instances, chacune matérialisée par un ou plusieurs threads, vont cohabiter dans le même processus mMosaic. Toutes ces instances vont accéder aux mêmes variables globales et/ou statiques de manière anarchique, ce qui aura des conséquences immédiates et dramatiques.

Kaffe possède un nombre assez important de variables globales et/ou statiques : de l'ordre de la centaine rien que pour les variables globales du noyau central (machine virtuelle).

Une première solution consiste à rassembler toutes les variables globales propres à une instance de Kaffe dans une structure C. Chaque instance de Kaffe possédera sa propre structure.

Dans la version actuelle de Kaffe, chaque thread possède déjà une structure nommée *'_thread'* qui sert aux changements de contexte. Ce serait l'endroit idéal pour rajouter nos variables globales:

```
/* globals.h */
typedef struct _threadGlobals
{
        /* all necessary globals */
        char    gC ;
        int     gN ;
        ..........
} threadGlobals ;

/* thread.h */
typedef struct _thread
{
        .........
        threadGlobals *   itsGlobals ;
} thread ;
extern thread*     currentThread ;
```

Les changements à effectuer sont :

- création et initialisation d'une structure 'threadGlobals ' lors de la création d'une instance de Kaffe par mMosaic.
- initialisation du pointeur sur cette structure depuis *'thread'* à chaque création d'un thread Java (fonction *allocThreadCtx*)
- mise à jour de toutes les références à des variables globales :
  gN devient *currentThread->itsGlobals->gN*

Cette solution, un peu brutale, a le mérite d'être assez mécanique. Cependant, cette solution ne peut pas s'appliquer aux variables statiques.

Une deuxième solution consiste à faire du cas par cas. Par exemple, il est tout à fait concevable de rendre le Garbage Collector commun à toutes les instances de Kaffe.
Cette solution, plus délicate à mettre en oeuvre que la première, nécessite une analyse et une compréhension plus précise des mécanismes de Kaffe, mais est peut être plus performante et plus générique.

2.2. Signaux / Exceptions

Comme tout programme Unix qui se respecte, Kaffe et mMosaic utilisent et traitent un certain nombre de signaux. Le mécanisme de signaux Unix ne connaît pas la notion de thread : la gestion des signaux est commune à tout le processus.

Certains signaux n'ont pas de raison d'être destinés à un thread précis mais à l'ensemble des threads. C'est le cas notamment de SIGHUP, SIGKILL, SIGSEGV, SIGTERM. Le traitement de ces signaux pourra rester au niveau du processus.

D'autres signaux ont plus vocation à être destinés à un thread spécifique. C'est le cas notamment de SIGUSR1, SIGALRM, SIGIO. Le traitement de ces signaux devra être capable de rediriger chaque signal sur le bon thread.

Les signaux suivants sont traités par Kaffe :

- SIGSEGV
- SIGBUS
- SIGFPE
- SIGPIPE
- SIGALRM

Les quatre premiers sont définis au lancement de Kaffe. Leur traitement, tout à fait classique, sera facile à intégrer à celui de mMosaic.

SIGALRM est utilisé par Kaffe pour le système de threads. Si mMosaic ne fait pas usage de ce signal, il ne devrait pas y avoir de complication. Dans le cas contraire, il faudra mixer le traitement fait par mMosaic avec celui fait par Kaffe.

A première vue, les difficultés soulevées par les signaux ne devraient pas être insurmontables, car Kaffe n'en fait pas un usage excessif.

Les changements à effectuer sont :

- modification des gestionnaires de signaux de mMosaic.
- modification des gestionnaires de signaux de Kaffe.

2.3. Evénements X-Window

Le système graphique X-Window ne connaît pas la notion de thread. La file d'attente des messages X-Window est commune à tout le processus. La boucle de traitement de ces messages devra donc être capable de rediriger chaque message sur le thread correspondant.

2.3.1. Rappels X-Window

X-Window est l'environnement graphique utilisateur des systèmes Unix. Mise à part ses caractéristiques client/serveur, X-Window ressemble, du point de vue du programmeur, à tous les autres environnements graphiques. A chaque application/processus est associée une file d'attente d'événements. Le système X-Window rajoute des événements dans cette file d'attente. L'application vient piocher dans sa file d'attente afin de traiter les événements retirés.

Le schéma d'une application graphique sera souvent le suivant :

```
void main
{
        /* initialize application */
        ........
        /* event loop */
        while(1)
        {
                /* get next event */
                XtAppNextEvent(appContext, &event) ;
                /* dispatch event */
                XtDispatchEvent(&event);
        }
}
```

2.3.2. Redirection des événements entre mMosaic et Kaffe

La file d'attente des événements est commune à tout le processus. Seront donc mélangés dans la même file d'attente des messages destinés à mMosaic et aux différentes instances de Kaffe. Le mécanisme de traitement des événements de mMosaic doit être modifié pour pouvoir rediriger les événements qui ne lui sont pas destinés aux bonnes instances de Kaffe.

Tous les objets graphiques d'une application X-Window sont organisés en une structure d'arbre. Chaque objet est une fenêtre fille d'une autre. La structure *XEvent* retournée par *XtAppNextEvent* indique l'objet X-Window auquel le message est destiné dans le champ *window*. A partir du champ *window*, il est possible de remonter dans l'arbre des fenêtres (*XQueryTree*). En remontant, il sera possible d'identifier la fenêtre mère d'une instance Kaffe. Sinon, le message sera destiné à mMosaic lui-même.

Les changements à effectuer sont :

- modification de la boucle d'événements de mMosaic.
- modification de la boucle d'événements de Kaffe.

2.4. Threads

La conversion d'un programme en mode multithread soulève toujours des problèmes de synchronisation, de timing, de sections critiques, surtout lorsque le programme initial, comme mMosaic, n'a pas été conçu dans l'optique multithread.

De plus, le multithread de Kaffe n'est pas preemptif, du moins dans la version actuelle 0.8.4 de Kaffe et sur les environnements Linux et Solaris.

Ainsi, une application Java mal programmée peut garder la main et bloquer définitivement l'ensemble mMosaic/Kaffe.

