Présentation ViSiDia

Laurent PHILIPPE

Master 2 Informatique
Ingénierie des Systèmes et Logiciels

2020/2021

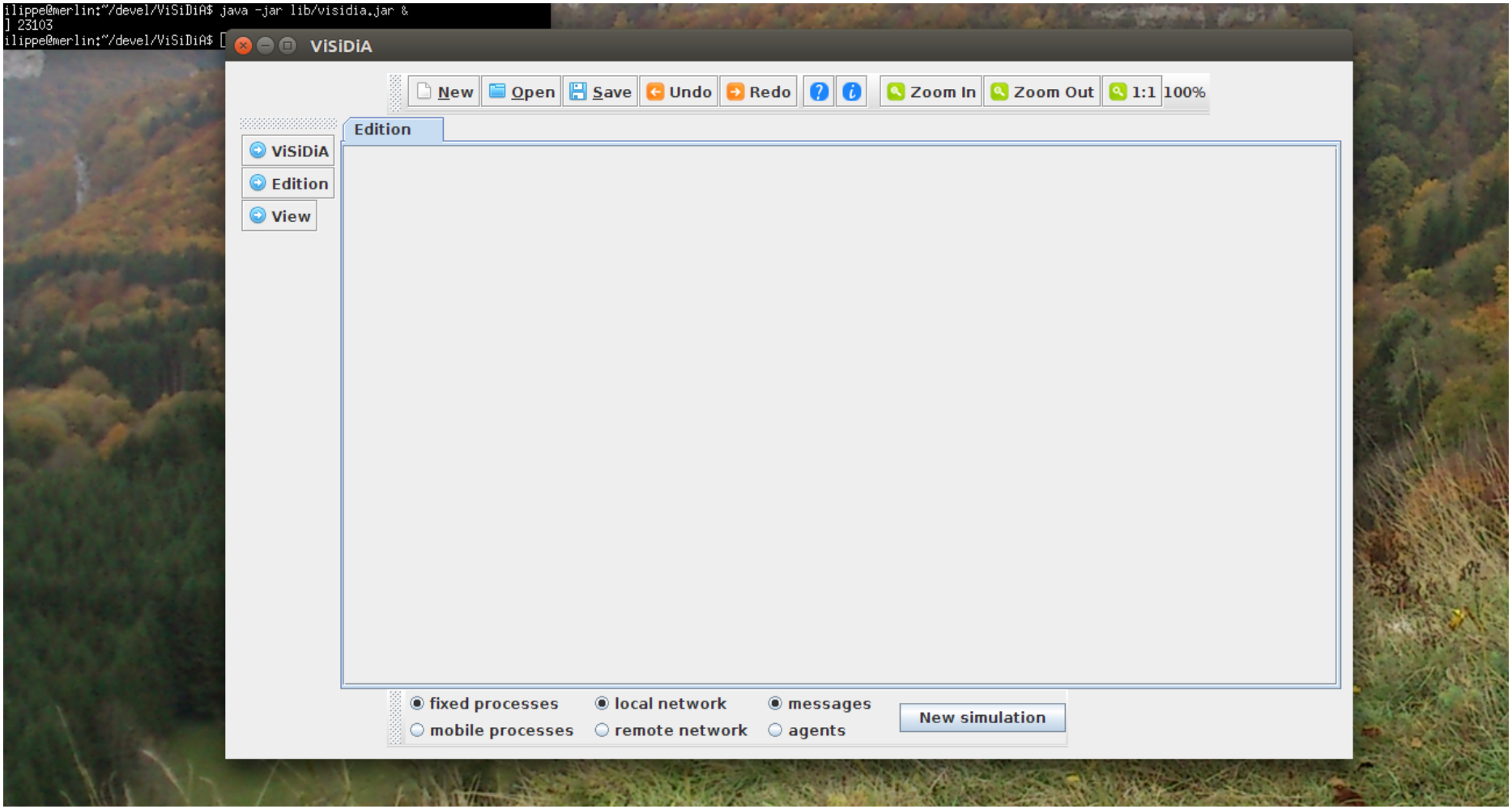
Présenation

- Simulateur de systèmes distribués
- Systèmes synchrones, asynchrones, agents
- B2Visidia
- Développé au Labri : http ://visidia.labri.fr

