# Distributed Algorithms in Java
## Manuel d'utilisation

http://www.risc.uni-linz.ac.at/software/daj/

Thomas FISHER
Cyril ARACHEQUESNE

13 janvier 2004

**Résumé**

Ce manuel présente l'outil de dévelopement DAJ, servant à implémenter, tester, simuler et visualiser des algorithmes distribués en Java. DAJ est une bibliothèque de classes Java avec une interface de programmation simple permettant de développer des algorithmes distribués basés sur un modèle d'échange de messages. Les programmes résultants peuvent être lancés seuls via l'interpréteur Java, ou inclus en tant qu'applet dans une page HTML, et exécutés par un explorateur. Le but de ce kit de développement est de fournir une plate-forme universelle accessible pour la recherche et l'éducation dans le domaine des algorithmes distribués. DAJ est disponible gratuitement sur internet.

# Table des matières

# Introduction

Une application développée pour DAJ peut être exécutée selon trois **modes** : standalone avec ou sans visualisation ou en tant qu'applet incluse dans une page web.

Le **modèle de programmation** bien que simple intègre un minimum de concepts :

- le modèle d'exécution consiste en un réseau de noeuds partiellement liés entre eux de manière unidirectionnelle et qui opérent indépendamment les uns des autres,
- le mécanisme de la base de la communication est l'échange de messages point à point (i.e. point-to-point messages) entre ces noeuds liés par un canal,
- un noeud peut envoyer/recevoir un message via un ensemble de canaux,
- une limite temporelle peut être spécifiée afin d'éviter qu'un noeud n'attende indéfiniment sur un canal vide (ou sur un ensemble vide de canaux),
- chaque noeud possède une horloge locale qui avance implicitement lors de chaque opération de communication et qui peut être utilisée par le programmeur.

La **visualisation** est basée sur les éléments suivants :

- un écran affiche le réseau tel que mis en place par l'utilisateur,
- les états des noeuds et des canaux sont indiqués par des couleurs et par des messages figurant en bas de la fenêtre,
- affichage de l'horloge locale à chaque noeud,
- affichage de l'état interne de chacun des noeuds et canaux par des pop-ups,
- l'exécution peut être ralentie, interrompue, exécutée pas à pas et redémarrée.

L'exécution d'un programme peut être **modifiée** par :

- la mise en place d'un ordonnanceur des noeuds qui détermine le prochain noeud prêt à être exécuté,
- l'instauration d'un sélecteur de messages qui sélectionne le prochain message que doit délivrer un certain canal.

Ainsi, différents modèles d'exécutions peuvent être implémentés : synchrones, asynchrones, avec duplication et/ou pertes de messages, etc.

De plus, le programmeur peut mettre en place des assertions permettant de vérifier l'intégrité des invariants du système.

# Chapitre 1

# Installation et Utilisation

## 1.1 Installation

La bibliothèque sous forme binaire est disponible gratuitement sous la seule condition de citer son utilisation dans toute application ou publication résultante. Les sources ne sont généralement pas disponibles mais tout bug, ou toute modification éventuelle peut être signalé à l'auteur.

Deux versions de DAJ sont disponibles, l'une pour la JDK 1.0.x et l'autre pour la JDK 1.1.x (les versions supérieures de la JDK ne permettant pas de compiler).

- les téléchargements se font sur :

