Université de Montréal
Cours IFT6802 – H2003

Les Aglets d'IBM

BETTAHAR Aoued

BETA08036508

Mobilité

Faculté de se déplacer dans un environnement.

Agent Mobile :

Agent capable de se déplacer d'une machine à une autre dans un réseau.

La mobilité peut être forte ou faible en fonction des éléments impliqués dans le processus de transfert (code, données, pile, tas, compteur, ..etc)

Pourquoi les Agents Mobiles:

Les principaux avantages sont :

Réduction de la charge Réseau :

Il est généralement plus avantageux, en terme de performances, de faire voyager du code plutôt que des informations.

Déplacer le Code vers les données:

Les serveurs contenant des données procurent un ensemble fixe d'opérations. Un agent peut étendre cet ensemble pour les besoins particuliers d'un traitement.

Plus sur et meilleure tolérance:

La vie d'un programme classique est lié à la machine ou il s'exécute. Un agent mobile peut se déplacer pour éviter une erreur matérielle ou logicielle ou tout simplement un arrêt de la machine.

Domaines d'Application:

- Commerce électronique
- Applications avec des données réparties
- Télémaintenance / administration de parc

Aglets

Définition :

Les aglets (**Ag**ents App**lets)** sont des objets Java mobiles qui peuvent se déplacer d'une machine à une autre. Ainsi, un aglet qui s'exécute sur un hôte peut stopper son exécution, se déporter vers un hôte distant et continuer cette exécution dans son nouvel environnement.

Histoire :

L'API Aglet est une norme basée sur java proposée pour développer des agents mobiles. Elle a été développée par une équipe de chercheurs du laboratoire de recherche d'IBM à Tokyo au début 1995; son but est de fournir une plate-forme uniforme pour les agents mobiles dans un environnement hétérogène tel que celui de l'Internet

Architecture

Les principaux éléments sont:

Aglet :
Objet mobile de Java qui visite les serveurs où les agents sont autorisés, dans un réseau informatique.
Un aglet est autonome puisqu'il peut reprendre son exécution dès son arrivée à destination et réactif car il peut répondre (réagir) à des événements de son environnement.

Proxy :
Un proxy est un représentant d'un aglet. Il sert de bouclier à l'aglet contre l'accès direct à ses méthodes publiques. Le proxy fournit également la transparence à l'emplacement pour l'aglet. C'est-à-dire qu'il peut cacher le vrai emplacement de l'aglet.

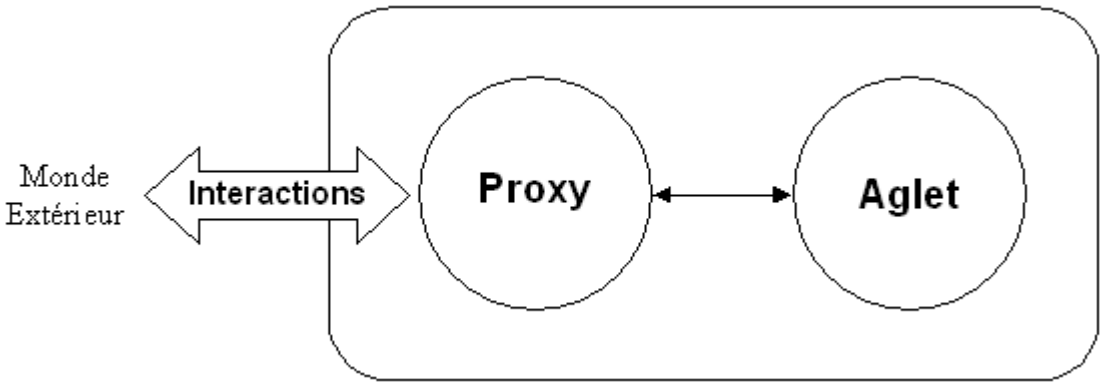


Fig. 1: Relation entre un Aglet et son Proxy.

Contexte :
Le contexte est l'environnement d'exécution de l'aglet. Il fournit des moyens pour mettre à jour et contrôler des aglets dans un environnement uniforme d'exécution.

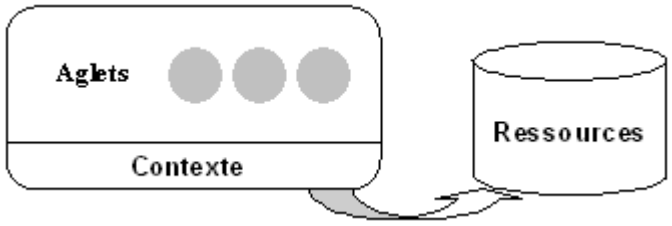


Fig. 2: Évolution des Aglets dans un Contexte.

Hôte ***:***
Un hôte est une machine capable d'héberger plusieurs contextes. L'hôte est généralement un nœud dans un réseau.

