# Un environnement général pour étudier et implémenter les algorithmes distribués

**Résumé**

Nous présentons une méthode générale et un outil pour concevoir, développer et visualiser les algorithmes distribués. Pour cela, nous utilisons un codage de haut niveau des algorithmes distribués à l'aide des systèmes de réécriture de graphe et des calculs locaux. Au final, nous obtenons un ensemble d'outils simples pour décrire, implémenter et visualiser une large famille d'algorithmes distribués. Nous avons aussi développé un logiciel pour programmer, expérimenter et visualiser l'exécution des algorithmes distribués.

## 1 Introduction

L'implémentation, le débogage, le test et l'expérimentation des algorithmes distribués sont une tâche assez complexe et délicate, associée à de nombreux pièges et difficultés. Dans ce contexte, il est essentiel de comprendre les idées algorithmiques de haut niveau, indépendamment du langage et de la plate-forme de l'implémentation. La visualisation de l'exécution d'un algorithme distribué fournit une abstraction permettant de mieux comprendre son comportement et donc de simplifier sa mise au point et sa preuve. La perception des réseaux et des événements obtenus à partir de la visualisation graphique est plus simple pour les humains et donne plus d'informations qu'une description textuelle. La complexité des algorithmes distribués, en raison de la communication entre processus et de la synchronisation, exige des modèles de conceptions pour simplifier et modéliser leur visualisation.

L'approche que nous allons suivre dans ce papier est fondée essentiellement sur les calculs locaux comme un modèle basique pour décrire les algorithmes distribués. Nous présentons une méthode qui produit une implémentation automatique pour tous les algorithmes distribués qui sont décrits en terme de calculs locaux. L'implémentation est alors utilisée pour animer l'exécution de l'algorithme, pour aider à prouver sa validité ou bien pour faire des expérimentations. Tout ceci est fait à travers un outil écrit en Java appelé Visidia [3, 2, 4, 19] (Visualization and simulation of distributed algorithms), son interface graphique permet à l'utilisateur de créer un réseau et de prototyper les algorithmes distribués. Notre description du calcul local est fortement liée au modèle de systèmes de réécriture de graphe développé par Métivier et al. [14]. Ce modèle fournit une base théorique solide avec beaucoup de résultats généraux. La contribution de ce papier est de démontrer que les calculs locaux sont utiles pour l'étude et la visualisation d'une large famille d'algorithmes distribués. Par conséquent, nous obtenons un modèle très général pour l'étude des algorithmes distribués, de la phase de leur conception théorique jusqu'à leur implémentation sur un système distribué. La caractéristique fondamentale de ce modèle est l'absence de connaissance globale. Tous les algorithmes effectuent des actions locales et l'objectif est de construire un résultat globalement cohérent.

Un réseau anonyme de processeurs avec topologie arbitraire est représenté par un graphe connexe, non orienté où les noeuds représentent les processeurs et les arêtes représentent les canaux de communication. Les états des processeurs sont représentés par des étiquettes et l'exécution d'un algorithme sur un réseau consiste simplement à faire évoluer ces étiquettes. Les étiquettes des noeuds et des arêtes sont modifiées localement, se basant sur un sous-graphe de rayon fixe $k$ du graphe donné, selon certaines règles qui dépendent seulement du sous-graphe. Les réécritures sont appliquées tant qu'il y a des règles applicables. Deux étapes de réécriture sont indépendantes si elles sont appliquées sur deux sous-graphes disjoints. Dans ce cas, elles peuvent être exécutées dans n'importe quel ordre et même en même temps.

Le modèle du système distribué est un réseau distribué et asynchrone de processeurs qui communiquent par échange de messages. Pour surmonter le problème de certains algorithmes distribués non déterministes et pour faciliter l'implémentation, nous utilisons des modèles probabilistes [9, 20, 15]. Métivier et al. [17, 18] ont étudié les algorithmes probabilistes pour implémenter les algorithmes distribués décrits sous forme de calculs locaux. Intuitivement, chaque processus essaie de se synchroniser de manière aléatoire avec un de ses voisins ou avec tous ses voisins selon le modèle choisi, puis, une fois synchronisé, le calcul local peut être fait. L'implémentation de ces procédures de synchronisation et leurs propriétés sont données et expliquées dans [17, 18].