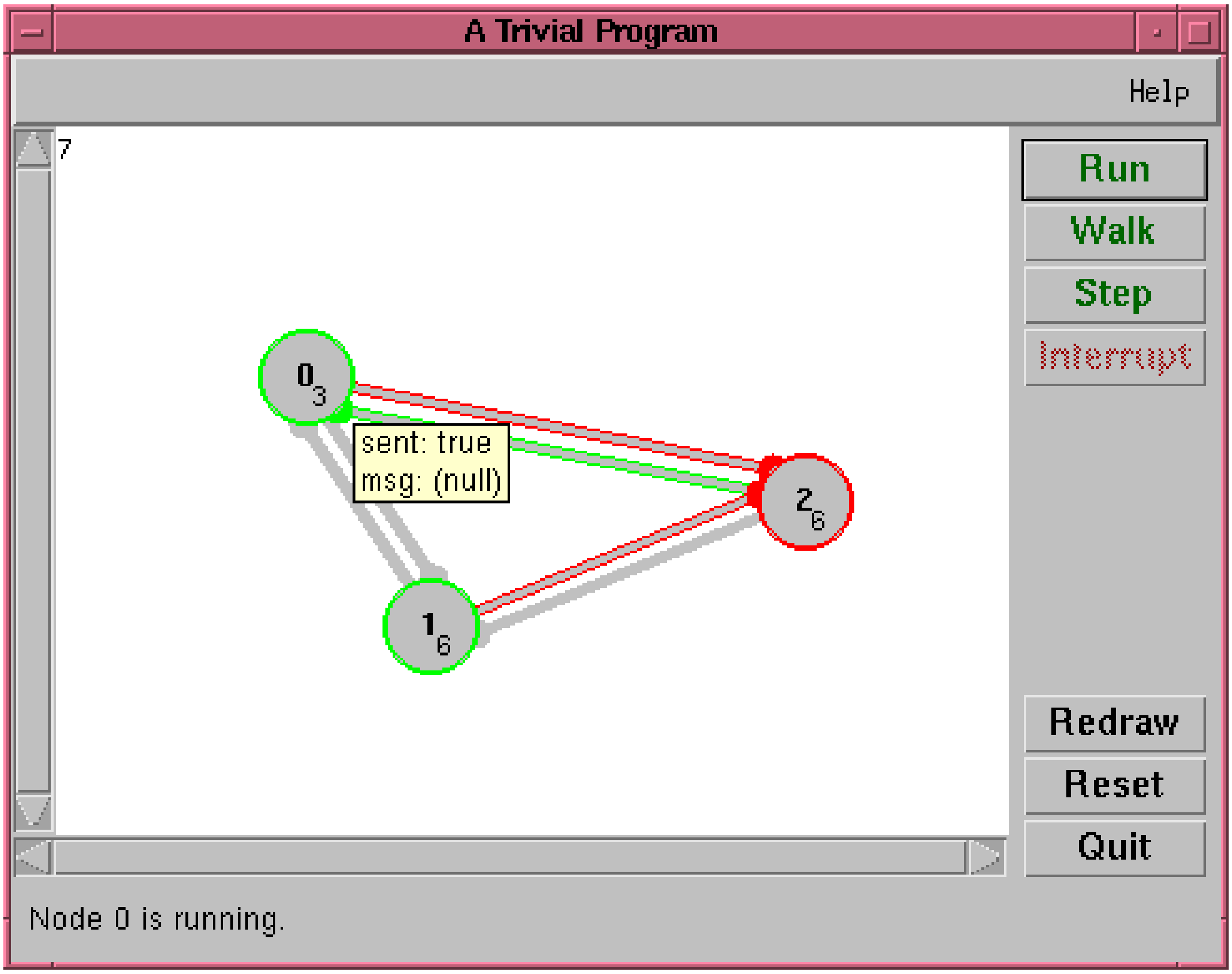
  `http://www.risc.uni-linz.ac.at/software/daj/`

  où il est possible de récupérer le package Daj et awt en version zip (Windows), tar.gz ou encore jar,

- l'archive Daj doit être extraite dans un répertoire *daj* et l'archive awt dans *daj/awt*,

- une fois les bibliothèques copiées et extraites, on peut en spécifier le chemin dans la variable d'environement *CLASSPATH* pour une utilisation locale.

## 1.2 Présentation de la fenêtre graphique




A Trivial Program

FIG. 1.1 – Aperçu de l'exécution d'un algorithme.

La fenêtre présente un réseau composé de noeuds liés partiellement les uns aux autres par des canaux unidirectionnels. Le côté récepteur d'un canal est marqué par un petit cercle collé au noeud.

D'autre part, les noeuds sont numérotés et en-dessous de ces numéros, il figure systématiquement un indice qui représente l'horloge locale à chacun d'entre eux. Enfin, le nombre figurant dans coin supérieur gauche de la fenêtre représente l'horloge globale du réseau.

Un appui sur le bouton **Run** déclenche l'exécution, qui peut être suspendue par **Interrupt**. **Walk** exécute le programme de manière ralentie, et **Step** avance le programme d'un pas (i.e. une opération de communication). **Reset** permet de réinitialiser la visualisation et **Redraw** redessine l'écran, Enfin, **Quit** termine la visualisation (si le réseau est en exécution, il faut d'abord l'interrompre).

Durant l'exécution, l'état actuel de chacun des noeuds est représenté par la couleur de son périmètre :

**vert** : le noeud est prêt à être exécuté,
**rouge** : le noeud est bloqué car il attend un message sur un quelconque canal,
**bleu** : le noeud a terminé son exécution.

De la même manière, l'état de chaque canal est représenté par la couleur de ses frontières :

**gris** : le canal est vide,
**vert** : le canal contient au moins un message,
**rouge** : le canal est vide et le noeud récepteur attend un message.

Des explications textuelles sur les états figurent en bas de la fenêtre lorsque le curseur de la souris est amené sur un noeud ou une arête. Simultanément une fenêtre s'ouvre (i.e. pop-up) affichant des informations internes (file de messages dans le cas d'un canal).