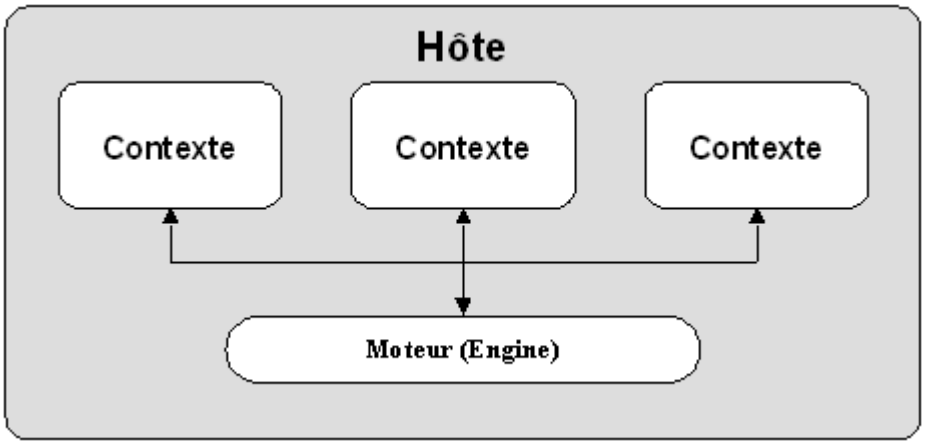


Fig. 3: Relations entre hôte, moteur et contexte

Cycle de Vie

Les types de comportement des Aglets ont été implémentés de manière à répondre aux principaux besoins des agents mobiles.

Les principales opérations affectant la vie d'un aglet sont :

- **Création**
  Se fait dans un Contexte. Un Identifiant unique est assigné. L'initialisation et l'exécution de l'aglet commence immédiatement.

- **Clonage**
  Création d'un clone dans le même contexte que l'original. Un Identifiant différent est alors attribué. A noter que les processus (thread) ne peuvent pas être clonés.

- **Déportation** (Dispatching)
  Transfert d'un aglet d'un contexte à un autre. On dit que l'aglet à été *poussé* vers son nouveau context.

- **Récupération** (Retractation)
  L'aglet déporté est récupéré (*tiré*) dans son contexte d'origine.

- **Activation et Désactivation**
  La désactivation d'un aglet une interruption temporaire de son exécution et stockage de son état dans un support secondaire de stockage.

- **Libération ou destruction**
  Fin de vie de l'aglet et son retrait du context.

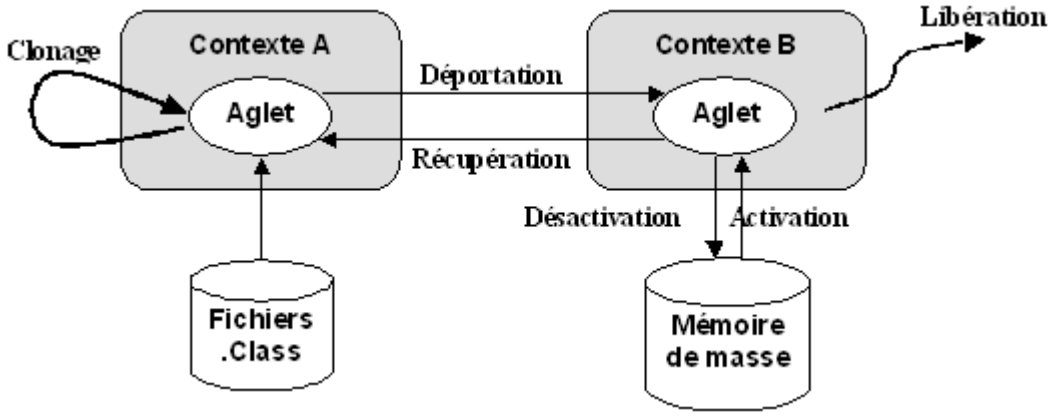


Fig. 4: Modèle du Cycle de Vie d'un Aglet

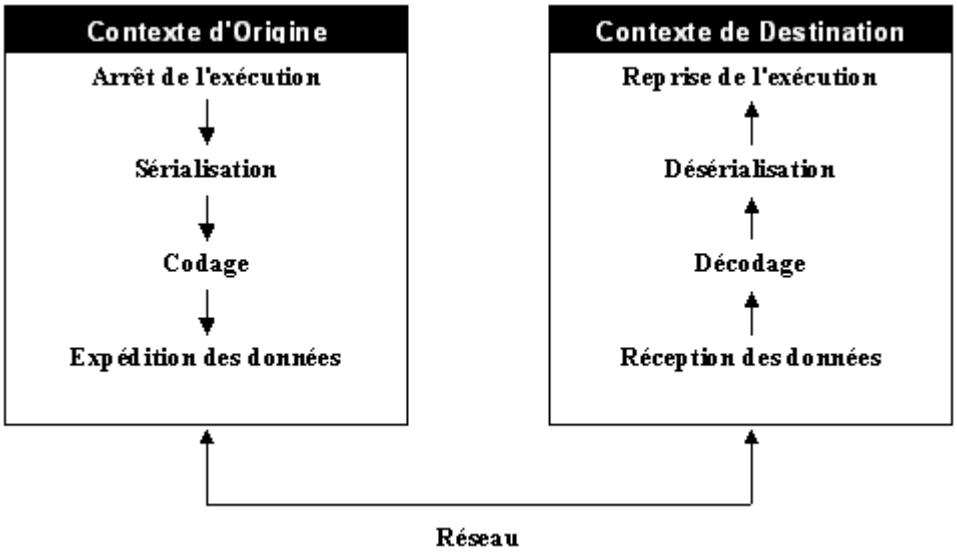


Fig. 4: Transfert d'un Aglet

La Création de l'Aglet

La création d'un aglet se fait par l'appel de la méthode **createAglet()** du contexte.

```
Public abstract AgletProxy createAglet( URL codeBase,
                                        String code,
                                        Object init)
```

Cette méthode retourne le proxy associé à l'aglet nouvellement crée.