Nous utilisons ces techniques pour la visualisation de l'exécution d'un algorithme distribué. Toutes les synchronisations locales aléatoires dans tout le réseau sont affichées et tous les messages échangés durant cette synchronisation peuvent être visualisés. Par conséquent, la visualisation de l'exécution de l'algorithme entier est effectuée jusqu'à l'arrêt. Nous avons développé un outil avec une interface graphique pour éditer le réseau et une autre interface pour implémenter et visualiser l'exécution des algorithmes distribués.

Le papier est organisé comme suit : dans la Section 2 nous rappelons les calculs locaux, les systèmes de réécriture de graphe et la façon de les utiliser pour décrire les algorithmes distribués. Dans la Section 3 nous présentons une méthode générale pour produire une implémentation automatique des algorithmes distribués par les calculs locaux. Dans la Section 4, Visidia, un outil basé sur ce modèle, est décrit. Enfin, dans la Section 5 nous concluons ce papier.

## 2 Les algorithmes distribués sous forme de calculs locaux

Dans cette section, nous illustrons, d'une manière intuitive, la notion des calculs locaux et en particulier des systèmes de réécriture de graphe montrant comment nous pouvons coder des algorithmes sur des réseaux des processeurs [13]. Un réseau est représenté par un graphe dont les sommets représentent des processeurs et les arêtes représentent les canaux de communication (bidirectionnels) entre ces processeurs. À chaque instant, chaque sommet (resp. arête) est dans un état particulier, codé par des étiquettes. Se basant sur son propre état et sur les états de ses voisins, chaque noeud peut décider de faire une *étape de réécriture*. Après cet étape, l'état de ce noeud , de ses voisins et des arêtes incidentes, peuvent changer selon une *règle de réécriture* spécifique.

Les systèmes de réécriture de graphe vérifient les conditions suivantes:

(C1) le graphe sous-jacent ne change pas. Seuls les états de ses composantes (arêtes et/ou sommets) sont modifiés, le résultat étant le réétiquetage final,

(C2) le calcul est local, chaque étape de réécriture ne modifie qu'un sous-graphe de taille fixe du graphe sous-jacent,

(C3) ils sont localement engendrés, c'est à dire, la condition d'application d'une réécriture dépend seulement des états locaux du sous-graphe considéré.

Pour de tels systèmes, l'aspect distribué vient du fait que plusieurs étapes de réécriture peuvent être exécutées simultanément sur des sous-graphes disjoints, donnant le même résultat qu'une réalisation séquentielle de leurs exécutions dans n'importe quel ordre.

Une large famille d'algorithmes distribués codés par les systèmes de réécriture de graphe est donnée dans [2, 4]. Afin de rendre les définitions plus faciles à lire, nous donnons un exemple d'un système de réécriture de graphe pour le calcul d'un arbre recouvrant et un autre exemple utilisant les calculs locaux pour le problème de conflits.

### 2.1 Calcul distribué d'un arbre recouvrant (premier exemple)

D'abord nous illustrons les systèmes de réécriture de graphe en considérant un algorithme distribué simple qui calcule un arbre recouvrant d'un réseau. Nous supposons l'existence d'un processeur distingué unique dans le réseau qui est dans un état "actif" (codé par l'étiquette **A**), tous les autres processeurs sont dans un état "neutre" (étiquette **N**) et tous les canaux de communications sont dans un état "passif" (étiquette **0**). Au début, l'arbre ne contient que le sommet actif. A n'importe quelle étape du calcul, un sommet actif peut activer un de ses voisins neutres et marque le canal de communication correspondant par la nouvelle étiquette **1**. Le calcul s'arrête dès que tous les processeurs sont activés. L'arbre recouvrant est alors obtenu en considérant tous les canaux de communications avec l'étiquette **1**.

Une étape élémentaire de calcul peut être présentée comme une *étape de réécriture* en utilisant la règle de réécriture suivante $R$ qui décrit les modifications d'étiquettes correspondantes:


A chaque fois qu'un noeud étiqueté **A** est lié par une arête étiquetée **0** à un noeud dont l'état est **N**, alors le sous-graphe correspondant peut se réécrire selon la règle $R$.

Un exemple de calcul utilisant cette règle est donné dans Figure 1. Les étapes de réécriture *peuvent* s'exécuter simultanément dans des parties disjointes du graphe. Quand le graphe devient irréductible, i.e. aucune règle ne peut s'appliquer, un arbre recouvrant, constitué des arêtes étiquetées **1**, est alors calculé.