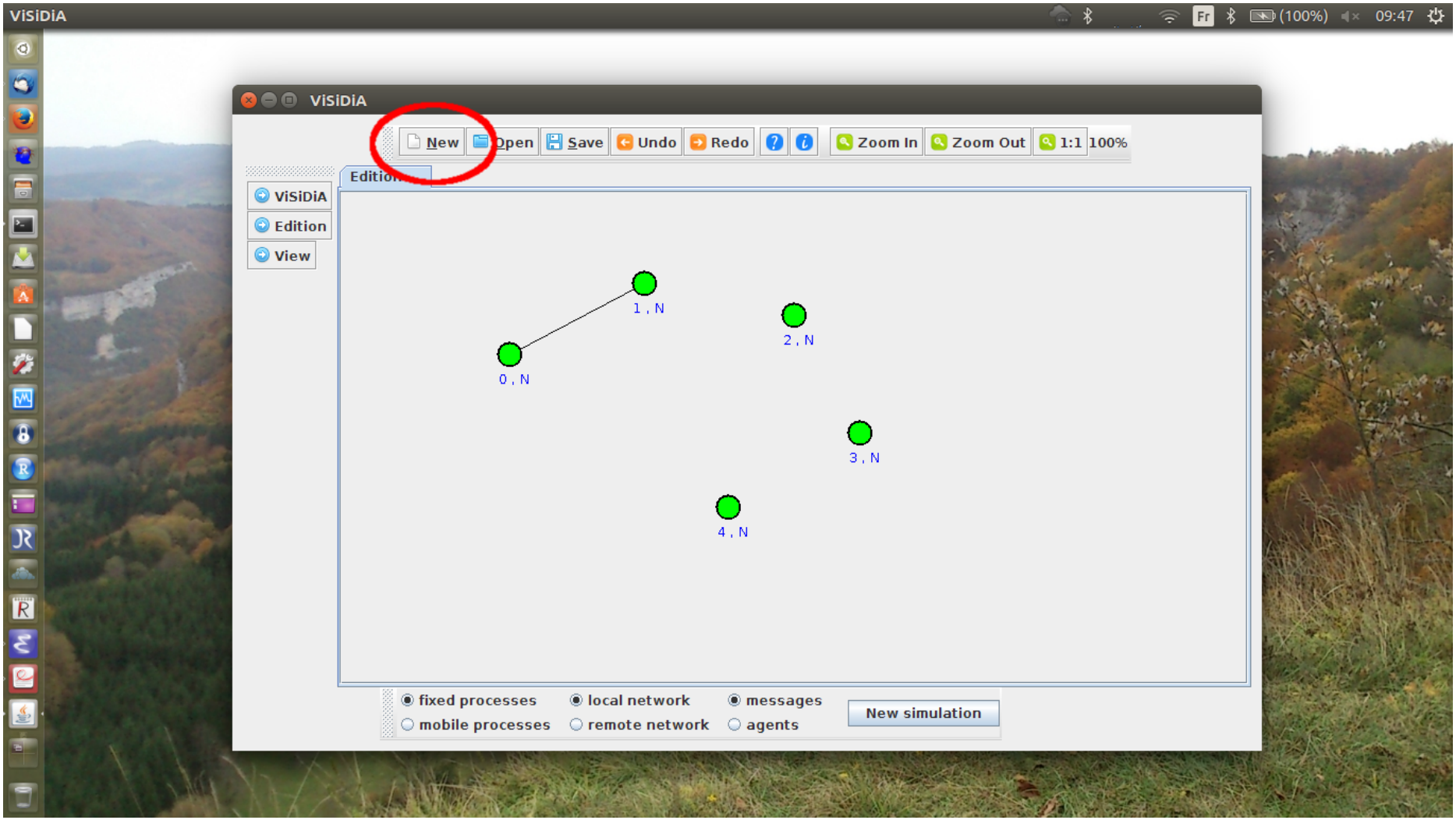
Principe

- Programmation algorithme : Java
- Bibliothèque : `visidia.jar`, `visidia_lib.jar`
- Interface graphique :
  - Définition du réseau
  - Animation/Simulation algorithme

Interface graphique



Définition réseau



Programmation algorithme

Classes de base

- `Class Algorithm` : code de chaque processus
- `Class Message` : définition des messages
- `Class Door` : boîte d'envoi/réception messages

Algorithmes : `Class Algorithm`

Méthodes à implémenter : `@Override`

- `clone()` : création de l'objet
- `getDescription()` : informations sur l'algorithme
- `init()` : code principal du processus

Méthodes disponibles

- `getArity()` : nombe de voisins
- `getId()` : identificateur du processus
- Envoi/Réception des messages