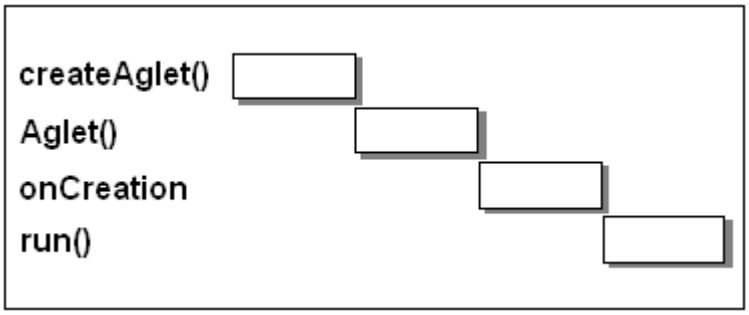


Fig. 5: Diagramme de Collaboration pour la Création de l'Aglet

L'exemple suivant montre comment un aglet peut créer un autre aglet :

```
import aglet.*;
public class CreationExample extends Aglet {
 public void run() {
  try {
   getAgletContext().createAglet(getCodeBase(), "CreationChild", null);
      }
  catch (Exception e) {System.out.println(e.getMessage());}
 }
}
```

Le modèle événementiel de l'aglet

La programmation des aglets est basée sur la gestion des événements. Des notifications alertent l'aglet sur les actions influant sur son cycle de vie. Le programmeur à la possibilité d'implémenter des écouteurs (*listeners*) pour faire réagir l'aglet.
Il existe trois catégories de notifications:

Notifications de clonage

Permettent la capture des événements liés au clonage de l'aglet. Cette fonctionnalité est obtenue par implémentation de l'écouteur (interface) **CloneListener**:

La méthode **clone()** permet de cloner un aglet. Elle retourne le proxy associé au clone. L'aglet qui fait l'objet du clonage reçoit alors les notifications suivantes :

```
Public void CloneAdapter.onCloning(CloneEvent event)
```

Annonce de clonage de l'aglet. La surcharge de cette méthode donne la possibilité d'exécuter les actions en conséquence.

Le clonage est réalisé et nous obtenons deux aglets : Un original est un clone.

L'original reçoit un événement de type :

```
Public void CloneAdapter.onCloned(CloneEvent event)
```

Et le clone reçoit l'événement :

```
Public void CloneAdapter.onClone(CloneEvent event)
```

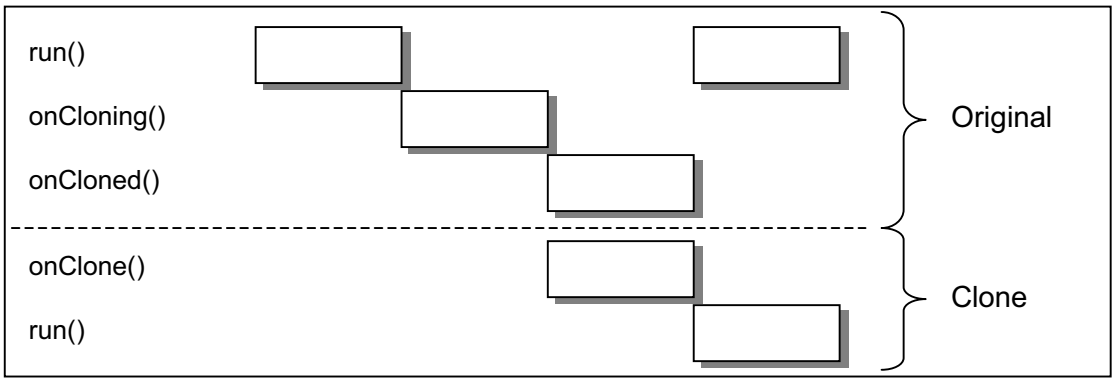


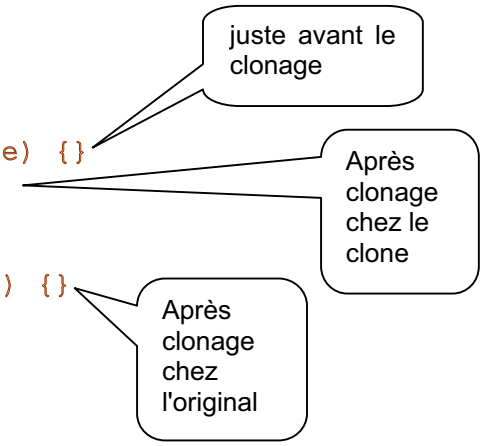
Fig. 6: Diagramme de Collaboration pour le clonage de l'Aglet