2.5. Fonctionnalités devant être rajoutées à Kaffe et mMosaic

Au delà de tous les obstacles énumérés plus haut, un certain nombre de fonctions spécifiques à la position d'une applet dans un navigateur devront être rajoutées :

- reconnaissance par mMosaic du tag <Applet> et de ses options.
- collaboration entre mMosaic et Kaffe pour le téléchargement des différentes classes de l'applet Java (Le tag <Applet> ne référence que la classe d'entrée. Seule l'analyse des pseudo-codes par Kaffe indique les références des autres classes utilisées par l'applet).
- mise en place des fonctions de communication entre le navigateur et les applets telles que définies dans les spécifications du JDK de Sun.
- mise en place éventuelle des protections contre des applets malveillantes telles que celles proposées par les navigateurs commerciaux (Netscape Navigator, Microsoft Explorer, ...).

2.6. Difficultés induites

L'intégration de Kaffe dans mMosaic nécessitera de profondes modifications du code de Kaffe. Kaffe est un produit encore jeune et qui évolue très vite. Sera-t-il possible de réintégrer nos évolutions dans la version officielle de Kaffe ? Si non, comment suivre le rythme des mises à jour, nouvelles version, corrections de bugs, et support des nouveaux standards (JDK 1.1, ...) ?

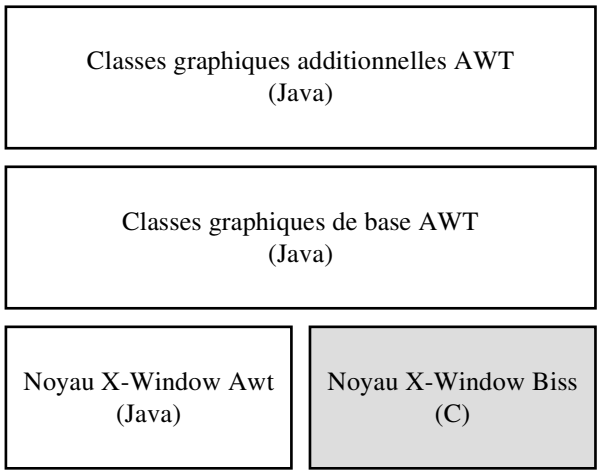
Après plusieurs mois d'utilisation de Kaffe par Gilles Dauphin et moi-même, Kaffe semble un produit très peu stable, malgré l'activité intense dont témoigne la mailing list kaffe@kaffe.org. Il semble difficile d'en envisager une utilisation courante dans un browser Web grand public tant que le produit n'est pas stabilisé.

3. Description du fonctionnement de Kaffe

3.1. Système graphique

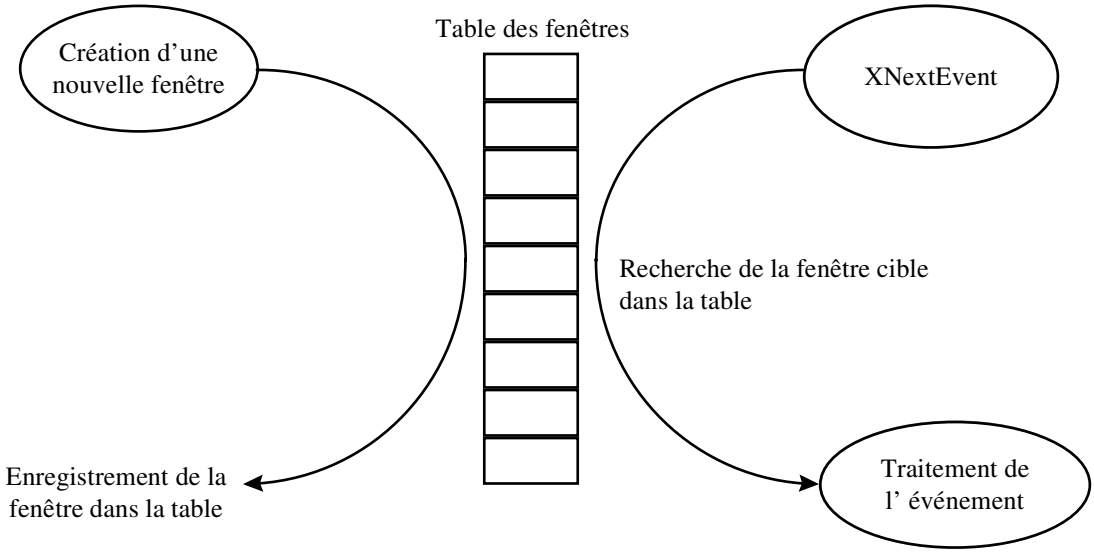
3.1.1. Principe général de Biss-AWT

La librairie graphique Biss-AWT comporte à la fois du code C et du code Java. Le schéma suivant illustre l'architecture de Biss-AWT:



La partie Biss écrite en C concentre tous les appels aux fonctions natives X-Windows (Java ne permet pas l'appel direct de fonctions systèmes). Biss assure aussi la gestion d'une liste des fenêtres (au sens X-Window du terme). Toute fenêtre créée est ajoutée à cette liste. Le cœur du système graphique remplit les fonctions très simples suivantes:

- initialisation de X-Window.
- gestion de la liste des fenêtres par une table de hachage.
- création et destruction des fenêtres.
- traitement des évènements X-Window.



Le noyau AWT écrit en Java assure principalement:

- l'encapsulation des appels des fonctions C du noyau Biss.
- la boucle principale d'événements.

La boucle principale d'événements se fait dans un thread dédié.

3.1.2. Initialisation de X-Window

X-Window est initialisé dans Biss au lancement de Kaffe:

packages/biss-net.com/APIawt/lib/display.c:

➥ void initialize (HNativeLib lib)

XSetErrorHandler(xErrorHandler);
XOpenDisplay(dspName);

3.1.3. Gestion de la liste des fenêtres

Biss assure la gestion de la liste des fenêtres dans les deux fonctions C suivantes:

- registerPeer ajoute une nouvelle fonction à la liste.
- lookupPeer recherche une fenêtre dans la liste.

L'existence dans Biss de ce gestionnaire de fenêtres est extrêmement utile pour l'intégration avec mMosaic: une simple recherche dans cette liste permet de savoir si un événement X donné est destiné à Kaffe ou à mMosaic.

packages/biss-net.com/APIawt/lib/display.c:

➥ int registerPeer (HNativeLib lib, HPeer peer)

les nouvelles windows sont placées dans le tableau lib-> PeerTable en utilisant une méthode de hashage simple.

➥ int lookupPeer(HNativeLib lib, Window wnd)

recherche une window particulière parmi celles enregistrées dans la table de hashage.