Un noeud peut être déplacé en pressant le bouton gauche de la souris et en le relachant à l'endroit désiré.

La sélection des éléments **About Algorithm** ou **About Toolkit** du menu **Help** lance des pop-up dans lesquelles figurent diverses informations. La sélection de **Home**, **Copying** ou **Help** charge la page Web correspondante dans une fenêtre du navigateur.

# Chapitre 2

# Programmation minimale avec l'outil DAJ

DAJ peut être utilisé simplement en utilisant les structures par défaut. Néanmoins, il faut tout d'abord étendre la classe*Application* qui va permettre de :

- Construire le réseau.
- Modifier l'exécution (optionnel).
- Modifier la visualisation du réseau (optionnel).

## 2.1 La classe *Application*

```
abstract class Application
{
  // construction et exécution de l'application
  Application()
  Application(String t, int w, int h)
  void run()
  String getText()

  // construction du réseau
  abstract void construct()

  // construction des noeuds et des canaux
  Node node(Program p)
  Node node(Program p, String t, int x, int y)
  void link(Node n1, Node n2)
  void link(Node n1, Node n2, Selector s)

  // paramétrage de l'exécution
  void setScheduler(Scheduler s)
  void setSelector(Selector s)
```

```
  // paramétrage de la visualisation
  void setNodeRadius(int r)
  void setNodeFonts(Font f1, Font f2)
  void setChannelWidth(int w)
  void setBrowser(String b)
}
```

### 2.1.1 Construction et exécution de l'application

Le constructeur par défaut, *Application()*, crée une application sans visualisation utilisable en mode standalone uniquement. Le second constructeur, *Application(Stringt, int w, int h)*, crée une application visualisable par une fenêtre nommée $t$ et une zone de visualisation de largeur $w$ et de hauteur $h$.

Si l'application doit être utilisable en mode standalone, le programmeur doit créer un objet de ce type et appeler sa méthode *run()* alors que si l'application doit être utilisée en tant qu'applet, le programmeur doit surcharger son constructeur par défaut pour appeler celui avec le titre, la largeur et la hauteur.

*getText()* est une méthode permettant de retourner une information (par défaut "(no information)") qui peut être affichée par l'appletviewer ou le navigateur dans le menu **Help**.

### 2.1.2 Construction des noeuds et des arêtes du réseau

Le programmeur doit surcharger la méthode *construct()* pour créer et lier les noeuds du réseau.

La construction d'un noeud est effectuée par l'appel de la primitive *node()*. Celle-ci ne peut être appelée que par la méthode *construct()* et elle retourne un noeud exécutant un programme passé en paramètre :

- *Node node(Program p)* utilisable uniquement si l'application n'est pas visualisable,
- *Node node(Program p, String t, int x, int y)* retourne un noeud dont l'étiquette placée aux coordonnées *(x,y)* comporte le texte $t$. Cette primitive est utilisable dans les deux modes.

La création d'un canal est effectuée par l'invocation de la méthode *link()* qui existe aussi sous deux formes :

- *void link(Node n1, Node n2)* crée un canal unidirectionnel point à point de *n1* vers *n2* et la délivration des messages se fait selon le sélecteur par défaut,
- *void link(Node n1, Node n2, Selector s)* a le même comportement que la fonction précédente excepté que le sélecteur $s$ est utilisé pour la délivration des messages.

### 2.1.3 Paramétrage de l'exécution

Modifier l'exécution consiste à redéfinir le sélecteur de messages et/ou l'ordonnanceur en faisant appel respectivement aux méthodes *setSelector()* ou *setScheduler()*. Par ailleurs, ces deux fonctions peuvent uniquement être appelées par le constructeur d'un objet de type *Application*.

Un sélecteur de messages retourne le prochain message qui doit être délivré par un canal donné parmi sa file de messages ; celle-ci est alors mise à jour. Le comportement par défaut est celui d'une FIFO (First In First Out) qui consiste à retourner le plus vieux message, c'est-à-dire le premier envoyé (cf. partie 3.4).

Quant à l'ordonnanceur, il consiste à élire un programme à exécuter sur un noeud. Par défaut celui-ci utilise un mécanisme de tourniquet pour élire le prochain programme prêt à être exécuter (cf. partie 3.5).

### 2.1.4 Paramétrage de la visualisation

La visualisation concerne une zone de l'écran définie par l'objet de type *Application* qui affiche les noeuds et arêtes du réseau (cf. figure 1.1). Divers aspects de la visualisation peuvent être modifiés par les fonctions suivantes qui peuvent être appelées seulement dans le constructeur de l'objet de l'application :

***setNodeRadius(int r)*** redéfinit à $r$ le rayon des cercles représentant les noeuds,
***setNodeFonts(Font f1, Font f2)*** : l'étiquette d'un noeud est affichée avec un indice représentant le temps local à ce noeud. Cette fonction peut être utilisée pour redéfinir le fond de l'étiquette en tant que *f1* et celui de l'indice en tant que *f2*.