Exemple de clonage :

```
public class CloningExample extends Aglet {
 boolean theClone = false;
 public void onCreation(Object o) {

 addCloneListener(
  new CloneAdapter() {
   public void onCloning(CloneEvent e) {}
   public void onClone(CloneEvent e)
    {theClone = true;}

   public void onCloned(CloneEvent e) {}
  });
 }
 public void run() {
  if(!theClone) {
   try {clone();}
   catch(Exception e ) { … }}
  else { … }
 }
}
```



Notifications de mobilité

Permettent la capture des événements liés au clonage de l'aglet. Cette fonctionnalité est obtenue par implémentation de l'écouteur (interface) **MobilityListener**:

Dispatching

La méthode **dispatch(URL destination)** permet de déporter un aglet vers un autre contexte. Si appliquée au proxy, cette méthode retourne le nouveau proxy associé a l'aglet. Ce dernier reçoit alors les notifications suivantes :

`Public void MobilityAdapter.onDispatching(MobilityEvent event)`
Annonce la déportation prochaine de l'aglet. La surcharge de cette méthode donne la possibilité d'exécuter les actions en conséquence.

`Public void MobilityAdapter.onArrival(MobilityEvent event)`
Annonce l'arrivée à destination.

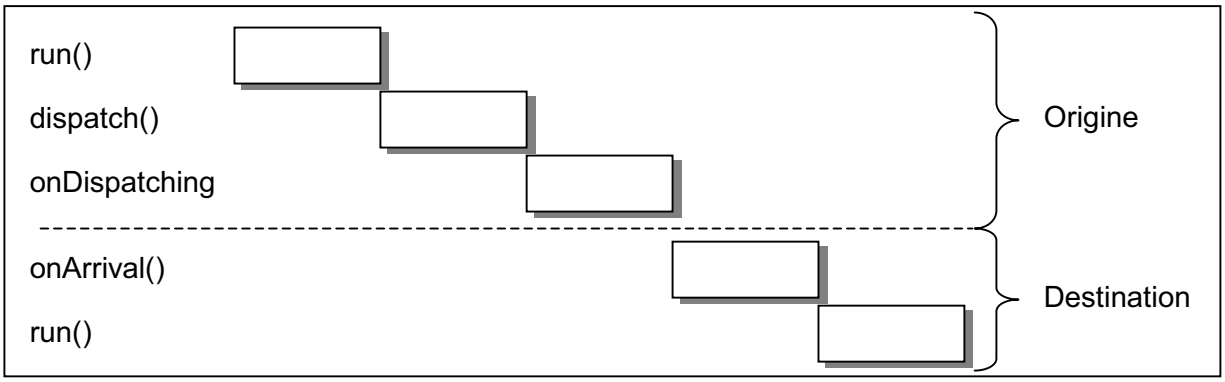


Fig. 7: Diagramme de Collaboration pour la Déportation de l'Aglet

Récupération

La méthode **retractAglet(URL contextAddress,AgletID aid)** permet de ramener un aglet vers son contexte d'origine. Si appliquée au proxy, cette méthode retourne le nouveau proxy associé a l'aglet. Ce dernier reçoit alors les notifications suivantes :

`Public void MobilityAdapter.onReverting(MobilityEvent event)`
Annonce le retour vers le contexte d'origine. La surcharge de cette méthode donne la possibilité d'exécuter les actions en conséquence.

`Public void MobilityAdapter.onArrival(MobilityEvent event)`
Annonce la fin du transfert.