Messages : Class Message

Fonctions à implémenter : @Override

- `clone` : création du message, pour éviter qu'émetteur et récepteur possèdent la même référence
- `getData` : texte de description
- `toString` : chaîne représtant le message, affichée par le simulateur
- Constructeur

Méthodes de la classe

- Type :
  - Type `MsgType` : type associé au message
  - `getType()` : obtenir le type
  - Définir les types de messages `enum`, par ex : `REQ, ACK`. Dans ce cas, tous les messages ont les mêmes données associées
- Définir différentes classes par type de message pour avoir des données indépendantes

Door : Class Door

Utilisation

- Entité pour envoyer/recevoir les messages
- Chaque porte possède un numéro au sein du processus
- `getNum()` : obtenir le numéro de la porte
- Permet principalement de savoir sur quelle porte un message a été reçu.

Communication

Réception des messages

- `receive( Door d )` : recevoir le premier message qui arrive au processus. Fonction bloquante, la porte est celle sur laquelle le processus reçoit le message (paramètre de sortie).
- `receiveFrom( int door )` : recevoir le premier message qui arrive au processuss sur la porte donnée en paramètre.

Envoi des messages

- `sendAll( Message msg )` : envoi le message à tous les voisins du processus
- `sendTo(int Door, Message msg )` : envoi le message à la porte donnée en paramètre

Exemple - I

Premier programme

- Envoyer un message `ACK` à chacun des voisins
- Recevoir un message `ACK` de chacun des voisins

Exemple - I

Message

```
class AckMessage extends Message {

  String type;
  int    value;

  public AckMessage( String s, int n) {
    type = s;
    value = n;
  }

  @Override
  public Message clone() {
    return new AckMessage("",0);
  }

  @Override
  public String toString() {
    String display = new String(type + " " + String.valueOf(value));
    return (display);
  }

  @Override
  public String getData() {
    return (this.toString());
  }

}
```

Exemple - I

Message

```
public class Premier extends Algorithm {

  @Override
  public Object clone() {
    return new Premier();
  }

  @Override
  public void init() {

    int nbNeighbors = getArity();
    Door d = new Door();

    for ( int i = 0 ; i < nbNeighbors; i++) {
      AckMessage sm = new AckMessage("ACK", getId());
      sendTo(i, sm);
    }

    AckMessage m = (AckMessage) receive(d);
    System.out.println("Node " + getId() + " recv " +
                       m.toString() + " from " + d.getNum());
  }
}
```

Exemple - I

Compilation

- Positionner `$CLASSPATH` : `visidia_api.jar`
- Compiler le programme : `javac *.java`

Simulation

- Lancer ViSiDia : `java -jar visidia.jar`
- Créer un réseau
- `New simulation`
- Régler la vitesse
- Charger l'aglorithme : `Simulation` $\rightarrow$ `Select algorithm`
- `Start`
- `Pause` / `Stop`

Implanter un algorithme d'AD

Principe

- Données : définitions locales à l'algorithme
- Messages : définir les différents types soit avec `MessageType`, soit avec des classes différentes
- Fonction `init()` de la classe `Algorithm` : règle principale

Difficultés

- Mise en place des règles, pas direct car la fonction de réception est bloquante
- Gestion des attentes
- Affichage du contenu des processus
- Contrôler la vitesse d'exécution

Définition des règles

Mise en place d'un listener

- Créer un nouveau thread
- Attente sur réception
- Appel des fonctions (régles) correspondantes

Code algorithme

```
@Override
public void init() {
  ReceptionRules rr = new ReceptionRules( this );
  rr.start();
  ...
}
```

Code réception des messages

ReceptionRules

```
public class ReceptionRules extends Thread {
  LamportMutualExclusion algo;
  public ReceptionRules( LamportMutualExclusion a ) { algo = a; }
  public void run() {
    Door d = new Door();
    while( true ) {
      SyncMessage m = (SyncMessage) algo.recoit(d);
      int door = d.getNum();
      switch (m.getMsgType()) {
        case REQ :
        algo.receiveREQ( m.getMsgClock(), m.getMsgProc(), door );
        break;
        case ACK :
        algo.receiveACK( m.getMsgClock(), m.getMsgProc(), door );
        break;
        case REL :
        algo.receiveREL( m.getMsgClock(), m.getMsgProc(), door );
        break;
        default:
        System.out.println("Error message type");
      }
    }
  }
}
```

Gestion des attentes

Solution

- Fonctions synchronized
- `wait()` / `notify()`
- Déblocage grâce au thread de réception