3.1.4. Boucle d'événements

La boucle d'événements proprement dite se situe dans le noyau AWT écrit en Java: c'est une simple boucle infinie.

biss/src/biss/awt/kernel/Toolkit.java:

```
➥ public void run () {
        Peer wnd;
        int  idx;

        while ( true ) {
                try {
                        while ( true ) {
                                while ( (idx = Lib.getNextEvent()) >= 0 ) {
                                        wnd = PeerTable[idx];

                                        if ( Lib.NextEvent.id == Event.WINDOW_EXPOSE ) {
                                                PaintGraphics.Wnd = wnd;
                                                PaintGraphics.ClipRect = null;
                                                Lib.initPaintGraphics( PaintGraphics);
                                                wnd.Target.paint( PaintGraphics);
                                                Lib.releasePaintGraphics( PaintGraphics);
                                        }

                                        wnd.handleEvent( Lib.NextEvent);
                                }
                                if ( !Lib.AttentionHint )
                                        Thread.sleep( 30);
                        }
                }
                catch ( Exception x ) {
                        x.printStackTrace();
                }
        }
}
```

Cette boucle d'événements fait appel à la fonction Biss *getNextEvent*, dont le rôle est de récupérer les événements X en attente et de les prétraiter. La suite du traitement se fait de nouveau dans la partie Java de Awt par l'appel des fonctions Java correspondantes. Par exemple, les événements de type *Expose* déclenchent un appel de la fonction Java *paint* de la fenêtre correspondante.

packages/biss-net.com/APIawt/lib/display.c:

```
➥ jint getNextEvent (HNativeLib lib)
        while(1)
        {
                XNextEvent( dsp, &evt);
                lookupPeer( lib, evt.xany.window);
                switch ( evt.xany.type )
                {
                        case :
                        .....
                        return;
                }
        }
```

On remarque ici que le traitement de tout événement commence par la recherche de la fenêtre destinataire. Kaffe sait ainsi immédiatement si un événement lui est bien destiné ou non. Cela permettra de rediriger les événements appropriés sur mMosaic.

3.2. Gestionnaire de threads

3.2.1. Principe général

Les threads Kaffe n'utilisent pas les librairies de threads disponibles sur certains systèmes (Solaris, Windows NT, ...). L'équipe de Kaffe a préféré, sans doute pour des soucis de portabilité, développer son propre gestionnaire de threads. La structure du gestionnaire de threads de Kaffe calque de très prés le principe des threads en Java.

Ainsi, un thread possède les fonctionnalités suivantes:

- start(): démarrage d'un thread
- stop(): arrêt d'un thread
- sleep(s): suspension d'un thread pendant s millisecondes
- yield(): permet de donner la main aux autres threads

Les threads de Kaffe ne sont pas préemptifs: si un thread refuse de rendre la main, les autres n'auront jamais l'occasion de s'exécuter! D'où l'importance des fonctions *yield* et *sleep* qui permettent de donner la main aux autres threads.

La fonction *sleep* est réalisée avec le signal SIGALRM.

3.2.2. Initialisation

Les structures du gestionnaire de threads sont initialisées au lancement de Kaffe dans la fonction initThread.

kaffe/kaffe/thread.c:

➥ void initThread () (appelée depuis main())

- *charge la classe java/lang/thread*
- *crée le thread principal*
- *crée deux threads pour le garbage collector*

3.2.3. Fonctionnement

Les fonctions de gestion des threads (création, priorités, ordonnancement) sont regroupées dans le fichier thread.c.

kaffe/kaffe/thread.c:

➥ void alarmException(int sig) (gestionnaire du signal SIGALRM)
quitte le thread actuel pour le suivant préempté

➥ void reschedule()
effectue le changement de contexte nécessaire entre deux threads

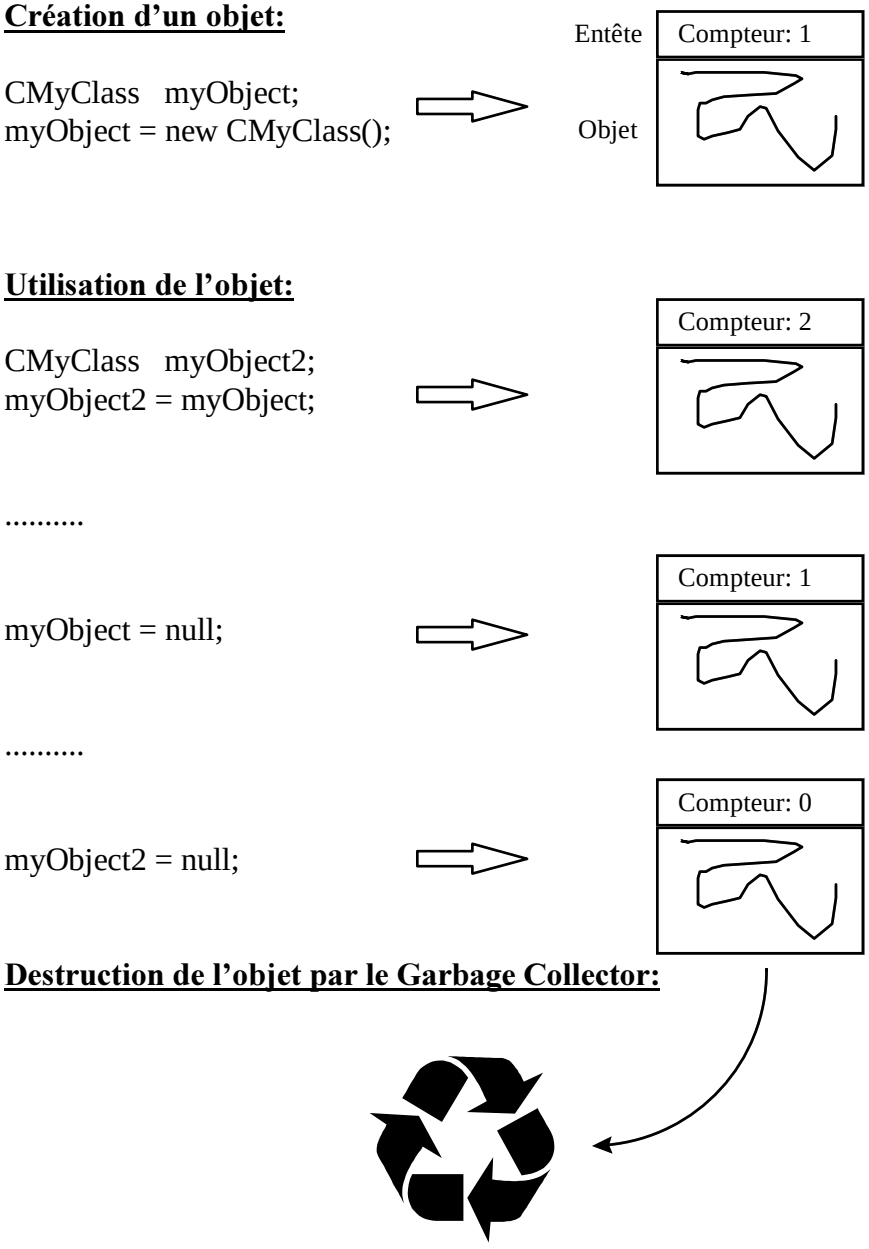
➥ void iresumeThread(thread* tid)
ajoute un thread à la liste des threads concurrents, et appelle reschedule() si la priorité de ce thread est supérieure à celle du thread actuel