***setChannelWidth(int w)*** redéfinit la largeur de la ligne représentant un canal à $w$.
***setBrowser(int b)*** : en mode standalone, un navigateur web est lancé quand l'utilisateur sélectionne l'élément du menu **Home**, **Copying** ou **Help** pour afficher le contenu de la page web correspondante. Cette fonction peut être utilisée pour redéfinir le chemin d'accès au navigateur (par défaut */usr/local/bin/netscape*) à $b$.

## 2.2 La classe *Program*

Un programme qui s'exécute sur un quelconque noeud consiste à :

- accéder à l'interface du noeud,
- communiquer avec les autres noeuds,
- communiquer avec les groupes des autres noeuds,
- contrôler son exécution de différentes autres manières.

Chaque noeud du réseau possède un objet instanciant la classe *Program* ou une de ses classes filles.

```
abstract class Program
{
  // à surcharger
  abstract void main()
  String getText()

  // interface de communication des noeuds
  InChannelSet in()
  InChannel in(int i)
  OutChannelSet out()
  OutChannel out(int i)

  // contrôle de l'exécution d'un programme
  void assert(GlobalAssertion a)
  int getTime()
  void sleep(n)
  void yield()
  void exit(int i)
}
```

Les fonctions de cette classe forment trois ensembles concernant respectivement :

- le programme exécuté par le noeud,
- l'interface avec le noeud,
- le contrôle de l'exécution du programme.

### 2.2.1 Le programme exécuté par le noeud

La méthode *main()* doit être surchargée par le programmeur, elle représente le programme exécuté par ce noeud.

La méthode *getText()* retourne une chaîne de caractères (par défaut "(no information)") qui consiste en une séquence de lignes séparées par des retours à la ligne. Ce texte est affiché par le visualiseur dans une pop-up chaque fois que le pointeur de la souris est déplacé sur la représentation du noeud. Le programmeur peut surcharger cette méthode pour afficher l'état interne de ce noeud.

### 2.2.2 L'interface des noeuds

Les primitives *in()* constituent l'interface d'entrée :

- *InChannelSet in()* retourne l'ensemble de canaux d'entrée entrant sur le noeud sur lequel le programme est exécuté.
- *InChannel in(int i)* avec $0 \leq i < s.getSize()$ retourne le canal d'entrée d'index $i$ entrant dans le noeud sur lequel le programme est exécuté.

Symétriquement les fonctions *out()* constituent l'interface de sortie :

- *OutChannelSet out()* retourne l'ensemble de canaux de sortie quittant le noeud sur lequel le programme est exécuté.
- *OutChannel out(int i)* avec $0 \leq i < s.getSize()$ retourne le canal de sortie d'index $i$ quittant le noeud sur lequel le programme est exécuté.

Les primitives de communications sur les canaux sont abordées à la section 3.1.1.

### 2.2.3 Contrôle de l'exécution

Un programme peut contrôler son exécution par l'utilisation des primitives suivantes :

- *assert(GlobalAssertion a)* vérifie si l'affirmation globale $a$ est vraie pour tous les noeuds du réseau. Si l'affirmation échoue, l'exécution du réseau est terminée avec un message d'erreur, sinon l'exécution continue.
- *int getTime()* retourne l'horloge locale au noeud courant. Il est garanti que l'horloge progresse à chaque invocation des méthodes *send()*, *receive()* et *sleep()*.
- *sleep(int i)* désactive l'exécution du noeud courant pendant $i$ unités de temps.
- *yield()* retire le noeud courant de l'ordonnanceur, i.e. cette primitive donne à un autre noeud la possibilité de s'exécuter. *yield()* est implicitement appelée par chaque invocation de *send()*, *receive()* et *select()*. A part ces opérations, l'exécution du noeud n'est pas interrompue. Une invocation de cette primitive peut avancer l'horloge locale du noeud.
- *interrupt()* : si le réseau n'est pas exécuté en mode visuel, cette fonction ne fait rien ; sinon, l'exécution du réseau est interrompue jusqu'à ce que l'utilisateur la relance.
- *exit(int i)* termine l'exécution de tout le réseau avec le code de retour $i$.