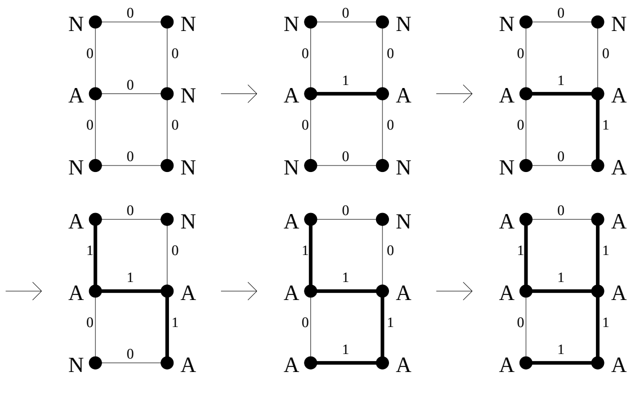


FIG. 1 – *Calcul distribué d'un arbre recouvrant*

### 2.2 Le problème des conflits (deuxième exemple)

Le deuxième exemple est le calcul distribué du problème des conflits entre processus [16] qui illustre la possibilité qu'offrent les systèmes de réécriture de graphe pour représenter les algorithmes distribués et les paradigmes classiques.

La résolution de conflits est une généralisation du problème du dîner des philosophes (the dining philosophers problem) [7, 12], connu aussi sous le nom de boisson des philosophes “the drinking philosophers problem” [5, 21], où plusieurs ressources sont partagés par plusieurs processus (ou processeurs). Une ressource doit être utilisée par un et un seul processus à la fois. Chaque processus, qui a besoin de la ressource, doit l’avoir après un temps fini et il peut aussi y accéder autant de fois qu’il le souhaite.

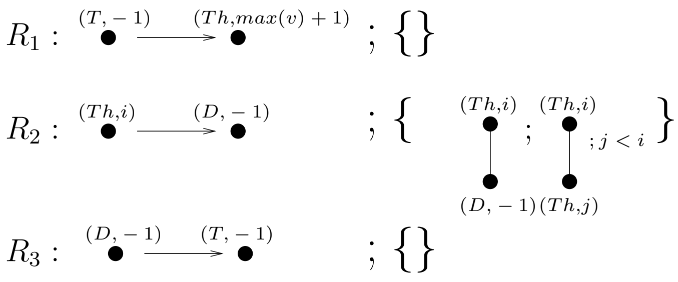
Chaque processus peut avoir trois états : *tranquille (tranquil)*, *assoiffé (thirsty)* ou *buveur (drinking)*. Ces états seront codés avec les étiquettes $T$, $Th$ et $D$. Un autre état est utilisé, cet état est un entier représentant le rang de la requête. C’est ainsi que chaque noeud sera étiqueté par $\mathcal{L}(v) = (X,i)$ avec $X \in \{T,Th,D\}$ et $i$ un entier correspondant au rang.

Initialement, tous les noeuds du graphe sont dans l’état *tranquille*, cela se traduit par l’étiquette $(T, -1)$. A chaque étape de calcul, chaque noeud $u$ à l’état *tranquille* peut demander l’accès à ses sections critiques, il devient alors *assoiffé*. Dans ce cas, $u$ change son état à $(Th,i+1)$ où $i = max\{k_v | v \in B(u,1)$ et $\lambda(v) = (X,k_v), X \in \{T,Th,D\}\}$, où $B(u,1)$ est la boule de centre $u$ et de rayon 1, i.e. $B(u,1)$ est l’ensemble des noeuds contenant $u$ et tous ses voisins et toutes les arêtes incidentes à $u$ . Cela signifie que l’ordre de $u$ est le plus grand parmi ceux de ses voisins. Dans un graphe complet, cet ordre peut être vu comme un temps universel puisque seulement un seul noeud peut changer son état en $Th$. Dans ce cas, notre algorithme calcule l’exclusion mutuelle.

Si un noeud $u$, avec une étiquette $(Th,i)$, n’a pas de voisin dans la section critique (avec l’étiquette $(D, -1)$) et n’a pas de voisin avec une étiquette $(Th,j)$ avec $j < i$ (le rang de $u$ est le plus petit de ses voisins), le noeud $u$ peut entrer dans la section critique. Dans ce cas, $u$ aura l’étiquette $(D, -1)$.

Une fois que le noeud dans la section critique a terminé, il retourne à un état *tranquille*. Son étiquette redevient $(T, -1)$.