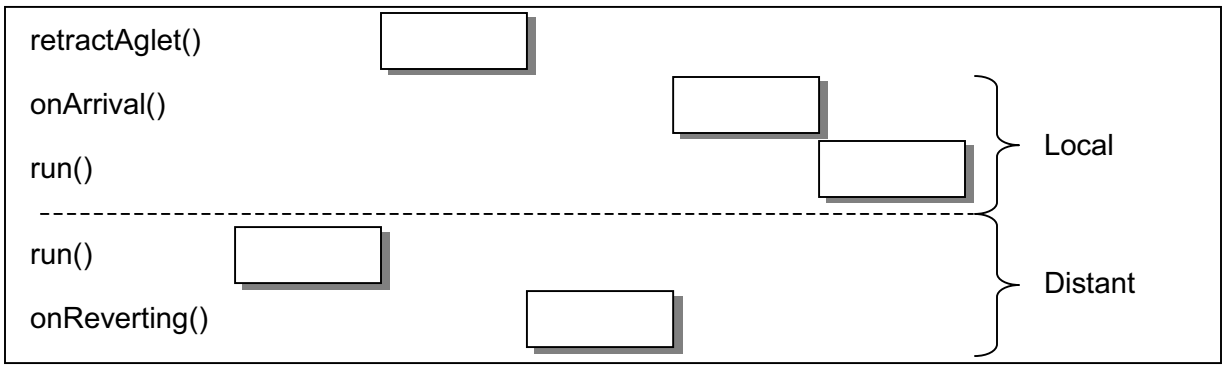


Fig. 8: Diagramme de Collaboration pour la Récupération de l'Aglet

Exemple de Déportation / Récupération :

```
package examples.listingAglet;

import com.ibm.aglet.Aglet;
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import com.ibm.aglet.util.*;

//import com.ibm.agletx.util.SimpleItinerary;

import java.lang.InterruptedException;
import java.io.*;
import java.io.Externalizable;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.io.IOException;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Enumeration;


public class ListingAglet extends Aglet {

  boolean back = false;
  // Un File vers le contenu du repertoire C:\Aglets\lib
  // qu'on veut lister

  File dir = new File("C:/Aglets/lib");
  String[]  list ;
  URL origin = null;
  String DispatchURL = null ;

  public void onCreation(Object o)
  {
    DispatchURL = "atp://localhost:5000";

    addMobilityListener(
        new MobilityListener()
    {

      public void onDispatching(MobilityEvent me){}
      public void onReverting(MobilityEvent me){}
      public void onArrival(MobilityEvent me) {

        if (back) {

          // Imprime sur l'ecran de l'origine le contenu du repertoire
          // C:/Aglets/lib du destinataire
          for (int i = 0; i < list.length; i++)
```

```
          {
            System.out.println(i + ":" + list[i++]);
          }//for

          // Mission accomplie;
          // dispose();
        }//if
        else  {
          try {
            // Obtains directory listing at remote host.
            list = dir.list();
            back = true;

            // Returns to origin.
            dispatch(origin);
          }//try

          catch (Exception e) {
            // Failed to return to origin.
            dispose();

          }//catch
        }//else
      }//mobilityListener
    }//inner class
        );
        try {
          origin =getAgletContext().getHostingURL();
          dispatch(new URL(DispatchURL));
        }//try

        catch (Exception e) {
          // Failed to dispatch aglet;
        }//catch
  }//met
}//class ListingAglet
```

Notifications de Persistance

Permettent la capture des événements liés à la persistance de l'aglet. Cette fonctionnalité est obtenue par implémentation de l'écouteur (interface) **PersistencyListener** :

La méthode **deactivate(long duration)** permet de désactiver un aglet. Il est stocké sur un support de masse. Il ne quitte pas son contexte. L'aglet qui fait l'objet de la désactivation reçoit alors la notification suivante :

```
Public void PersistencyAdapter.onDeactivating(PersistencyEvent event)
```

La surcharge de cette méthode donne la possibilité d'exécuter les actions en conséquence.

La méthode **activate()** permet de réactiver un aglet "endormi". Cela a pour effet de générer la notification suivante :

```
Public void PersistencyAdapter.onActivation(PersistencyEvent event)
```

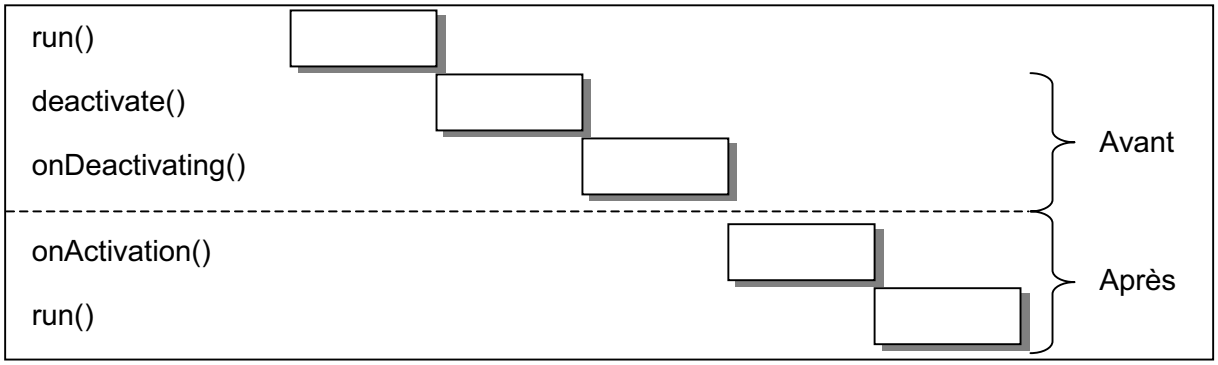


Fig. 8: Diagramme de Collaboration pour la Désactivation / Activation d'un Aglet

L'interface Context

Un aglet peut accéder aux informations relatives à son contexte d'exécution et communiquer avec son environnement grâce à l'interface **AgletContext**.

Le contexte est crée par un serveur d'aglets qui n'est rien d'autre qu'une application permettant de gérer des aglets.

Les principales méthodes offertes par **AgletContext** sont :

```
Public abstract AgletProxy AgletContext.createAglet( URL codebase,
                                                     String code,
                                                     Object init)
```

Le paramètre **init** de type **Object** est transmis à l'aglet nouvellement crée. Ce dernier le récupère dans sa méthode **onCreation(Object init)**.