Gestion des attentes sur condition

Attente

```
while( !meTurn ) {
  boolean meTurn = true;
  for(int i = 0; i < nbNeighbors +1 ; i++ ) {
    if ( (F_H[i] < F_H[procId]) || ((F_H[i] == F_H[procId]) && (i < procId)) )
    meTurn = false;
  }
  if ( !meTurn ) { try { this.wait(); } catch( InterruptedException ie) {} }
}
```

Réveil

```
synchronized void receiveACK( int H, int p, int d ){
  if ( clock < H ) clock = H;
  clock = clock + 1;
  if ( F_M[p] != MsgType.REQ ) {
    F_H[p] = H;
    F_M[p] = MsgType.ACK;
  } this.notify();
}
```

Affichage des données

Solutions

- Affichage des données des messages : `toString()`
- Affichage de l'état des processus, pas pris en charge par ViSiDiA
- Faire la même chose que pour les messages :
  - Fonction d'affichage dans le processus
  - Créer des `Frames`
  - Appel de la fonction d'affichage lorsque l'état du processus change : `display`
- Affichage possible dans la fenêtre de lancement de ViSiDiA

Affichage de l'état du processus

JFrame Java

```
public class DisplayFrame {
  JFrame frame;
  JTextArea textArea;
  int size;
  public DisplayFrame(int proc) {
    frame = new JFrame("Process " + proc);
    frame.setAlwaysOnTop (true);
    frame.setLocation(200+100*proc,200+100*proc);
    JPanel panel = new JPanel();
    frame.getContentPane().add(panel);
    // Create initial text area
    String empty = "     "; size = empty.length();
    // The size of the text area must be adapted here
    textArea = new JTextArea(empty, 10, 20);
    textArea.setEditable(false);
    panel.add(textArea);
    //Display the window.
    frame.pack();
    frame.setVisible(true);
  }
  public void display(String s) {
    int nextSize = s.length();
    textArea.replaceRange(s, 0, size);
    size = nextSize;
  }
}
```

Affichage de l'état du processus

Dans l'algorithme

```
@Override
public void init() {
  ...
  df = new DisplayFrame( procId );
  ... ; displayState();
  ... ; displayState();

}
```

Fonction displayState()

```
void displayState() {
  String state = new String("Clock = " + clock + "\n");
  if ( inCritical )
  state = state + "** ACCESS CRITICAL **";
  else if ( waitForCritical )
  state = state + "* WAIT FOR *";
  else
  state = state + "-- SLEEPING --";
  df.display( state );
}
```

Contrôle de la vitesse d'exécution

Problèmes

- Simuler le temps d'exécution des processus : temps de travail, temps en section critique, ...
- ViSiDiA contrôle la vitesse de transmission des messages sur les liens :
  - La vitesse d'exécution des processus n'est pas impactée par le curseur de vitesse
  - Difficile à réaliser, sinon programme interprété
- Ajouter des temporisations

Solution

```
try { Thread.sleep( attente ); } catch( InterruptedException ie ) {}
```

Desynchronisation des processus

Problème

- Tous les processus font tous la même chose au même moment :
  - Même vitesse d'exécution
  - Vitesse uniforme sur les liens
- Observation difficile

Solution

```
Random rand = new Random( procId );
time = (1 + rand.nextInt(2)) * 1000;
System.out.println("Process " + procId + " enter SC " + time);
try {
  Thread.sleep( time );
} catch( InterruptedException ie ) {}
```

Types des messages - cas 1

Tous les messages ont les mêmes données

- Exemple : Lamport, tous les messages contiennent l'horloge
- Définition d'une seule classe de messages
- Identification du type à l'aide d'un `enum`
- Données associées dans l'objet
- Setter / Getter

Types des messages - cas 1

Implementation enum

```
public enum MsgType {
  REQ,
  ACK,
  REL;
}
```

Classe `MessageType`

- Attention ViSiDiA possède aussi un type de message spécifique : `MessageType`. A ne pas confondre avec notre définition
- Obtenu avec la fonction `MessageType getType()`
- Méthodes gèrent les propriétés d'affichage des messages

Types des messages - cas 1

Implementation Message

```
import visidia.simulation.process.messages.Message;
public class SyncMessage extends Message {
  MsgType type;
  int clock;
  int proc;

  public SyncMessage( MsgType t, int c, int p ) { type = t; clock = c; proc = p; }

  public MsgType getMsgType() { return type; }
  public int getMsgClock() { return clock; }
  public int getMsgProc() { return proc; }

  @Override public Message clone() { return new SyncMessage( MsgType t, int c, int p ); }
  @Override public String toString() {
    String r = type.toString() + "_" + clock + "_" + proc;
    return r;
  }
  @Override public String getData() { return this.toString(); }
}
```