# Chapitre 3

# Approfondissement de l'outil DAJ

## 3.1 Les canaux d'entrée/sortie

Un ensemble de canaux d'entrée, respectivement de sortie, est un objet du type :

```
class InChannelSet
{
  InChannnelSet()
  int getSize()
  void addChannel(InChannel c)
  InChannel getChannel(int i)
  int select()
  int select(int t)
}
```

```
class OutChannelSet()
{
  OutChannelSet()
  int getSize()
  void addChannel(OutChannel c)
  OutChannel getChannel(int i)
  void send(Message m)
}
```

Un objet de type *InChannel* (*OutChannel*) représente un canal sur lequel un message peut être reçu (envoyé) :

```
class InChannel
{
  Message receive()
}
```

```
class OutChannel
{
  void send(Message m)
}
```

### 3.1.1 Les ensembles de canaux

Soient les objets suivants :

- $s$ de type *InChannelSet* ou *OutChannelSet*,
- $c$ de type *InChannel* ou *OutChannel*.

Pour chacune des deux classes précédentes :

- le constructeur par défaut crée un ensemble vide,
- *int s.getSize()* retourne le nombre de canaux de l'ensemble $s$,
- *s.addChannel(c)* ajoute un canal $c$ à l'ensemble $s$ ; $c$ récupère l'index $i$ = *c.getSize()* (avant que le canal ne soit ajouté),

- *InChannel s.getChannel(int i)* et *OutChannel s.getChannel(int i)* retournent le canal d'indice $i$ avec $0 \leq i <$ *s.getSize()* de l'ensemble $s$,
- *int s.select()* retourne l'indice d'un canal non vide dans l'ensemble (non vide) $s$ ; si tous les canaux sont vides, le noeud courant est bloqué jusqu'à ce qu'un canal devienne non vide. Cet indice peut être utilisé pour référencer le canal correspondant de l'ensemble.
- *int s.select(int t)* se comporte comme la première variante excepté que le noeud courant est bloqué pour au plus $t$ unités de temps. Ensuite, si tous les canaux sont encore vides, -1 est retourné.

**L'utilisation des canaux : l'émission et la réception de messages**

Soient les objets suivants :

- $c$ de type *OutChannel*,
- $s$ de type *OutChannelSet* et
- $m$ de type *Message*.

L'émission d'un message est effectuée par l'une des deux primitives suivantes :

- *c.send(m)* envoie un message $m$ au canal de sortie $c$ ; si le noeud récepteur lié à $c$ est bloqué, ce noeud est relâché,
- *s.send(m)* envoie $m$ à tous les canaux de sortie de l'ensemble $s$.

Dans les deux cas, le noeud émetteur n'est pas bloqué même si le noeud récepteur lié au canal correspondant n'est pas prêt à recevoir $m$. Par conséquent, $c$ peut détenir un nombre arbitraire de messages.

Soit $c$ un objet de type *InChannel*, la réception d'un message est effectué par la fonction *receive()* :

- *Message c.receive()* retourne un message reçu sur le canal $c$ ; si $c$ est vide, ce noeud est bloqué jusqu'à ce que $c$ ne soit plus vide.
- *Message c.receive(int t)* se comporte comme la fonction précédente excepté que le noeud courant est bloqué pour au plus $t$ unités de temps. Ensuite, si $c$ est toujours vide, *null* est retourné.

Le message retourné par $c$ dépend du sélecteur de messages qui lui est associé ; par défaut, il s'agit du plus vieux message détenu par $c$.

## 3.2 Les files de messages

```
class MessageQueue
{
  MessageQueue()
  boolean isEmpty()
  void enqueue(Message m)
  Message dequeue()
  MessageCell getCell()
}
```

Ce type est utilisé pour implémenter le sélecteur de messages.

Une file de messages est une FIFO (First In First Out) liste de messages. Le constructeur par défaut crée une file vide. *isEmpty()* retourne vrai si et seulement si la file est vide, *enqueue()* ajoute un message à la fin de la liste et *dequeue()* extrait la tête de la file (la supprimant de la file).

Pour les manipulations bas-niveau de la file, *getCell()* retourne la cellule représentant la tête de la file (ou *null* si celle-ci est vide).

## 3.3 Les cellules de messages

```
class MessageCell
{
  MessageCell(Message m)
  MessageCell(Message m, MessageCell c)
  Message getMessage()
  MessageCell getNext()
  void setMessage(Message m)
  void setNext(MessageCell c)
}
```

Ce type est utilisé pour les accès bas-niveau aux files de messages. Une cellule de messages contient un message et un pointeur sur une cellule *next* (qui peut être *null*).

Le constructeur à un argument crée une cellule avec le message $m$ et positionne la cellule suivante, *next*), à *null*. Le constructeur à deux arguments crée une cellule avec le message $m$ et dont la cellule suivante pointe sur $c$.

*getMessage()* retourne le message de la cellule, *getNext()* retourne la cellule suivante. *setMessage()* change le message de la cellule en $m$ et *setNext()* positionne la cellule suivante à $c$.