3.3. Garbage Collector

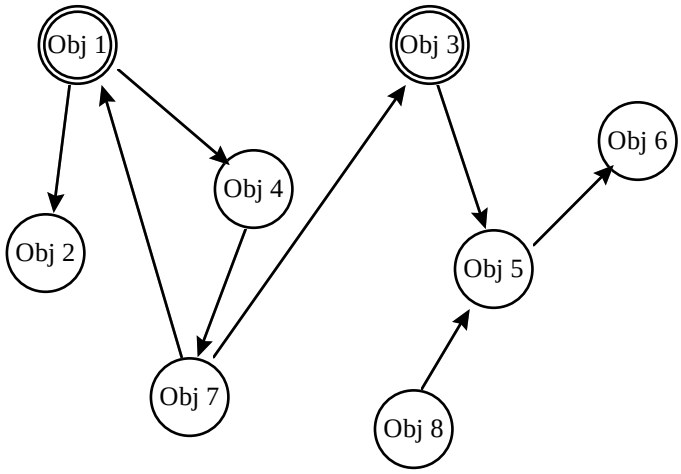
3.3.1. Principe général

Dans les langages traditionnels de type C, C++, Pascal, le programmeur est habitué à gérer lui-même la mémoire dynamique: allouer la mémoire nécessaire (malloc, new, ...), désallouer cette mémoire lorsqu'elle n'est plus utilisée (free, delete, ...). Cette tache est d'ailleurs l'une des plus prenantes et des plus source d'erreurs.

Sensibles à cette difficulté, les concepteurs du langage Java ont grandement allégé et automatisé la gestion de la mémoire. Le programmeur alloue de la mémoire lorsque nécessaire avec l'instruction *new*, mais ne la libère jamais (il n'y a pas d'instruction de type *free* ou *delete*). Le moteur de Java comptabilise le nombre de pointeurs qui référencent chaque objet. Lorsque ce compteur tombe à zéro, cela signifie que l'objet n'est plus utilisé, et il pourra être détruit.



Le Garbage Collector fonctionne selon le principe habituel des garbage collectors incrémentaux: par coloration des objets. L'ensemble des objets avec leurs pointeurs peut être représenté par une structure de graphe orienté.

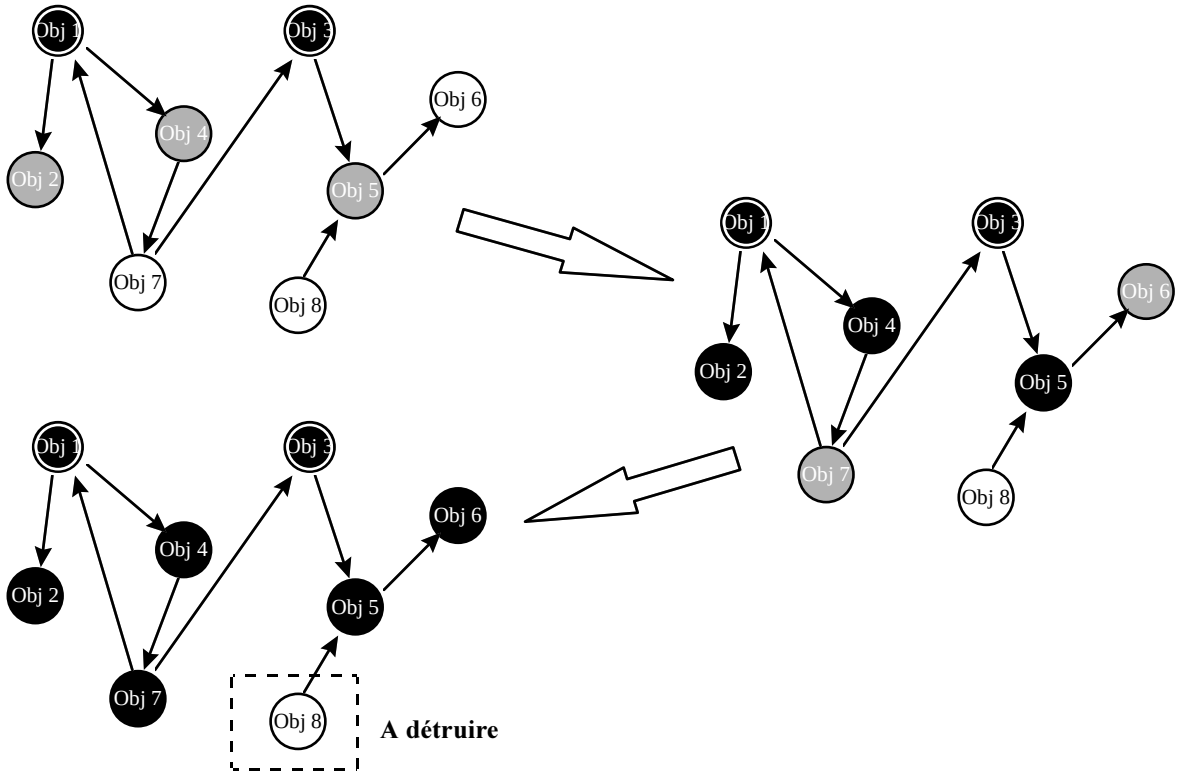


Les ronds représentent des objets, c'est à dire des blocs de mémoire. Les flèches représentent des pointeurs. Les ronds avec un trait double représentent les objets racines, c'est à dire les premiers objets créés. Ce sont les points d'entrée dans le graphe des objets.