On peut coder cet algorithme par le système suivant. Initialement, tous les sommets sont étiquetés $(T,-1)$. A chaque étape, un calcul local est appliqué en utilisant les règles de réécriture de graphe avec contextes interdits suivantes :




Les contextes interdits sont les sous-graphes entre accolades. Les accolades vides qui figurent dans les règles $R_1$ et $R_3$ indiquent qu’il n’y a pas de restriction pour appliquer ces deux règles. Toutefois, la règle $R_2$ ne peut être appliquée que si aucun des deux sous-graphes du contexte interdit n’est présent dans la partie du graphe où l’on veut appliquer la règle. Autrement dit, un noeud étiqueté $(Th,i)$ ne possède aucun voisin avec l’étiquette $(D, -1)$ ni de voisin avec l’étiquette $(Th,j)$ avec $j < i$, peut appliquer la règle $R_2$.

**Remarque 2.1** *Les définitions formelles et les propriétés des calculs locaux se trouvent dans [4].*

# 3 Des règles de réécriture de graphe aux systèmes d'échange de messages

L'exécution d'un algorithme distribué se ramène à l'exécution de ses règles de réécriture. Par conséquent, il suffit d'implémenter les étapes élémentaires des réécritures locales. Pour cela, nous présenterons le modèle du système distribué puis une méthode d'implémentation des calculs locaux.

## 3.1 Un modèle de système distribué

Dans la représentation du système distribué exécutant des règles de réécriture, chaque noeud peut être considéré comme une entité autonome de calcul et sera implémenté par un processus, un Thread etc. Chaque arête modélise simplement un canal de communication entre les entités de calculs correspondantes. Le système est asynchrone; i.e. il n'y a aucune horloge globale. Les sommets ont seulement une vue locale du graphe et communiquent seulement avec leurs voisins par des messages asynchrones. En fait, un sommet $v$ est équipé de ports, numérotées de 0 à $(deg(v)-1)$, qui seront utilisés pour communiquer avec ses voisins. L'attribution de ces nombres est complètement arbitraire et ne dépend pas des identités des noeuds voisins. Nous supposons que, pour un couple de sommets voisins, l'ordre d'envoi de messages est identique à celui de leur réception. Pour la plupart des algorithmes, le réseau est anonyme ce qui signifie que les sommets n'ont pas d'identités.

## 3.2 Les différents types des calculs locaux

Bien qu'ils aient en commun le fait que l'état d'un noeud dépend seulement de l'état de ses voisins ou de certains de ses voisins, les calculs locaux sont de trois types :

$RV$ (Rendez-Vous): dans une étape de calcul, les étiquettes d'un couple de sommets reliés par une arête sont modifiées selon la règle où les étiquettes apparaissant sur l'arête et sur ses sommets. Le premier exemple présenté dans la section 2.1 correspond à ce type de calcul.

$LC_1$ (Calcul local de type 1): dans une étape de calcul, l'étiquette attachée au centre de l'étoile est modifiée selon une règle qui dépend des étiquettes de l'étoile (les étiquettes des feuilles ne sont pas modifiées). L'exemple présenté dans la section 2.2 correspond à ce type de calcul. On appelle étoile un sous-graphe composé d'un sommet et de tous ses voisins.

$LC_2$ (Calcul local de type 2): dans une étape de calcul, les étiquettes attachées au centre et aux feuilles de l'étoile peuvent être modifiées selon une règle qui dépend des étiquettes de l'étoile.

Les algorithmes distribués que nous considérons dans cet article sont définis par des règles appartenant aux trois types précédents. Par conséquent, pour implémenter un algorithme distribué, il faut d'abord implémenter ces types de calculs locaux.

## 3.3 Implémentation avec des procédures probabilistes

Puisque Angluin [1] a montré qu'il n'y a aucun algorithme déterministe pour implémenter des synchronisations locales dans un réseau anonyme par échange de messages asynchrones (voir [20]), nous allons utiliser des procédures probabilistes pour les implémenter (voir [17, 18, 2]). D'ailleurs, l'aspect aléatoire fournit des réalisations efficaces et faciles, en particulier, dans le contexte de la visualisation parce qu'il permet à l'utilisateur d'observer l'exécution entière de l'algorithme comme nous allons le montrer dans la suite.