Types des messages - cas 1

Création Messages

```
SyncMessage sm = new SyncMessage(MsgType.REQ, clock, procId);
```

Utilisation Messages, identifier le type

```
SyncMessage m = (SyncMessage) algo.recoit(d);
int door = d.getNum();

switch (m.getMsgType()) {

  case REQ :
  algo.receiveREQ( m.getMsgClock(), m.getMsgProc(), door );
  break;
}
```

Types des messages - cas 2

Tous les messages ont données différentes

- Exemple : Suzuki-Kazami
  - Message `REQ` : horloge
  - Message `TOKEN` : tableau des accès

Solution

- Tous les messages possèdent toutes les données : même implémentation que précédemment
- Messages avec des données différentes :
  - Une classe par type de message
  - Chaque classe possède ses propres données + Setter/Getter
- Pas de définition de type

Types des messages - cas 2

Implementation des classes de messages

```
import visidia.simulation.process.messages.Message;
public class ReqMessage extends Message {
  int clock;
  int proc;

  public ReqMessage( int h, int p ) { clock = h; proc = p; }
  public int getMsgClock() { return clock; }
  public int getMsgProc() { return proc; }

  @Override public Message clone() { return new ReqMessage( clock, proc); }
  @Override public String toString() { String r = "REQ_" + clock; return r; }
  @Override public Strigng getData() { return this.toString(); }
}
```

Types des messages - cas 2

Implementation des classes de messages

```
import visidia.simulation.process.messages.Message;
public class TokenMessage extends Message {
  int[] tokenTab;
  int proc;

  public TokenMessage( int[] tt, int p ) { tokenTab = tt;   proc = p; }
  public int[] getTokenTab() { return tokenTab; }
  public int getMsgProc() { return proc; }

  @Override public Message clone() { return new TokenMessage( tokenTab, proc ); }
  @Override public String toString() { String r = "TOKEN_" + proc; return r; }
  @Override public String getData() {  return this.toString(); }
}
```

Types des messages - cas 2

Création Messages

```
ReqMessage rm = new ReqMessage(H, procId);
TokenMessage tm = new TokenMessage(TokenMsg, procId);
```

Utilisation Messages, identifier le type

```
Message m = algo.recoit(d);
int door = d.getNum();

if ( m instanceof ReqMessage ) {
  ReqMessage rm = (ReqMessage) m;
  algo.receiveREQ( rm.getMsgClock(), rm.getMsgProc(), door );
} else if ( m instanceof TokenMessage ) {
  TokenMessage tm = (TokenMessage) m;
  algo.receiveTOKEN( tm.getTokenTab(), tm.getMsgProc() );
} else {
  System.out.println("Error message type");
}
```

Accès aux voisins

Exemples

- Lelann : accès au successeur
- Suzuki-Kasami : envoi au premier processus qui a une valeur de demande supérieure à la date de sa dernière visite
- Naimi-Tréhel : envoi au `next`

Problème

- Processus accède à ses voisins en envoyant des messages
- Les messages ne sont pas envoyés directement mais plutôt à partir des portes
- Une porte ne dispose pas du numéro de processus (`procId()`) à qui elle donne accès
- La numérotation des portes dépend de l'ordre dans lequel on dessine le réseau

Accès aux voisins

Solution 1, cas de l'anneau

- Dessiner un anneau dans l'ordre, puis utiliser toujours la porte 1 pour envoyer
- Que se passe-t-il si nous n'avons pas dessiné le réseau ? Cas du processus 0 ?
- Processus de l'anneau ne possèdent que 2 portes : recevoir sur l'une, envoyer sur l'autre

Solution 1, cas de l'anneau

```
int next = 0;
...
// Start token round
if ( procId == 0 ) {
  next = 1; // depends on the network drawing
  TokenMessage tm = new TokenMessage(MsgType.TOKEN);
  boolean sent = sendTo( next, tm );
}
...
// Rule 2 : receive TOKEN
synchronized void receiveTOKEN(int d){

  System.out.println("Process " + procId + " reveiced TOKEN from " + d );
  next = ( d == 0 ? 1 : 0 );

  if ( waitForCritical == true ) {

    token = true;
    displayState();
    notify();

  } else {
    // Forward token to successor
    TokenMessage tm = new TokenMessage(MsgType.TOKEN);
    boolean sent = sendTo( next, tm );
  }
}
```