Le principe du garbage collector par coloration est le suivant:

- chaque objet est coloré en blanc, gris ou noir.
- au départ, tous les objets sont blancs.
- le garbage collector parcourt le graphe des objets en largeur en partant des racines (Breadth First Search).
- lorsque un objet blanc est atteint, il passe en noir et tous les objets blancs qu'il pointe passent en gris.
- à la fin de ce parcourt, les objets encore blancs indiquent qu'ils ne sont plus référencés: il peuvent donc être détruits.

Le schéma de la page suivante illustre ce principe.



3.3.2. Structures de données principales

```
typedef struct _gcInfo {
        unsigned int            size;
        unsigned int            ref;
        struct _gcInfo*         prev;
        struct _gcInfo*         next;
        gcFuncs*                funcs;
        char                    colour;
        char                    final;
        char                    __align__[2];
} gcInfo;
typedef gcInfo gcList;

#define  GC_WHITE               0           /* Unmarked */
#define  GC_GREY                1           /* Marked but not traversed */
#define  GC_BLACK               2           /* Marked and traversed */

#define  GC_NORMAL              0           /* Has no finalise method */
#define  GC_FINALIZED           GC_NORMAL   /* Has been finalised */
#define  GC_ROOT                1           /* A root object */
#define  GC_NEEDFINALIZE        2           /* Needs finalising */

static gcList white;
static gcList grey;
static gcList black;
static gcList root;
static gcList finalise;
```

3.3.3. Initialisation

L'initialisation du garbage collector est faite au lancement de Kaffe.

kaffe/kaffe/gc-incremental.c:

➥ void initGc() (appelée depuis main())

3.3.4. Fonctionnement

Les fonctions opérationnelles du garbage collector sont regroupées dans le fichier gc-incremental.c.

kaffe/kaffe/gc-incremental.c:

➥ void* gc_malloc(size_t size, gcFuncs* funcs)
allocation de la mémoire pour un nouvel objet

➥ void soft_addreference(void* f, void* t)
un pointeur a été établi depuis un objet f vers un objet t: si f est black, il faut passer t en grey

➥ void markObject(gcInfo* obj)
passe un objet en grey s'il est white

➥ void startGC()
démarre une boucle de garbage collector

➥ void finishGC()
termine une boucle de garbage collector

➥ void gcMan()
boucle infinie de garbage collection

➥ void finaliserMan()
boucle infinie de destruction des objets collectés

L'ensemble des objets existants à un instant donné est réparti dans cinq listes:
- *root* les racines de persistance (ce sont les premiers objets créés au lancement de Kaffe)
- *finalise* les objets à détruire
- *black* les objets déjà parcourus par le garbage collector
- *grey* les objets atteints par le garbage collector mais non encore parcourus (c'est-à dire les objets pointés par un objet de *black*)
- *white* tous les autres objets

La destruction d'un objet se fait en deux étapes, chacune étant effectuée par un thread distinct:
- le *gcMan* identifie les objets à détruire et les place dans la liste *finalise*
- le *finaliserMan* détruit les objets de la liste *finalise*

Tout nouvel objet créé est placé soit dans la liste *root*, soit dans la liste *white*. Chaque boucle du *gcMan* se déroule de la façon suivante:
1. Au départ, tous les objets sont soit *white*, soit *root*.
2. Tous les objets de *root* sont parcourus:
   ➥ parcourir un objet signifie: passer un *grey* tous les objets *white* qu'il pointe, puis passer l'objet lui-même en *black*.
3. Les objets de *grey* sont parcourus, jusqu'à ce que la liste *grey* soit vide.
4. Tous les objets *white* qui restent sont isolés: ils sont transférés dans la liste *finalise* pour être détruits ultérieurement.