**Implémentation de** $RV$. Nous considérons la procédure probabiliste suivante donnée dans l'algorithme 1 pour implémenter le Rendez-Vous. L'exécution est divisée en rounds; dans chaque round, chaque sommet $v$ choisit au hasard un de ses voisins $c(v)$, et lui envoie 1 et envoie 0 à tous les autres. Il y a un rendez-vous entre $v$ et $c(v)$ si $c(v) = v$, on dit alors que $v$ et $c(v)$ sont synchronisés. Quand $v$ et $c(v)$ sont synchronisés il y a un échange des messages entre $v$ et $c(v)$ qui permet aux deux noeuds de modifier leurs étiquettes.

**algorithme 1** Rendez-vous probabiliste

*Chaque noeud v répète infiniment les actions suivantes :*
*Un noeud v choisit au hasard un de ses voisins* $c(v)$*;*
*le noeud v envoie* 1 *à* $c(v)$;
*le noeud v envoie* 0 *aux autres voisins à part* $c(v)$;
*le noeud v reçoit les messages de ses voisins.*
*(* Il y a un rendez-vous entre v et* $c(v)$ *si v reçoit* 1 *de* $c(v)$; *dans ce cas, une étape de calcul peut être effectuée. *)*

**Implémentation de** $LC_1$. Soit $LE_1$ l'élection locale donnée par l'algorithme 2 pour implémenter $LC_1$; elle est divisée en rounds, dans chaque round, chaque processeur $v$ choisit un nombre entier $rand(v)$ de manière aléatoire et l'envoie à ses voisins. $v$ est alors élu dans $B(v,1)$ si pour chaque sommet $w$ de $B(v,1)$ différent de $v$ : $rand(v) > rand(w)$. Dans ce cas, une étape de calcul dans $B(v,1)$ sera exécutée: le centre reçoit les étiquettes de ses feuilles (voisins) puis il change son étiquette.

**algorithme 2** Élection $LE_1$ probabiliste

*Chaque noeud v répète indéfiniment les actions suivantes:*
*le noeud v choisit au hasard un entier* $rand(v)$*;*
*le noeud v envoie* $rand(v)$ *à tous ses voisins;*
*le noeud v reçoit les entiers de tous ses voisins.*
*(* le noeud v est élu si* $rand(v)$ *est strictement supérieur à tous les entiers reçus par* $v$; *dans ce cas, une étape de calculs peut être exécutée dans* $B(v,1)$*. *)*

**Implémentation de** $LC_2$. Soit $LE_2$ l'élection locale donnée par l'algorithme 3 pour implémenter $LC_2$; elle est divisée en rounds, dans chaque round, chaque processeur $v$ choisit aléatoirement un nombre $rand(v)$.

Le processeur $v$ envoie à ses voisins $rand(v)$. Quand il reçoit de tous ses voisins leurs entiers, il envoie à chaque voisin $w$ l'entier maximum de l'ensemble des entiers reçus de ses voisins sauf $rand(w)$. $v$ est élu dans $B(v,2)$ si $rand(v)$ est strictement supérieur à tous les entiers $rand(w)$ pour tout sommet $w$ dans la boule de centre $v$ et de rayon 2; dans ce cas, une étape de calcul peut être faire dans $B(v,1)$. Durant cette étape de calcul, il y a un échange d'étiquette des noeuds de $B(v,1)$ qui permet aux noeuds de $B(v,1)$ de mettre à jour leurs étiquettes.

## 3.4 Application des règles de réécriture

Comme nous l'avons déjà expliqué, chaque processeur essaie de manière aléatoire d'obtenir une synchronisation, avec un voisin ou avec tous ses voisins (selon le type de calculs locaux). Une fois qu'un processeur $v$ est impliqué dans une synchronisation, une étape de réécriture peut être exécutée. C'est-à-dire, $v$ échange ses étiquettes et attributs avec son voisin(s), il cherche si

**algorithme 3** Élection $LE_2$ probabiliste

*Chaque noeud v répète indéfiniment les actions suivantes :*
*le noeud v choisit au hasard un entier* $rand(v)$*;*
*le noeud v envoie* $rand(v)$ *à ses voisins;*
*le noeud v reçoit un message de tous ses voisins;*
*soit* $Int_w$ *l'entier maximum de l'ensemble des entiers que v a reçu de ses voisins différents de* $w$*;*
*le noeud v envoie à chaque sommet w* $Int_w$;
*le noeud v reçoit les entiers de tous ses voisins;*
*(* Il y a une élection* $LE_2-$ *dans* $B(v{,}2)$ *si* $rand(v)$ *est strictement supérieur à tous les entiers reçus de v; dans ce cas, une étape de calcul peut être faite dans* $B(v{,}1)$*. *)*