Exemple :

```
public class InitExample extends Aglet {
 public void run() {
  try {
   getAgletContext().createAglet(getCodeBase(),
                                 "agletbook.InitChild",
                                 "Hello World");
  }
  catch (Exception e) {System.out.println(e.getMessage());}
 }
}
```

```
public class InitChild extends Aglet {
   String text;

   public void onCreation(Object init) {
      text = (String)init;
   }

   public void run() {
      // Print the initilized text field...
      System.out.println("The text: \"" + text + "\"");
   }
}
```

Méthodes d'accès au Proxy:

```
public abstract Enumeration AgletContext.getAgletProxies()
```
ENUMERATION DE TOUS LES PROXY DU CONTEXTE COURANT. LA LISTE ENGLOBE LES PROXY DES AGLETS DESACTIVES.

```
public abstract AgletProxy AgletContext.getAgletProxy(AgletID aid)
```
ACCES AU PROXY D'UN AGLET DONT L'IDENTIFICATEUR **aid** EST CONNU.

```
public abstract AgletProxy AgletContext.getAgletProxy( URL context,
                                                       AgletID aid)
```
ACCES AU PROXY D'UN AGLET DISTANT DONT L'IDENTIFICATEUR **aid** EST CONNU.

Le contexte possède des propriétés accessibles aux aglets. Cet accès est possible en lecture et en modification. Les méthodes sont :

```
public abstract Object AgletContext.getProperty( String key)
```
RETOURNE LA VALEUR ASSOCIEE A LA CLE **key**.

```
public abstract Object AgletContext.getProperty( String key,Object def)
```
RETOURNE L'OBJET ASSOCIE A LA CLE **key**.

```
public abstract Object AgletContext.setProperty( String key,Object obj)
```
AFFECTE L'OBJET **obj** A LA CLE **key**.

Cette fonctionnalité permet aux aglets de laisser des informations (empreintes électroniques) lorsqu'ils parcourent le réseau.

Le Modèle de Communication des Aglets :

La communication est basée sur l'échange d'objets de la classe **Message**. Ce modèle de communication est indépendant de la localisation de l'aglet et peut être asynchrone ou synchrone.

Un aglet désirent envoyer un message doit obligatoirement passer par le proxy du destinataire. En fait, le proxy reste l'intermédiaire obligatoire pour tout échange.

Chaque aglet possède un gestionnaire de messagerie **MessageManager** qui lui permet de les traiter un par un et dans l'ordre de leur arrivées respectives. Toutefois, cet ordre peut être changé par l'aglet en modifiant les priorités des messages dans la file d'attente. Ceci est possible grâce à la méthode **setPriority**.

```
public void MessageManager.setPriority(String kind, int priority)
```

Les priorités ont des valeurs allant de 1 à 10.

D'autres mécanismes existent pour le traitement parallèle des messages.

La classe Message

Chaque message est caractérisé par la catégorie (**kind**) à laquelle il appartient. La création d'une instance message nécessite l'affectation d'une valeur à son paramètre **kind**.

```
Message msg = new Message("hello");
Message msg = new Message("Mon nom", "...");
Message msg = new Message("Age",50);
```

Catégorie kind

mesDonnes est de type Object

ou même

```
Message msg = new Message("Données",mesDonnees);
```

Le constructeur **Message** supporte plusieurs formes:

```
public Message(String kind)
public Message(String kind, Object  arg)
public Message(String kind, int     arg)
public Message(String kind, long    arg)
public Message(String kind, float   arg)
public Message(String kind, double  arg)
public Message(String kind, char    arg)
public Message(String kind, boolean arg)
```

De plus, deux méthodes sont fournies pour organiser le paramètre **arg** en tableaux de valeur :

```
public void Message.setArg(String key, Object value)
public void Message.getArg(String key)
```

Exemple :

```
Message msg = new Message("Position");
msg.setArg("Horisontal",40);
msg.setArg("Vertical",60);
```

Messages synchrones

La méthode **sendMessage(Message msg)** permet l'envoi de messages synchrones. Elle est donc bloquante car elle retourne la réponse du récepteur.

```
public Object AgletProxy.sendMessage(Message message)
```
ENVOI **message** ET ATTEND LA REPONSE.

Le destinataire du message doit implémenter un gestionnaire des messages reçus. Cette fonctionnalité est rendue possible par la surcharge de la méthode **Aglet.handleMessage(Message msg)**.

La méthode **Message.sendReply** permet de retourner une réponse a l'envoyeur. Elle supporte plusieurs constructions :

```
public void Message.sendReply ()
public void Message.sendReply (boolean reply)
public void Message.sendReply (char    reply)
public void Message.sendReply (double  reply)
public void Message.sendReply (float   reply)
public void Message.sendReply (int     reply)
public void Message.sendReply (long    reply)
public void Message.sendReply (Object  reply)
```

Exemple d'échange simple de messages :

```
public class SimpleMessageExample extends Aglet {
  public void run() {
    try {
      AgletProxy  proxy = getAgletContext().createAglet(getCodeBase(),
"SimpleMessageChild",null);
      try {
        proxy.sendMessage(new Message("How are you?") );
      }
      catch (NotHandledException e){} //Child failed to handle message.
    }
    catch (Exception e ) {}           //Failed to create the child.
  }
}

public class simpleMessageChild extends Aglet {
```

```
  public boolean handleMessage(Message msg) {
    if (msg.sameKind("How are you?")  ) {
      msg.sendReply("I'm fine. thanks"); // Respond to message
      return true; }
    else
      return false;   // No, I did not handle this messge.
  }
}
```