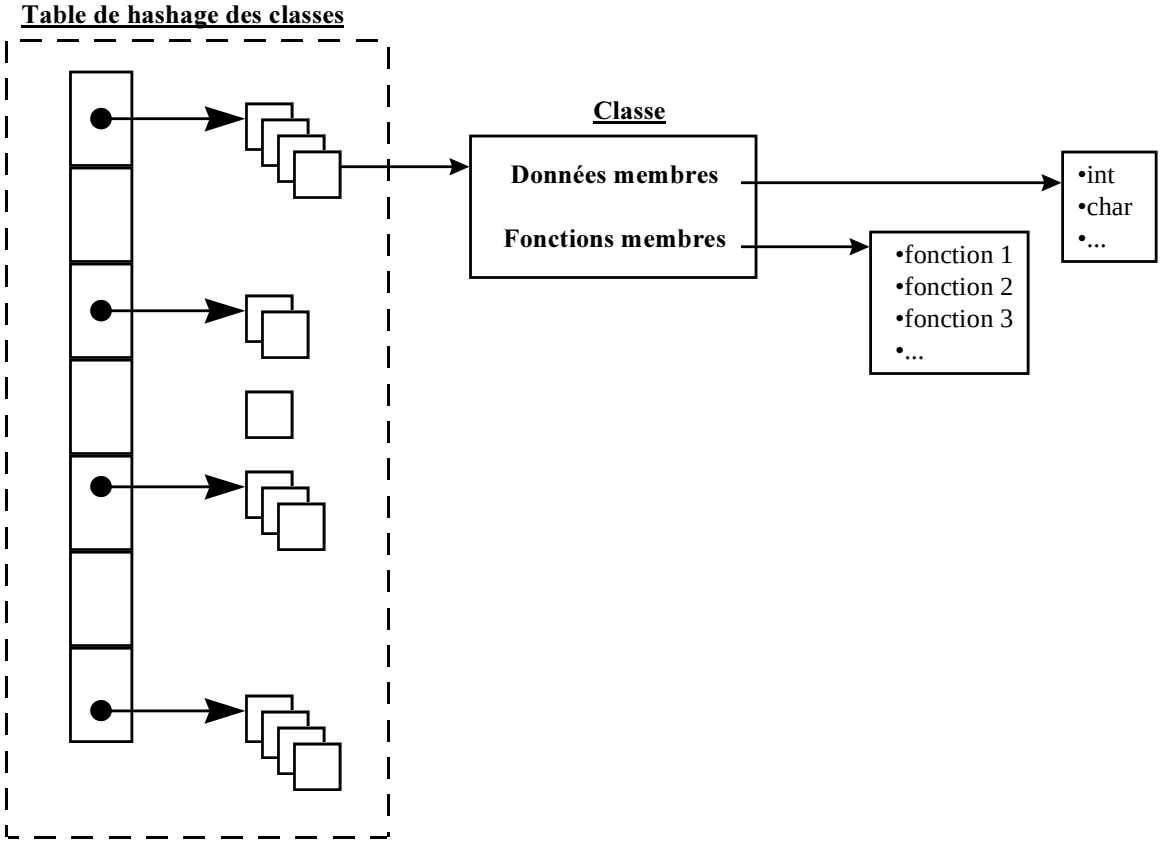
3.4. Gestionnaire de classes

3.4.1. Principe général

Java est un langage purement objet. Toutes les fonctions sont des fonctions membres (ou méthodes) d'une classe. Cet aspect diffère d'un langage objet de type C++ où il est possible de définir des fonctions non membres d'une classe. Ainsi, en Java, tout le code est contenu dans les classes. Une classe est composée de:

- Un nom de classe.
- Des données membres (types élémentaires ou pointeurs sur d'autres objets).
- Des fonctions membres.

Java est un langage interprété. Dans les langages compilés, les fonctions sont repérées par une adresse mémoire relative. Cette adresse est calculée lors de la phase de Link. Au lancement du programme, le code binaire est intégralement chargé en mémoire. Un appel de fonction se résume alors à un saut du compteur de programme. En Java, il n'y a pas de phase de Link. Lors de l'exécution d'un programme, l'interpréteur recherche les classes au fur et à mesure de leur utilisation. Les classes avec leurs données et fonctions membres sont stockées dans une table de hachage.



Au lancement d'un programme Java, que ce soit une application, uneapplet, ou autre, seul le nom de la classe d'entrée est spécifié. L'interpréteur charge et analyse la classe d'entrée. En faisant cela, l'interpréteur va voir que le code de la classe d'entrée fait appel à d'autres classes, qu'il chargera lorsque nécessaire, et ainsi de suite. Les classes chargées sont stockées dans la table de hashage des classes.

3.4.2. Structures de données principales

```
typedef struct Hjava_lang_Class {
        Hjava_lang_Object       head;           /* A class is an object too */

        /* Used to chain classes in the same hash bucket of classPool. */
        struct Hjava_lang_Class* next;

        Utf8Const*              name;
        accessFlags             accflags;

        /* If non-NULL, a pointer to the superclass.
         * However, if state < CSTATE_DOING_PREPARE, then
         * (int) superclass is a constant pool index. */
        struct Hjava_lang_Class* superclass;

        struct _constants  constants;
        /* For regular classes, an array of the methods defined in this class.
           For array types, used for CLASS_ELEMENT_TYPE.
           For primitive types, used by CLASS_ARRAY_CACHE. */
        Method*                         methods;
        short                   nmethods;

        /* Number of methods in the dtable. */
        /* If CLASS_IS_PRIMITIVE, then the CLASS_PRIM_SIG. */
        short                   msize;

        /* Pointer to array of Fields, on for each field.
           Static fields come first. */
        Field*                  fields;

        /* The size of the non-static fields, in bytes.
           For a primitive type, the length in bytes.
           Also used temporarily while reading fields.  */
        int                     bfsize;

        /* Number of fields, including static fields. */
        short                   nfields;
        /* Number of static fields. */
        short                   nsfields;

        /* The data for the static fields. */
        void*                   staticData;

        struct _dispatchTable*  dtable;

      struct Hjava_lang_Class** interfaces;
        int                     interface_len;
        struct _classLoader*    loader;
        int                     state;
        bool                    final;
} Hjava_lang_Class;

static Hjava_lang_Class* classPool[CLASSHASHSZ];
```

3.4.3. Fonctionnement

Les classes chargées sont stockées dans une table de hashage, ce qui permet un accès et une recherche rapides.

kaffe/kaffe/classMethod.c:

➥ int hashClassName(Utf8Const *name)
fonction de hashage sur le nom des classes

➥ Hjava_lang_Class* internalAddClass(Hjava_lang_Class* cl, Utf8Const* name, int flags, int su, struct _classLoader* loader)
ajoute une nouvelle classe dans la table de hashage

➥ Class* simpleLookupClass(Utf8Const* name, struct _classLoader* loader)
recherche une classe dans la table de hashage

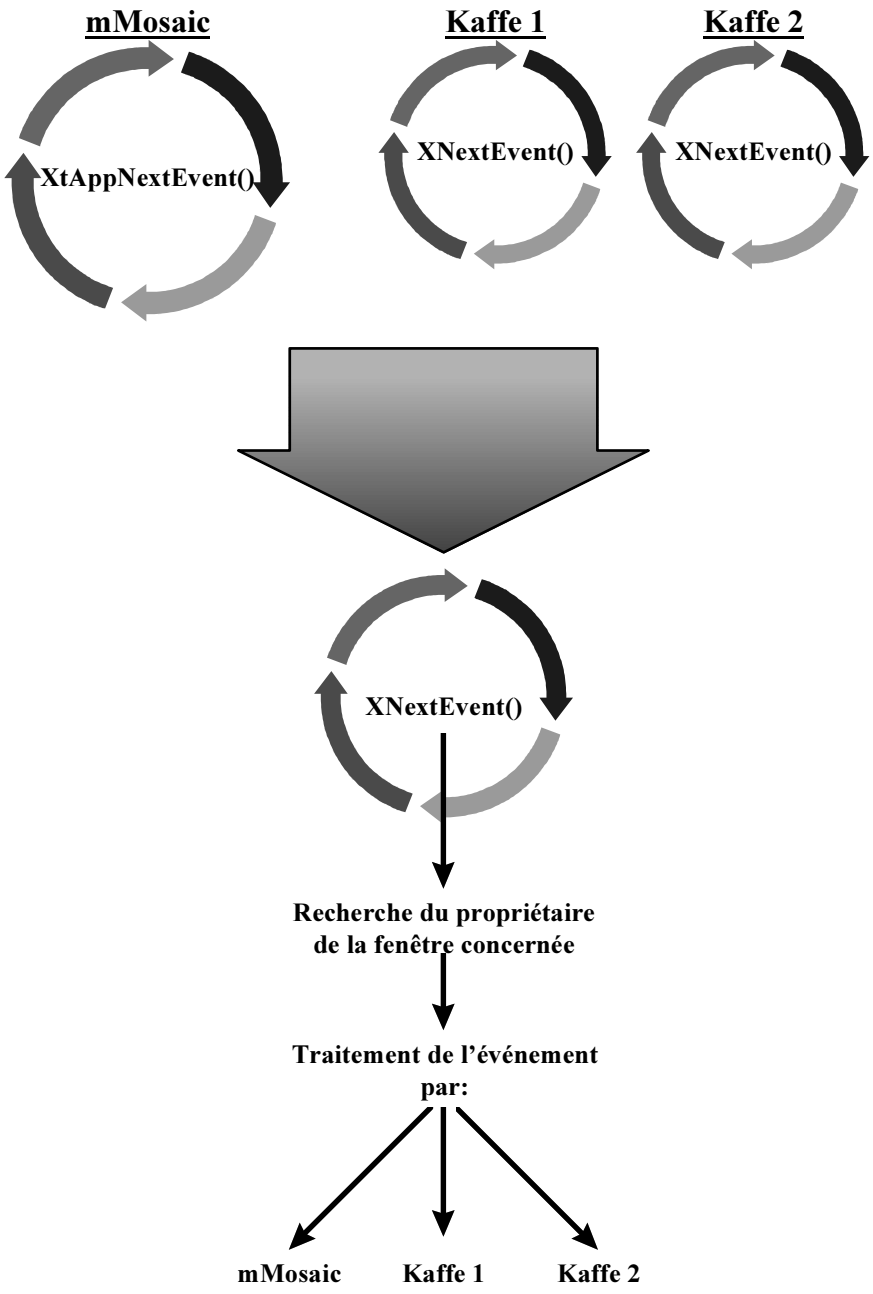
➥ Class* loadClass(Utf8Const* name, struct _classLoader* loader)
recherche une classe, et la charge si elle n'est pas présente