une configuration d'un membre gauche des règles est trouvée, et si oui, il exécute un calcul local et met à jour ses étiquettes et ses attributs selon la configuration du membre droit de la règle et enfin sort de la synchronisation. Après, il essaie d'obtenir une autre synchronisation et ainsi de suite.
Notons que beaucoup de synchronisations peuvent se produire en même temps dans le réseau. Comme les procédures probabilistes implémentant ces synchronisations sont du type Las Vegas [18], chaque sommet devient impliqué dans une synchronisation dans un temps fini (expérimentalement raisonnable).

# 4 Visidia: un outil pour prototyper et visualiser les algorithmes distribués

Nous avons développé un outil appelé Visidia [3, 2, 19] pour appliquer notre approche à la visualisation et à l'animation des algorithmes distribués. Il est écrit en Java où les processeurs distribués sont implémentés par des Threads Java.

L'interface graphique de l'outil, comme expliqué ci-dessous, est un environnement graphique qui permet à l'utilisateur de dessiner facilement un réseau et visualiser l'exécution d'un algorithme distribué. L'architecture de l'outil se compose de trois parties principales, à savoir, l'interface graphique, le simulateur et la bibliothèque d'algorithmes distribués (voir **Fig.2**). Ces modules sont bien séparés, ce qui implique que la possibilité de les modifier et de les développer de manière indépendante.

## 4.1 L'interface graphique (GUI)

L'interface graphique de Visidia permet à l'utilisateur de construire un réseau par une simple manipulation de la souris "Drag & Drop". L'utilisateur peut ajouter, supprimer, ou choisir des sommets, des arêtes ou des sous-graphes. Les attributs visuels d'un sommet : étiquettes, couleurs, et formes— peuvent également être modifiés par l'utilisateur. Il est également possible d'importer un autre format de fichier, par exemple le GML [11] offrant ainsi la possibilité de charger des graphes produit par d'autres éditeurs de graphes.
Une fois le réseau dessiné, la simulation est exécutée dès que l'utilisateur a choisi un algorithme. Durant l'exécution, les messages échangés entre les noeuds et leurs valeurs sont affichés et le statut des arêtes et des noeuds sont mis à jour en temps réel. Une fenêtre permet à l'utilisateur de contrôler la visualisation comme début, pause et arrêt. D'ailleurs, il est possible de modifier la vitesse d'animation des messages en utilisant le menu.

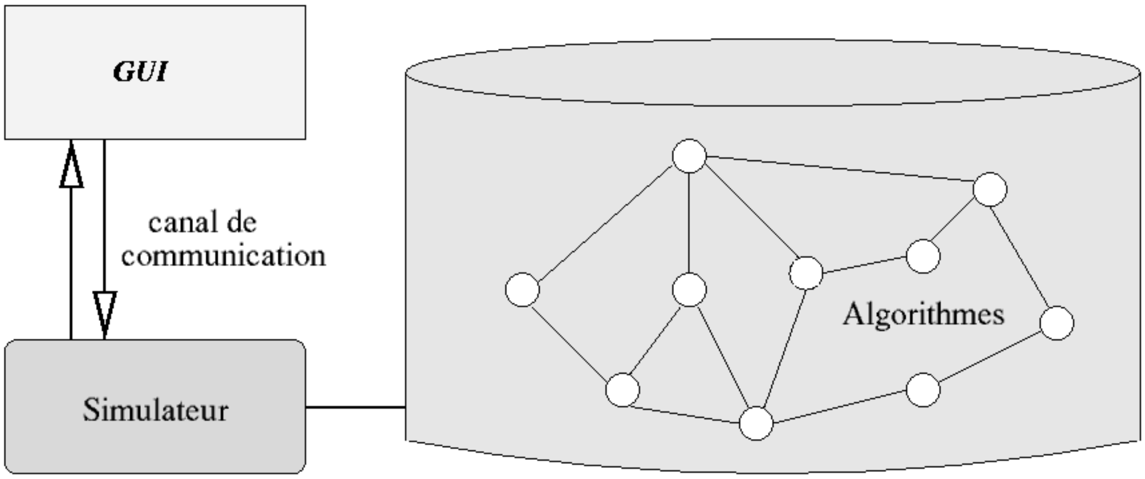



FIG. 2 – *Architecture de Visidia*

### 4.2 Le simulateur

Le simulateur est le lien entre l'interface graphique et les algorithmes. Il modélise un réseau de processeurs asynchrones. Chaque processeur communique seulement avec ses voisins par échange de message. Dans cette version de l'outil, chaque processeur est implémenté par un *Thread Java* comme indiqué ci-dessus. Le simulateur contrôle les échanges de messages entre les Threads, ainsi que la visualisation des événements.

