Type document : *SRS*	*Projet RNTL MORSE*	**Date :** 12/06/2003
Sous-projet n° 2		**numéro de doc :** MORSE-DRAFT-040610-V0.01-FGI
tâche : 2.1		**rédacteur (s)** Frédéric Gilliers

EXEMPLE DE GENERATION DE CODE A PARTIR DE L*f*P

Table des matières

1 Suivi des modifications

date	Objet de la modification
10/06/2004	Création

2 Introduction

L'objectif de de ce document est de fournir un premier apperçu de la génération de code à partir de **L*f*P**. Il s'articule autour de l'exemple simplifié d'une application de *chat*.

La présentation de l'exemple sera faite en trois étapes : la section 3 présentera de manière informelle l'application à réaliser ; la section 4 présentera la spécification **L*f*P** déduite de la spécification informelle ; enfin, la section 5 présentera le code généré à partir de la spécification **L*f*P** ainsi que le code écrit manuellement pour implémenter les composants externes.

3 Présentation de l'exemple proposé

Cette section est dédiée à la présentation de l'exemple étudié dans ce document. Cette présentation est faite en deux étapes : une première étape présente l'exemple de manière textuelle, la seconde propose une présentation plus précise axée sur la notation UML.

3.1 Description de l'exemple

L'exemple choisi pour ce document est une application de *chat* simplifiée. Cette application doit donc gérer une conversation entre un nombre variable de personnes, ce qui signifie que l'application doit implémenter les trois fonctionnalités suivantes :

- accepter l'arrivée de nouveaux participants ;
- gérer le départ des participants enregistrés ;
- diffusion des messages de chaque participant à la conversation vers tous les participants.

Pour sélectionner la fonctionnalité dont il a besoin, un utilisateur dispose des quatres commandes suivante :

- *join* qui permet de s'enregistrer pour participer à une conversation ;
- *quit* qui permet de se retirer d'une conversation en cours ;
- *send* qui permet d'envoyer un message à tous les participants ;
- *nickname* qui permet d'associer un prefixe à tous les messages qui seront envoyés par la suite (permet d'identifier l'expéditeur du message), les messages de ce type doivent être ignorés et ne sont pas diffusés aux clients.

Afin de ne pas surcharger inutilement le modèle, on ne gèrera pas les cas suivants :

- utilisateur essayant de diffuser un message sans s'être préalablement enregistré ;
- interruption de communication (utilisateur oubliant de se retirer d'une conversation ou coupure de communication).

3.2 Description UML de l'exemple

La description UML est découpée en deux étapes : la description statiques basée sur le diagramme de classe de l'exemple, puis la description du protocole d'interaction entre les composants basée le diagramme de séquence.

3.2.1 Description statique

la figure 1 fournit le diagramme de classe[1] de l'application de chat.

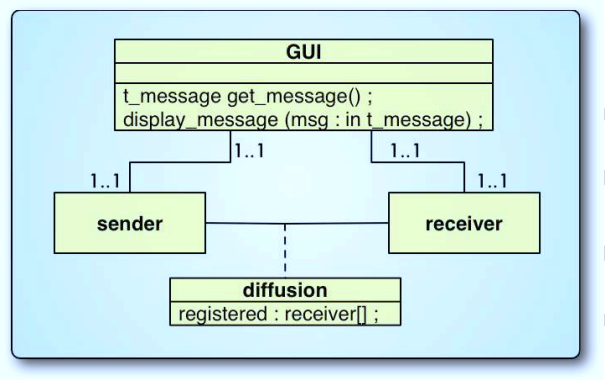

FIG. 1 – Diagramme de classe de l'exemple

La classe d'interaction `diffusion` est chargée de représenter la partie serveur de l'application. C'est elle qui sera responsable de la gestion des ajouts et des retraits des utilisateurs, ainsi que de la diffusion des messages.

La classe `sender` est chargée de l'envoi des messages à au serveur de diffusion. Le message doit être fourni par l'interface graphique (classe `GUI`).

La classe receiver est chargée de la réception des messages envoyés par le serveur de diffusion. Elle doit transmettre le message à l'interface graphique pour affichage.

La classe `GUI` représente l'interface graphique de l'application. Cette classe a donc deux rôles :

- fournir le prochain message à envoyer au serveur de diffusion ;
- afficher les messages envoyés par le serveur de diffusion ;

Ces deux rôles sont respectivement assurés par les méthodes `get_message` et `display_message`.

Un client participant à la conversation est défini comme suit :

- une instance de `sender` pour les messages sortants ;
- une instance de `receiver` pour les messages entrants ;
- une instance de `GUI` pour l'interaction avec l'utilisateur.

3.2.2 Description du protocole d'interaction

Le serveur de diffusion communique avec ses clients par envoi de message. On doit distinguer trois scénariis :

1. Le cas le plus fréquent : un envoi de message devant être distribué à tous les clients du service de diffusion.
2. Le cas d'un abonnement d'un nouveau client au service de diffusion (commande **join**).
3. Le cas d'un désabonnement d'un client du service de diffusion (commande **quit**).

3.2.3 Scénario de diffusion d'un message

La figure 2 donne les interactions entre les composants du modèle lors de l'envoi d'un message au service de diffusion.