Messages asynchrones

La messagerie asynchrone est implémentée grâce à la notion de *futur objet*. L'envoi d'un message asynchrone retourne un lien vers la réponse même si cette dernière n'existe pas encore. Ce lien permet de tester si la réponse est arrivée ou pas. Cette technique permet à l'aglet d'envoyer un message sans être obligé d'interrompre son exécution en attente de la réponse.

```
public FutureReply AgletProxy.sendFutureMessage(Message msg)
```

La classe **FutureReply** offre des méthodes pour la récupération des messages.

```
public boolean FutureReply.isAvailable()
```
Vérifie si la réponse est arrivée ou non.

```
public void FutureReply.waitForReply(long duration)
```
Attente de l'arrivée de la réponse pour la durée duration.

```
public void FutureReply.waitForReply()
```
Blocage en attente de l'arrivée de la réponse.

```
public Object FutureReply.getReply()
```
Récupère la réponse.

Exemples :

```
public class FutureExample extends Aglet {
  public  void  run()  {
    try {
      AgletProxy  proxy = getAgletContext().createAglet(
          getCodeBase(), "FutureChild", null);
       try {
        FutureReply  future = proxy.sendFutureMessage(
            new Message("Please reply") );
        while ( ! future.isAvailable() )
        {
          // doincrement();
        };
        String reply = (String) future.getReply();
      } catch ( NotHandledException e ) {
```

```
        // Failed to send the message.
      }
    } catch ( Exception e ) {
      // Failed to create the child.
    }
  }
}
```

```
public  class  TimeoutExample  extends Aglet {
  public void run() {
    try {
      AgletProxy  proxy = getAgletContext().createAglet(getCodeBase(),
"TimeoutChild", null);
      try {

        FutureReply  future = proxy.sendFutureMessage(new
Message("Please reply") );

        future.waitForReply(4000); // Wait up to 4 seconds.

        if (future.isAvailable()) {
          String reply = (String) future.getReply();
          // Got the reply ...
        }
        else {} // Time out ...  Forget the reply.
      }
      catch (NotHandledException e) {} // Failed to send the message.
    }
    catch (Exception e) {} // Failed to create the child.
  }
}
```

Un autre moyen d'envoyer les messages est d'utiliser la méthode suivante :

```
public void AgletProxy.sendOnewayMessage(Message msg)
```

Elle permet l'envoi de messages sans attente de réponse.

Diffusion de messages (Multicast)

Un aglet peut s'abonner à un ou plusieurs groupes au sein d'un même contexte. Chaque groupe est identifié par une catégorie de message.

```
public final void subscribeMessage(String kind)
```

les autres méthodes sont :

```
public final boolean unsubscribeMessage(String kind)
public final boolean unsubscribeAllMessage()
```

Pour envoyer un message multicast :

```
public class MulticastExample extends Aglet {

  public void run() {
    try {
      getAgletContext().createAglet(getCodeBase(),
                                    "MulticastSubcriber", null);
      // Creates multiple instances of "MulticastSubscriber".
      Message msg = new Message("Hello Everybody");
      getAgletContext().multicastMessage( msg );

     } catch ( Exception e ) {
      // Failed to create a subscriber.
    }
  }
 }
```

Pour la réception des messages :

```
public class ReplySetExample extends Aglet {
 public void run() {
  try {
   getAgletContext().createAglet(getCodeBase(),"ReplySetChild", null);
   // Creates multiple instances of "ReplySetChild".
   try {
    Message msg = new Message("Hello Everybody");
    ReplySet replies = getAgletContext().multicastMessage(msg);

    while ( replies.hasMoreFutureReplies() ) {
      FutureReply future = replies.getNextFutureReply();
      System.out.println((String) future.getReply());
    }
   } catch ( Exception e ) {} // Failed to multicast the message.
  } catch ( Exception e ) {}  // Failed to create a subscriber.
 }
}
```

Les Gabarits de Conception

Ils sont disponibles pour le programmeur, pour lui faciliter la mise en oeuvre de canevas d'interactions, tels que le canevas maître-esclave, client-serveur et notifieur/notifié. Ces gabarits sont disponibles sous la forme de classes réutilisables.

Les Serveurs d'Aglets:

• **Tahiti : un gestionnaire d'agents visuel**. Tahiti utilise une interface graphique unique pour suivre et contrôler l'exécution des aglets. Il est possible en utilisant le glisser-déposer de faire communiquer deux agents ou de les faire migrer vers un site particulier. Tahiti dispose d'un gestionnaire de sécurité paramétrable qui détecte toute opération non autorisée et empêche l'agent de la réaliser.

• **Fiji - un lanceur d'agent sur le Web**. Fiji est un applet Java capable de créer ou de détruire un aglet sur un navigateur Web. Fiji utilise comme unique paramètre l'URL de l'agent, qui est intégré directement dans le code HTML d'une page Web.

Comme pour un applet, l'exécution d'un aglet commencera par le téléchargement du code, puis par son lancement grâce à Fiji. Si le serveur Web est complété par un démon ATP, il devient très facile de répartir des aglets sur les sites Web.