### 4.3 La bibliothèque d'algorithmes

Un algorithme est implémenté en Java et sera copié à chaque sommet du graphe et exécuté de manière asynchrone par son processus. Un sommet est implémenté par une classe qui contient son identifiant, son état interne, son degré, et éventuellement la taille du graphe. Il est possible pour le programmeur d'utiliser aussi les primitives suivantes:

– `rendezVous()`: une fonction qui retourne le numéro de port du voisin avec qui il y a eu une synchronisation.
– `starSynchro1()`: retourne le centre d'une étoile pendant une synchronisation en étoile. Seul le centre peut changer son étiquette.
– `starSynchro2()`: retourne le centre d'une étoile pendant une synchronisation en étoile. Le centre et ses voisins peuvent changer leurs étiquettes.
– `breakSynchro()`: Arrête la synchronisation. Les sommets appartenant à la synchronisation seront débloqués.
– `getId()`: retourne l'identifiant d'un noeud (pour les réseaux avec identités),
– `getProperty(String)`: (resp. `putProperty(String,Object)`) donne (resp. change) l'état d'un noeud,
– `getArity()`: retourne le degré d'un noeud, i.e. le nombre de ses voisins,
– `getNetSize()`: permet de connaître la taille totale du graphe pour les algorithmes qui ont besoin de cette connaissance.

Puisque les communications entre les processeurs sont basées sur des messages, les fonctions pour manipuler les messages sont fournies. Un message est programmé par une classe qui contient toutes les informations indispensables. Le programmeur peut manipuler un message par les méthodes suivantes:

- `sendTo(voisin, message)`: (resp. `sendAll(message)`) envoie un message à un voisin particulier (resp. tous ses voisins),
- `receiveFrom(int)`: (resp. `receive(Port)`) reçoit un message d'un voisin particulier (resp. le premier message dans la file d'attente du noeud et met dans Port le numéro du port de l'émetteur). D'autres méthodes existent pour manipuler les messages d'un type particulier, i.e. les messages du type entier, chaîne de caractères ou vecteurs.
  Rappelons que les messages sont stockés dans la file d'attente du récepteur, l'implémentation permet aussi de manipuler les messages qui arrivent sur un canal particulier.

Par exemple, le pseudo-code qui permet de programmer le calcul d'un arbre recouvrant du système décrit dans la section 2.1 est donné par l'algorithme 4.

**algorithme 4** : Calcul d'un arbre recouvrant dans un réseau

```
while (run) {
   neighbour = rendezVous();    // Synchronisation.
   sendTo(neighbour,myLabel);        // Echange des états.
   neighbourLabel=receiveFrom(neighbour);
   if (myLabel == 'N') && (neighbourLabel == 'A'){  // Exécution de la règle.
        myLabel = 'A';
        edge[neighbour]=1
        }
   breakSynchro();    // Fin de la synchronisation.
}
```

Beaucoup d'autres algorithmes distribués décrit par les calculs locaux sont déjà implémentés et peuvent être animés [4, 19]. Citons par exemple les algorithmes suivants:

- élection dans les arbres, les graphes triangulés et graphes complets,
- Rendez-vous probabiliste et élections locale probabilistes,
- arbre recouvrant dans des réseaux avec identités,
- l'algorithme de Mazurkiewicz de numérotation et de reconstruction de graphe (universal graph reconstruction),
- détections de propriétés stables,
- l'algorithme de Chang-Robert,
- la 3-coloration d'un anneau,
- l'algorithme de Dijkstra-Scholten pour la détection de terminaison,
- l'algorithme de Dijkstra-Feijen-Van Gasteren pour la détection de la terminaison,
- l'algorithme de Ricart-Agrawala pour l'exclusion mutuelle,
- l'exclusion mutuelle distribuée dans les arbres et les graphes quelconques,
- la résolution distribuée de conflits,
- les synchronizers.