4. Solutions proposées

Les solutions proposées découlent logiquement de l'analyse du chapitre précédent.

4.1. Système graphique

Le travail consiste à intégrer les boucles d'événements de mMosaic et des différentes instances de Kaffe en une seule. Un exemple de cette intégration est réalisé dans la maquette associée au présent mémoire.



4.2. Gestionnaire de threads

Le gestionnaire de threads ne semble pas nécessiter de modification, si ce n'est dans sa phase d'initialisation qui ne doit bien sûr être effectuée qu'une fois. Cette modification est facile à réaliser puisque le fonction d'initialisation est appelée depuis la fonction main() de Kaffe.

4.3. Garbage collector

De la même manière, le garbage collector ne semble pas nécessiter de modification, si ce n'est dans sa phase d'initialisation qui ne doit bien sûr être effectuée qu'une fois. Cette modification est facile à réaliser puisque le fonction d'initialisation est appelée depuis la fonction main() de Kaffe.

4.4. Gestionnaire de classes

De même, l'initialisation du gestionnaire de classe ne doit être effectué qu'une fois. Une autre modification concerne l'heuristique de recherche des classes Java: Kaffe ne recherche que dans le CLASSPATH local au disque dur. Il faut que cette recherche soit étendue au serveur Web d'origine de l'applet. La fonction principale à modifier sera *loadClass* dans le fichier classMethod.c.

4.5. Reconnaissance du tag <Applet> par mMosaic

Cette tâche a déjà été réalisée par Gilles Dauphin lors des premiers essais d'intégration de Kaffe dans mMosaic.

4.6. AppletViewer

Les applets Java sont toujours exécutées à l'aide de la classe Java AppletViewer. La classe AppletViewer de Biss-AWT entoure l'applet proprement dite par des menus, status bar et autres gris-gris (cf maquette présentée). Ces atouts graphiques n'ont pas lieu d'apparaître dans une browser Web. La classe AppletViewer doit donc être élaguée.

5. Maquette

La dernière étape de ce mémoire a été la réalisation d'une maquette. L'objectif de cette maquette est de démontrer et valider les solutions techniques retenues.

Le sujet de la maquette est:

1. Réaliser une application X-Window réduite.
2. Intégrer Kaffe dans cette application.

Le thème de la maquette s'oriente sur les horloge. En effet, une horloge est par nature toujours animée. Ainsi, démonstration est faite que les applications X-Window et Kaffe sont 'vivantes' toutes les deux !!!

L'application X-Window présente une horloge digitale. L'applet Java présente une horloge à aiguilles.

5.1. Modifications effectuées sur Kaffe

Les modifications qui ont dû être effectuées sur Kaffe sont localisées sur quatre fichiers:

- kaffe/kaffe/main.c
- packages/biss-net.com/APIawt/lib/awt.h
- packages/biss-net.com/APIawt/lib/display.c
- packages/biss-net.com/APIawt/lib/window.c

kaffe/kaffe/main.c contient la fonction main() de Kaffe. Cette fonction a été renommée en kaffeMain() pour ne pas la confondre avec la fonction main() de l'application X. kaffeMain() est ensuite appelée depuis l'application X.

L'essentiel des modifications concerne la fonction getNextEvent() de display.c. Cette fonction redirige maintenant les événements appropriés sur l'application X. Un petit détail à noter est l'appel successif des fonctions XNextEvent() et XtAppNextEvent().XNextEvent() est une fonction de la XLib. XtAppNextEvent() est son correspondant Motif. Il se trouve que XNextEvent() ne renvoie pas tous les événements Motif. D'où la nécessité d'appeler les deux fonctions.