## 3.4 Le sélecteur de messages

Le comportement du sélecteur par défaut est celui d'une FIFO (First In First Out) qui consiste à retourner le plus vieux message, c'est-à-dire le premier envoyé.

```
class SelectorDefault implements Selector
{
  Message select(MessageQueue queue)
  {
    return queue.dequeue();
  }
}
```

L'intérêt de redéfinir cette interface, par l'appel à la fonction *setSelector(Selector s)*), est de pouvoir implémenter un mécanisme de canaux défectueux consistant, par exemple, à perdre ou dupliquer des messages.

## 3.5 L'ordonnanceur de programmes

```
abstract class Scheduler
{
  int getNumber()
  boolean isReady(int i)
  int getTime(int i)
  abstract int nextProgram()
}
```

Un ordonnanceur retourne le prochain programme à exécuter parmi ceux qui correspondent à des noeuds dans l'état prêt.

L'intérêt de redéfinir un ordonnanceur - par l'appel à *setScheduler(Scheduler s)*) - est de pouvoir implémenter différents modèles de réseaux : synchrones, asynchrones, etc. En effet, avec l'ordonnanceur par défaut, un programme peut être interrompu au milieu de son exécution par un appel de *receive()* sur un canal vide par exemple. Ainsi aucun fonctionnement synchrone n'est vérifié.

Afin de modifier l'ordonnanceur, la fonction *int nextProgram()* doit être redéfinie de manière à ce que la valeur retournée $i$ soit comprise entre 0 et *getNumber()*. D'autre part, l'appel à *isReady(i)* doit retourner vrai ; cependant, si cette condition ne peut être satisfaite, l'ordonnanceur doit retourner -1 dénotant un interblocage.

La fonction *getTime(i)* retourne le temps écoulé de la simulation pour le programme $i$ pouvant être utilisé pour la décision d'ordonnancement.

L'ordonnanceur par défaut utilise un mécanisme de tourniquet pour élire le prochain programme prêt à être exécuter ; en voici le code :

```
class SchedulerDefault extends Scheduler
{
  int last = -1;
  int nextProgram()
  {
    int n = getNumber();
    boolean reset = false;
    do
      {
        last++;
        if (last == n)
          {
            last = 0;
            if (reset) return -1;
            reset = true;
          }
      }
    while (!isReady(last));
    return last;
  }
}
```

Par ailleurs, les ordonnanceurs suivants sont prédéfinis :

- *SchudelerMinTime* : choisit le programme prêt de manière à ce que le temps de la simulation écoulé pour ce programme soit minimal,
- *SchudelerMaxTime* : choisit le programme prêt de manière à ce que le temps de la simulation écoulé pour ce programme soit maximal.

## 3.6 Les assertions

```
abstract class GlobalAssertion
{
  abstract boolean assert(Program program[])
  String getText()
  static Message[] getMessages(InChannel c)
  static Message[] getMessages(OutChannel c)
}
```

Quand un objet $a$ de ce type est passé à la fonction *assert()*, la fonction *a.assert(p)* est ensuite invoquée sur $p$ le tableau de programmes associés aux noeuds du réseau, selon l'ordre dans lequel ils ont été créés. Si la fonction retourne faux alors l'exécution est interrompue avec le message d'erreur retourné par *a.getText()* ; sinon l'exécution continue.

Quand *a.assert(p)* est invoqué, cette fonction peut s'appuyer sur le fait que chaque programme *p[i]* est soit dans l'état initial, soit a terminé son exécution, soit bloqué sur une opération de communication **avant** que le message correspondant ait été envoyé, soit a explicitement interrompu son exécution par un apppel à la fonction *yield()*.

Soit $c$ un objet de type *InChannel* ou *OutChannel* ; la fonction *Message[] getMessages(c)* peut être utilisée par une assertion pour déterminer le contenu d'un canal $c$. Sa valeur de retour est le tableau de messages contenu dans $c$ selon l'ordre dans lequel ils ont été envoyés.

Un objet de ce type (ou d'un sous-type de celui-ci) peut être envoyé, respectivement reçu, via ce canal. La méthode *getText()* retourne une chaîne de caractères (par défaut "(no information)") qui est affichée par le visualiseur dans une pop-up chaque fois que le pointeur de souris est déplacé sur la représentation visuelle d'un canal qui contient un message. Le programmeur peut surcharger cette méthode pour afficher le contenu d'un message.

# Conclusion

Daj est une bibliothèque de classes Java permettant d'implémenter et de visualiser des algorithmes distribués. Son utilisation minimale consiste à redéfinir certaines méthodes des classes :

- *Application* dédiée principalement à la mise en place du réseau physique et
- *Program* gèrant les programmes s'exécutant sur chacun des noeuds du réseau.