## 5 Conclusion

Nous avons proposé une méthode générale pour l'exécution et la visualisation de l'exécution des algorithmes distribués. Ce travail est motivé par les résultats théoriques importants sur l'utilisation des calculs locaux sur des graphes pour coder les algorithmes distribués. L'interface de Visidia permet à l'utilisateur d'éditer un réseau et d'animer l'exécution d'un algorithme distribué. Visidia peut être également exécuté sur un réseau de machines distinctes. Dans ce cas, les sommets du graphe, et par conséquent les Threads, peuvent être situés sur des machines différentes. Nous avons utilisé la bibliothèque RMI de Java pour étendre la distribution de Visidia [6]. Nous avons utilisé la version distribuée de Visidia pour effectuer des expérimentations sur les graphes de grande taille. Par exemple, nous avons exécuté un algorithme d'élection sur un arbre de 12000 noeuds. D'autres paradigmes comme la terminaison [8] ou la tolérance au fautes [10] peuvent être exprimés dans notre modèle.

## Références


[1] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th Symposium on theory of computing*, pages 82–93, 1980.

[2] M. Bauderon, S. Gruner, Y. Métivier, M. Mosbah, and A. Sellami. Visualization of distributed algorithms based on labeled rewriting systems. In *Second International Workshop on Graph Transformation and Visual Modeling Techniques, Crete, Greece*, July 12-13, 2001.

[3] M. Bauderon, S. Gruner, and M. Mosbah. A new tool for the simulation and visualization of distributed algorithms. Technical Report 1245-00, LaBRI, 2000. Accepted in MFI'01, Toulouse, 21-23 May 2001.

[4] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. From local computations to asynchronous message passing systems. Technical Report RR-1271-02, LaBRI, 2002.

[5] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[6] B. Derbel and M. Mosbah. Distributing the execution of a distributed algorithm over a network. In *7th International Conference on Information Visualization (IV03), IEEE, London*, 16-18 July 2003.

[7] E.W. Dijkstra. Cooperating sequential processes. In *Programming Languages*, pages 43–112. Academic Press, 1968.

[8] E. Godard, Y. Métivier, M. Mosbah, and A. Sellami. Termination detection of distributed algorithms by graph relabelling systems. In 2002 LNCS, editor, *First International Conference on Graph Transformation Barcelona (Spain)*. Springer-Verlag, October 7-12, 2002.

[9] R. Gupta, S. A. Smolka, and S. Bhaskar. On randomization in sequential and distributed algorithms. *ACM Comput. sur.*, 26(1):7–86, 1994.

[10] B. Hamid and M. Mosbah. Détection de pannes dans un système distribué par échange local de messages. In *Journées Scientifiques Francophones, JSF 2003, Tozeur 20-22 décembre*, 2003.

[11] M. Himsolt. Gml — graph modelling language, university of passau, germany, 1997. *http://infosun.fmi.uni-passau.de/Graphlet/GML/*.

[12] D. J. Lehmann and M. O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–138, 1981.

[13] I. Litovsky, Y. Métivier, and E. Sopena. Different local controls for graph relabelling systems. *Math. Syst. Theory*, 28:41–65, 1995.

[14] I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In H. Ehrig, H.J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation*, volume 3, pages 1–56. World Scientific, 1999.

[15] J. Ramirez-Alfonsin M. Habib, C. McDiarmid and B. Reed, editors. *Probabilistic Methods for Algorithmic Discrete Mathematic*. Springer-Verlag, 1998.

[16] A. Sellami M. Mosbah and A. Zemmari. Résolution distribuée de conflits dans un réseau par les systèmes de réécritures. In *Journées Scientifiques Francophones, JSF 2003, Tozeur 20-22 décembre*, 2003.

[17] Y. Métivier, N. Saheb, and A. Zemmari. Randomized rendezvous. In *Mathematics and computer science : Algorithms, trees, combinatorics and probabilities*, Trends in mathematics, pages 183–194. Birkhäuser, 2000.

[18] Y. Métivier, N. Saheb, and A. Zemmari. Randomized local elections. *Inform. Proc. Letters*, 82:313–120, 2002.

[19] M. Mosbah and A. Sellami. Visidia: A tool for the VIsualization and SImilation of DIstributed Algorithms. *http://www.labri.fr/visidia/*.

[20] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 2000.

[21] J. L. Welch and N. A. Lynch. A modular drinking philosophers algorithm. In *Distributed Computing*, volume 6(4), pages 233–244, 1993.