Le code ci-dessous présente les changements dans getNextEvent():

```
#ifdef MOSAIC_RELEASE

   if ( nEvents == 0 )
    {
          nEvents = XEventsQueued( dsp, QueuedAfterFlush);
          if (!nEvents)
            {
              if(mMosaicGetEvent(&evt) == 0)
                {
                      unhand(lib)->EventsAvailable = 0;
                      UNLOCK_RETURN( -1);
                }
               else
                {
                      unhand(lib)->EventsAvailable = 0;
                      nEvents = 0;
                }
            }
          else
            {
              XNextEvent(dsp, &evt);
              unhand(lib)->EventsAvailable = --nEvents;
            }
    }
   else
    {
          XNextEvent( dsp, &evt);
          unhand(lib)->EventsAvailable = --nEvents;
    }

   if ( (peerIdx = lookupPeer( lib, evt.xany.window)) < 0 )
    {
          /* this event might be for mMosaic */
          mMosaicXLoop(&evt);
          continue;
    }

#else

   if ( nEvents == 0 ) {
     if ( !(nEvents = XEventsQueued( dsp, QueuedAfterFlush)) ){
          unhand(lib)->EventsAvailable = 0;
          UNLOCK_RETURN( -1);
    }
   }
   XNextEvent( dsp, &evt);
   unhand(lib)->EventsAvailable = --nEvents;
   if ( (peerIdx = lookupPeer( lib, evt.xany.window)) < 0 ) {
     continue;
   }

#endif
```

5.2. Application X

Le code central de l'application X réalisée est présenté ici à titre d'illustration:

```
/*****************************************
**
** Basic Motif Application
**
*****************************************/

#include <Xm/Xm.h>
#include <Xm/DrawingA.h>
#include <Xm/Label.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>
#include <Xm/BulletinB.h>

#include <ClockLabel.h>

/*****************************************
 ** Prototypes and Globals
 *****************************************/

/* intern */
void MyXtAppMainLoop();
void pushKaffe(Widget w, XtPointer client_data, XtPointer callData);

extern Window mosaicKaffeWindow;

static XtAppContext mosaicAppContext;
static Widget box, kaffeWind, clock;
static int argc;
static char **argv;
static char kaffeStarted;

/* extern */
int kaffeMain(int argc, char* argv[]);

/*****************************************
 ** Main
 *****************************************/

main(int argc1, char **argv1)
{
  Widget topWindow, button;
  int argcnt ;
  Arg arg[10];

  kaffeStarted = 0;
  argc = argc1;
  argv = argv1;
  topWindow = XtAppInitialize(&mosaicAppContext, "mMosaicKaffeDemo", NULL, 0, &argc, argv,
NULL, NULL, 0);

  /* create other widgets */
  argcnt = 0;
```

```
  XtSetArg(arg[argcnt], XmNwidth, 300); argcnt++;
  XtSetArg(arg[argcnt], XmNheight, 300); argcnt++;
  box = XmCreateBulletinBoard(topWindow, "box", arg, argcnt);
  XtManageChild(box);

  argcnt = 0;
  XtSetArg(arg[argcnt], XmNx, 100); argcnt++;
  XtSetArg(arg[argcnt], XmNy, 30); argcnt++;
  button = XmCreatePushButton(box, "Start Applet", arg, argcnt);
  XtAddCallback(button, XmNactivateCallback, pushKaffe, NULL);
  XtManageChild(button);

  clock = XtCreateWidget("Clock", xgClockLabelWidgetClass, box, NULL, 0);
  XtManageChild(clock);

  /* kaffe window */
  argcnt = 0;
  XtSetArg(arg[argcnt], XmNx, 10); argcnt++;
  XtSetArg(arg[argcnt], XmNy, 60); argcnt++;
  XtSetArg(arg[argcnt], XmNwidth, 250); argcnt++;
  XtSetArg(arg[argcnt], XmNheight, 200); argcnt++;

  kaffeWind = XmCreateLabel(box,
                            "the APPLET is not running",
                            arg, argcnt);
  XtManageChild(kaffeWind);

  /* realize widgets */
  XtRealizeWidget(topWindow);

  /* start main loop */
  MyXtAppMainLoop();
}

/*******************************************
 ** startKaffe
 *******************************************/

void startKaffe()
{
  int argcnt ;
  Arg arg[10];
  XWindowAttributes xwa;
  XSetWindowAttributes xswa;

  XFlush(XtDisplay(box));
  XGetWindowAttributes(XtDisplay(kaffeWind),
                       XtWindow(kaffeWind),&xwa);

  xswa.do_not_propagate_mask = xwa.do_not_propagate_mask & (~(PointerMotionMask));
  XChangeWindowAttributes(XtDisplay(kaffeWind),
                          XtWindow(kaffeWind),CWDontPropagate, &xswa);
  XFlush(XtDisplay(box));

  /* get display */
  mosaicKaffeWindow = XtWindow(kaffeWind);

  /* initialize kaffe */
  kaffeMain(argc, argv);
```

```
}

/*******************************************
 ** pushKaffe (button callback)
 *******************************************/

void pushKaffe(Widget w, XtPointer client_data, XtPointer callData)
{
  if(kaffeStarted == 0)
    {
      kaffeStarted = 1;
      startKaffe();
    }
}

/*******************************************
 ** MyXtAppMainLoop (Motif main loop)
 *******************************************/

int mMosaicGetEvent(XEvent* evt)
{
  if(XtAppPending(mosaicAppContext))
    {
      XtAppNextEvent(mosaicAppContext, evt);
      return 1;
    }
  else
    {
      return 0;
    }
}

void mMosaicXLoop(XEvent *evt)
{
  XtDispatchEvent(evt);
}

void MyXtAppMainLoop()
{
  XEvent event;

  for(;;) /* forever */
    {
      /* get event */
      XtAppNextEvent(mosaicAppContext, &event);

      /* dispatch event */
      XtDispatchEvent(&event);
    }
}

/*******************************************
 ** The End...
 *******************************************/
```

6. Conclusion

La réalisation de ce mémoire de fin d'études s'est relativement bien passée. L'essentiel du travail a été la découverte et la compréhension détaillée du fonctionnement de Kaffe. Contrairement à d'autres projets du domaine public, le projet Kaffe n'est absolument pas documenté. Pas un seul document sur l'architecture ou le fonctionnement, ni de commentaires dans le code. De plus, le code C de Kaffe est très souvent un apanage d'astuces et de hacks. Grâce à ces difficultés, cette phase de découverte s'est avérée passionnante et pas du tout fastidieuse. C'est une sorte de jeu de chercher à décortiquer un tel code.

Sur le plan personnel, la découverte d'un logiciel aussi complexe et technique que peut l'être un compilateur/interpréteur Java est toujours enrichissante. J'espère voir très bientôt Kaffe tourner dans mMosaic...

La seule petite déception portera sur Kaffe lui-même. Ce produit nous avait semblé au départ, à Gilles Dauphin et moi même, de très bonne facture. Après quelques mois de travail dessus, cet avis a beaucoup évolué. Kaffe, dans sa version actuelle 0.8.4, reste un produit peu mature, instable et énormément buggé. La compilation et l'installation de Kaffe sur une machine relève souvent du challenge ou du miracle. J'espère que cette situation évoluera dans le bon sens dans les prochains mois...