Accès aux voisins

Solution 2, envoi à un prosssus spécifique

- Mise en place d'une table des voisins : correspondance porte - numéro de processus
- Initialisation dynamique :
  - Dans Suzuki-Kasami un processus qui demande la section critique envoie une requête à tous ses voisins.
  - Lorsqu'on envoie le jeton c'est donc que nous avons déjà reçu la requête de demande.
  - Suppose de mémoriser le numéro du processus dans chaque message
  - Pas de message supplémentaire
  - Dépend de l'algorithme
- Initialisation statique :
  - Phase d'intialisation
  - Envoie un message spécifique sur chaque porte (`WHOIS`)
  - Réponse des voisins avec leur numéro
  - A la réception de la réponse mémorisation dans le tableau

Tableau des voisins : solution dynamique

Mémorisation des voisins

```
int[] neighborDoors;
nbNeighbors = getArity();
neighborDoors = new int[nbNeighbors+1];
...
// Rule 3 : receive REQ( H )
synchronized void receiveREQ( int H, int p, int d){
  ...
  // records corresponding door
  neighborDoors[p] = d;
  ...
}
// Rule 5 : exit critical section
void endCriticalUse() {
  ...
  boolean sent = sendTo( neighborDoors[ next ], tm );
  ...
}
```

Remarques :

- Pas besoin d'initialiser puisque le processus reçoit toujours `REQ` avant d'envoyer le Token
- Mémorisation à chaque réception vs. booléen `init`

Tableau des voisins : solution statique

Classe Neighbors

```
import java.util.ArrayList;
public class Neighbors {
  private ArrayList<Neighbor> neighbors = null;
  public Neighbors(int numberDoors) {
    this.neighbors = new ArrayList<Neighbor>();
    for (int i = 0; i < numberDoors; ++i) { this.neighbors.add(new Neighbor(i)); }
  }
  public int getDoorForNeighbor(int numNeighbor) {
    for (Neighbor neighbor : neighbors) {
      if (neighbor.getProcId() == numNeighbor) { return neighbor.doorNum(); }
    }
    return -1;
  }
  public void setDoorForNeighbor(int numDoor, int numNeighbor) {
    for (Neighbor neighbor : neighbors) {
      if (neighbor.doorNum() == numDoor) { neighbor.setProcId(numNeighbor); }
    }
  }
  public int getNumberNeighbor() { return neighbors.size(); }
}
```

Classe Neighbor

La classe Neighbor est définie par ViSiDiA.

Tableau des voisins : solution statique

Définition du type de message

```
public enum MsgType {
  JETON,
  REQ,
  WHOIS,
}
```

Défintion du message associé

```
public class SyncMessage extends Message {
  private MsgType type;
  ...
  public SyncMessage(MsgType type, int procId) {
    this.type = type;
    ...
  }
}
```

Tableau des voisins : solution statique

Interrogation des voisins

```
public class SuzukiKazamiAlgorithm extends Algorithm {
  private Neighbors neighbors = null;
  private int neighborAnswer = 0;
  ...
  synchronized public void initNeighbors() {
    for (int i = 0; i < neighbors.getNumberNeighbor(); ++i) {
      sendTo(i, new SyncMessage(MsgType.WHOIS, procId));
    }
    while (neighborAnswer < neighbors.getNumberNeighbor()) {
      try { this.wait(); } catch (InterruptedException e) { ... }
    }
    return;
  }
}
```

Tableau des voisins : solution statique

Réponses des voisins

```
SyncMessage message = algo.receiveMessage(door);
switch (message.getMsgType()) {
  case WHOIS: algo.setupDoor(door.getNum(), message.getMsgProc());
  break;
  ...
}
```

Enregistrement de la porte

```
synchronized public void setupDoor(int numDoor, int numNeighbor) {
  neighbors.setDoorForNeighbor(numDoor, numNeighbor);
  neighborAnswer++;
  this.notifyAll();
}
```

Pour la suite...

Documentations

- Manuel d'utilisation de ViSiDiA : disponible sous `/applis/masterad/ViSiDiA`
- Javadoc : `http://visidia.labri.fr/javadoc_api/index.html`
- Site du projet : `http://visidia.labri.fr`

Utilisation

- Copier les fichiers : `visidia.jar` et `visidia_lib.jar`
- Version Java ≥ 6