Par ailleurs, afin que le programmeur puisse élaborer ses propres modèles d'exécutions synchrones, asynchrones, avec pertes de messages, etc. il lui est possible de redéfinir le comportement de certaines structures utilisées dont le sélecteur de messages et l'ordonnanceur.

Enfin, cet outil intègre un ensemble de mécanismes permettant de va certaines propriétés du réseau telles que les invariants.

# ANNEXES

## .1 Les problèmes connus

L'outil a été développé et testé sous Linux, Solaris, Windows NT 4.0 avec la JDK 1.0.2 (Linux, Solaris), JDK 1.0.3 (Linux), JDK 1.0.4 (Solaris), Netscape Navigator 3.02 (Linux, Solaris), Netscape Communicator 4.03 (Linux, Solaris, Windows NT 4.0).

Les problèmes suivants concernent l'interface utilisateur mais ne peuvent être résolus à cause des limites et des bugs des différentes versions de java AWT :

**JDK pop-up dans les applets**
Toutes les version de la JDK ont apparemment des sérieux bugs dans la classe window en mode applet qui empêchent de déterminer la taille d'une pop-up et d'afficher leurs contenus de manière à ce qu'ils soient visibles sous différents navigateurs. Le message d'avertissement de l'applet est affiché dans différents cotés de la fenêtre selon les navigateurs sans ajustement de la taille et de la coordination du système.

Daj pour la JDK 1.0 utilise un "*hack*" qui rend les contenus visibles sous la plupart des navigateurs (JDK 1.0 appletviewer, Netscape 3.0/4.0 sous Linux, Solaris et Windows NT 4.0) mais produit des fenêtres démesurées. DAJ pour la JDK 1.1 utilise un "*hack*" qui rend les contenus visibles sous appletviewer de la JDK 1.0 et donnent des résultats encore plus horribles.

**JDK 1.1 ascenseurs**
Dans la JDK 1.1 les ascenseurs de positions se comportent différemment de ceux de la JDK 1.0 ; par conséquent le code source est différent pour ces deux versions. L'utilisation d'ascenseurs dans la fenêtre de visualisation pour une application développée avec la JDK 1.0 ne fonctionne pas correctement sous la JDK 1.1 (et problablement réciproquement).

**JDK 1.0 consommation de la mémoire des X-Server**
Dans la JDK 1.1 sous X-Windows, durant l'exécution d'une visualisation, la consommation de mémoire du serveur X augmente jusqu'à interrompre la visualisation (la mémoire redevient alors immédiatement disponible). Ceci pourrait écrouler un terminal X durant les exécutions de longues visualisations.

**JDK 1.0 ascenseurs dans les applets**
Dans la JDK 1.0, sur différents navigateurs, la taille de l'ascenseur vertical ne correspond pas avec la taille de la zone de visualisation. Dans Netscape 4.0 sous Solaris, cet ascenseur est trop petit d'un pixel, et dans Netscape 4.0 sous Windows NT 4.0, cet ascenseur est trop petit d'environ une largeur

du cadre.

**JDK 1.0 les positions des pop-up**
Dans la JDK 1.0 le calcul de l'emplacement d'un composant à partir de son composant parent ne semble pas fonctionner correctement. Les pop-up n'apparaissent à l'endroit spécifié mais à quelques pixels aux alentours.

**JDK 1.0 redimensionnement des fenêtres**
Dans la JDK 1.0 le redimensionnement de la fenêtre de visualisation retourne une coordonnée 0 à l'événement correspondant. Par conséquent, il n'est apparemment pas possible de déterminer la position correcte de la fenêtre si celle-ci est redimensionnée depuis le coin en haut à gauche. Les pop-up n'apparaissent pas à l'endroit voulu jusqu'à ce que la fenêtre soit déplacée.

**JDK 1.0 fermeture des fenêtres**
Dans la JDK 1.0 il n'est apparemment pas possible de déclencher une action lorsque l'utilisateur sélectionne l'élément du menu "Close" dans un cadre d'une fenêtre. Celle-ci reste ouverte.

**JDK 1.0 fenêtre de Visualisation pour les applets**
Dans la JDK 1.0 il n'y a apparemment pas de possibilité de déterminer la position d'une applet dans les coordonnées de l'écran. La fenêtre de visualisation s'ouvre par conséquent toujours à la même place.

**JDK 1.0 Menu Help**
Dans la JDK 1.0 sous Linux, mettre en place les menus d'aides dans la barre des menus ne fonctionne pas. Par conséquent, le menu d'aide est généralement ajouté en tant qu'un menu normal justifié à gauche dans DAJ pour JDK 1.0.