Les Aglets et La sécurité

Dans un réseau en pleine expansion tel que Internet, le problème de la sécurité trouve tout son sens dans l'utilisation des agents. On peut déjà identifier trois cibles possibles :

L'agent :
-L'agent qui visite une machine hôte pourrait être la cible d'une tentative d'extraction d'informations.
-Même scénario avec un agent malveillant.
-Interception de la messagerie.

L'hôte :
-Un agent malveillant pourrait visiter un hôte dans le but d'accéder ou de corrompre ses fichiers.
-Une entité malveillante pourrait envoyer un nombre important d'agents vers un serveur dans le but de le surcharger.

Le réseau :
-Multiplication sans fin d'agent dans le but d'encombrer et de surcharger le réseau.

Les aglets disposent d'un système de sécurité en couches. La première couche est offerte par Java et ses mécanismes de sécurité. La couche suivante est constituée du gestionnaire de sécurité qui permet aux utilisateurs d'implémenter leurs propres mécanismes de protection. La troisième et dernière couche est composée des API de sécurité Java permettant au programmeur d'inclure des fonctionnalités de sécurité dans leur agent.

Comparaison avec les autres systèmes d'agents mobiles :

	Aglets	Grasshopper	Voyager
Architecture and Components	Aglets as agents, Context as host/interface to runtime environment.	Mobile and Stationary Agents. Region, Agency, Place.	Distributed object connected by an ORB. Space for broadcasting.
Transport and Communication Technology	Agent Transfer Protocol (ATP).	Socket, RMI, CORBA, Socket+SSL, RMI+SSL	Socket, CORBA, RMI, DCOM.
Concept of Stationary Agent	No	Yes	No
Control of Migration	Agent itself or others.	Agent itself or others.	Agent itself or others.
Communication Type	Message passing	Method invocation	Method invocation

	Aglets	Grasshopper	Voyager
Synchronous Communication	yes	yes	yes
Asynchronous Communication	yes	yes	yes
One-way Communication	yes	yes	yes
Multicast Communication	yes	yes	yes
Message Forwarding	no	yes	yes
Scope of Directory Service	Host	Region	global

Conclusion

Les aglets offrent une très grande simplicité d'implémentation. Ceci constitue un avantage certain par rapport aux autres systèmes. Toutefois, à notre avis, des mécanismes plus performants pour le recensement des aglets devraient accroître l'efficacité de cette technologie. La localisation d'un aglet reste une tache qui pourrait poser des problèmes surtout dans un environnement à forte fréquence de pannes.

BIBLIOGRAPHIE

Ouvrages :

- Danny Lange and Mitsuru Oshima. "Programming and Deploying Java Mobile Agents With Aglets". Addison-Wesley, 1998.

Liens Internet:

- Site d'IBM Japon : http://www.trl.ibm.com/aglets/index_e.htm
- Aglets : IBM ASDK 2.0 http://dess-gla.infop6.jussieu.fr/~caillett/asdk20/
- IBM Aglets : http://cc.usu.edu/~kirank/Aglets.ppt
- Comparison Study of Three Mobile Agent Systems : http://www.engr.uconn.edu/~steve/Cse298300/Fall02Projs/Agents/APres.ppt
- Distributed Computing with Aglets : http://www.vistabonita.com/papers/DCAglets/DCWithAglets.html
- Agents Mobiles : http://www.ift.ulaval.ca/~kone/Cours/AM/PlanCoursHTML.htm
- The Aglets portal : http://aglets.sourceforge.net/
- Les Agents Mobiles et Actifs pour l'Administration de Réseaux http://www.iufm.unice.fr/~reuter/rapport-dea.pdf
- Les Aglets : http://etna.int-evry.fr/~bernard/sarp/projets97-98/agents-rapport/aglet.htm

ANNEXES

L'API Aglets

com.ibm.aglet	Les classes "fondamentales" telles que com.ibm.aglet.Aglet.
com.ibm.aglet.event	Les évènements liés au cycle de vie et à la mobilité.
com.ibm.aglet.security	La protection des aglets (voir chapître Sécurité).
com.ibm.aglet.system	Aides à la gestion du Contexte d'exécution.
com.ibm.aglets	Aides à l'implémentation d'un serveur d'Aglets.
com.ibm.aglets.security	Aides à la gestion de la sécurité dans un contexte JDK 1.1.
com.ibm.aglets.tahiti	Implémentation du serveur d'Aglets Tahiti.
com.ibm.agletx.patterns	Utilitaires définissant des comportements de haut niveau.
com.ibm.agletx.util	
com.ibm.atp	Protocoles de communication réseau.
com.ibm.atp.auth	
com.ibm.awb.launcher	
com.ibm.awb.misc	
com.ibm.awb.weakref	
com.ibm.maf	
com.ibm.maf.atp	
com.ibm.maf.rmi	
com.ibm.net.protocol.atp	
com.ibm.net.protocol.rmi	
org.aglets.log	Fonctionnalités de journalisation.
org.aglets.log.console	
org.aglets.log.log4j	
org.aglets.log.quiet	