**JDK ascenseurs**
Dans les deux versions de la JDK, il est apparemment impossible de déterminer la largeur/hauteur d'un ascenseur avant que le composant auquel il se rattache ne soit créé. Par conséquent la fenêtre de visualisation est légèrement plus petite que selon les spécifications du programmeur (la zone inclu à la fois l'ascenseur horizontal et vertical).

**Netscape 4.0 rafraîchissemnt d'une applet**
Dans Netscape 4.0 sous Windows, la zone de visualisation d'une applet DAJ n'est pas systématiquement redessinée quand la pop-up est fermée.

**Netscape 4.0 arrêts d'une applet**
Netscape 4.0 ne déclenche pas la méthode *stop* d'une applet quand le navigateur est iconifié, par conséquent la fenêtre de visualisation reste ouverte.

## .2 Un exemple d'implémentation d'un algorithme simple

Le code source original est disponible sur `http://www.risc.uni-linz.ac.at/software/daj/3nodes/index_1.html`, nous avons commenté cette exemple en Français afin d'en faciliter la compréhension.

```
// ------------------------------------------------------------------------
// $Id: algorithm.tex,v 1.2 1997/11/06 14:37:28 schreine Exp schreine $
// a trivial test application
//
// (c) 1997, Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
// http://www.risc.uni-linz.ac.at/software/daj
// ------------------------------------------------------------------------
import daj.*;

public class Main extends Application
{

  // Exécution de l'application
  public static void main(String[] args)
  {
    new Main().run();
  }

  // Lancement de la fenêtre graphique
  public Main()
  {
    super("A Trivial Program", 400, 300);
  }

  // Construction du réseau
  public void construct()
  {

    // 3 noeuds
    Node node0 = node(new Prog(0), "0", 100, 100);
    Node node1 = node(new Prog(1), "1", 150, 200);
    Node node2 = node(new Prog(2), "2", 300, 150);

    // canaux unidirectionnels reliant
    // chacun des noeuds
    link(node0, node1);
    link(node1, node2);
    link(node2, node0);
    link(node0, node2);
    link(node2, node1);
    link(node1, node0);
```

```
  }

  // Message d'information affiché
  // à partir du menu Help
  public String getText()
  {
    return "A Trivial Program\n \n" +
      "2 messages circling bidirectionally\n" +
      "through a ring of three nodes";
  }
}

// Programme exécuté par chacun des noeuds
class Prog extends Program
{
  private int number;
  public int index;
  public Message msg;
  public boolean sent;

  // initialisation
  public Prog(int i)
  {
    number = i;
    msg = null;
    sent = false;
  }

  // programme lui-même
  public void main()
  {
    // le noeud 0 envoie un msg
    // sur chacun de ses canaux sortants (out(0)
    // et out(1))
    if (number == 0)
      {
        out(0).send(new Msg(0));
        out(1).send(new Msg(1));
        sent = true;
      }

    // nouvelle assertion sur le nombre de messages
    GlobalAssertion assertion = new NumberOfMessages();

    // chacun des noeuds va ensuite récupérer
    // un message sur ses canaux d'entrée (in(index).receive())
    // (si ceux-ci sont non vides) et le renvoie à l'émetteur
```

```
    // (out(index).send(msg))
    for (int i = 0; i < 5; i++)
      {
        assert(assertion);
        index = in().select();
        msg = in(index).receive();
        out(index).send(msg);
        msg = null;
      }
  }

  // informations sur l'état d'un noeud
  public String getText()
  {
    String msgString;
    if (msg == null)
      msgString = "(null)";
    else
      msgString = msg.getText();
    return
      "sent: " + String.valueOf(sent) +
      "\nmsg: " + msgString;
  }
}

// Extension de la classe message, celle-ci
// contient simplement un entier
class Msg extends Message
{
  int val;
  public Msg(int i)
  {
    val = i;
  }
  public int value()
  {
    return val;
  }
  public String getText()
  {
    return "content = " + String.valueOf(val);
  }
}

// Assertion vérifiant qu'il existe toujours deux
// messages dans le réseau lors de l'exécution
class NumberOfMessages extends GlobalAssertion
```

```
{
  int count;

  public String getText()
  {
    return "invalid count of messages: " +
      String.valueOf(count);
  }

  public boolean assert(Program prog[])
  {
    if (!((Prog)prog[0]).sent) return true;
    count = 0;
    for (int j = 0; j < prog.length; j++)
      {
        Prog program = (Prog)prog[j];
        count += getMessages(program.out(0)).length;
        count += getMessages(program.out(1)).length;
        if (program.msg != null) count++;
      }
    return count == 2;
  }
}
```

L'exécution de cet exemple est disponible en version applet sur http://www.risc.uni-linz.ac.at/software/daj/3nodes/