Le scénario se déroule comme suit :

[1] Le diagramme proposé est en UML 1.X, il serait interessant de permettre de le représenter en utilisant le profil **L*f*P** proposé par Aonix

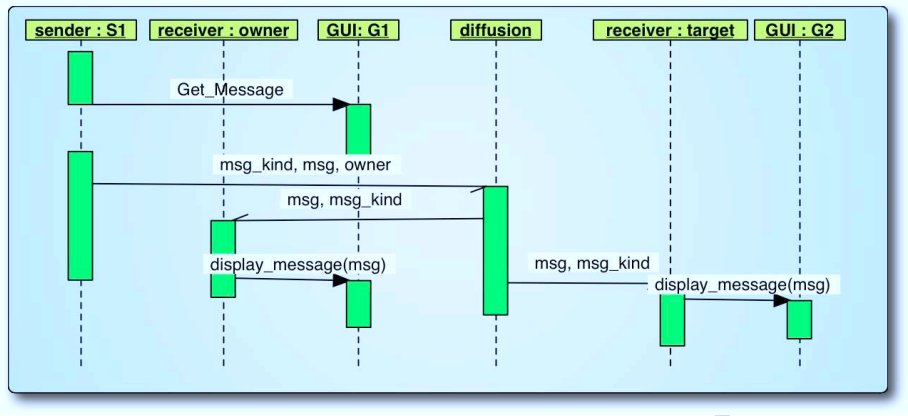

FIG. 2 – Envoi d'un message lors d'une diffusion avec deux clients

- l'émetteur S1 récupère un message depuis l'interface graphique ;
- Il envoie un message au serveur de diffusion contenant la référence du client qui a émis le message, le message à diffuser ainsi que son type (message à diffuser, commande *join* ou command *quit*.
- Le serveur de diffusion transfert le message ainsi que son type à tous les clients connectés.
- Chaque récepteur affiche le message en le transferant à l'interface graphique qui lui est associée.

3.2.4 Ajout d'un client

Lorsque le type du message est `subscribe` (ce qui correspond à la commande *join*, la référence du récepteur associé au site est enregistrée dans la liste des clients connus du service de diffusion. Cette opération est effectuée avant la diffusion du message aux autres clients. Le contenu du message doit les informer qu'un nouveau client participe à la discution.

3.2.5 Retrait d'un client

Lorsque le type du message envoyé au service de diffusion est `unsubscribe` (ce qui correspond à la commande *quit*, le client est retiré de la liste des clients. Puis son message est diffusé à tous les autres clients pour les avertir.

4 Modèle L*f*P de l'application

Cette section présente les diagrammes **L*f*P** permettant de réaliser la partie contrôle de l'application de *chat*. La partie contrôle recouvre les transferts de message, et la gestion de la diffusion des messages entre tous les clients du service.

4.1 Diagramme d'architecture

Le diagramme d'architecture du système est présenté sur la figure 3. Il reprend la structure statique du système et fait apparaitre les éléments qui seront implémentés par la spécification **L*f*P**, et ceux qui seront implémentés par des composants externes.

Les composants implémentés directement en **L*f*P** sont les classes `sender`, `receiver` et `diffusion` du diagramme de classe de la figure 1. Les classes `sender` et `receiver` sont implémentées sous

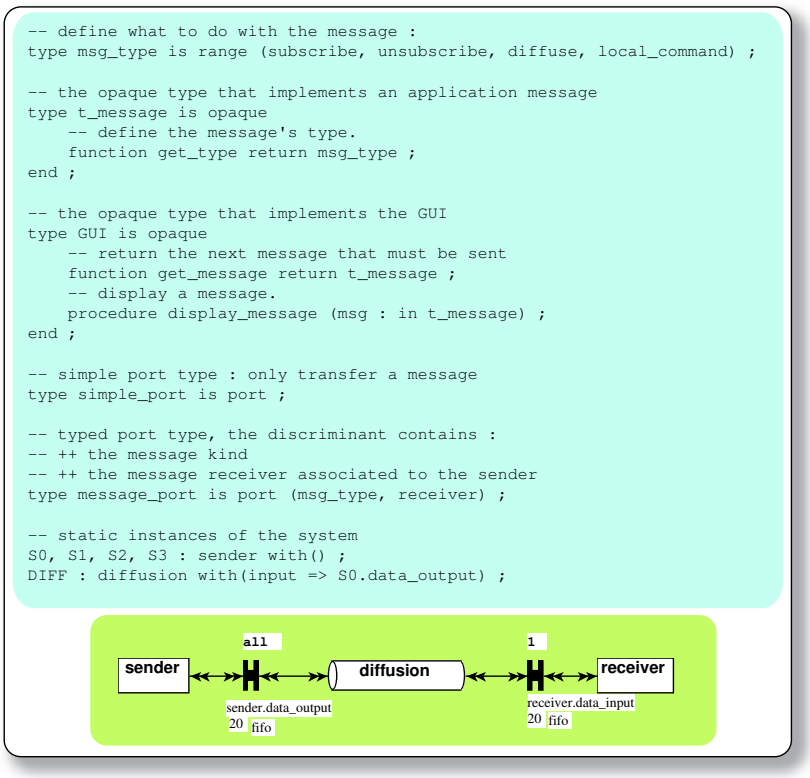

FIG. 3 – Diagramme d'architecture de l'application

la forme de classes **L*f*P**. La classe d'interaction `diffusion` est implémentée sour la forme d'un média qui assure la communication entre les instances de `sender` et de `receiver`.

La classe `GUI` représente l'interface graphique mise à disposition de l'utilisateur. Elle ne peut donc pas être modélisée sous la forme d'une classe **L*f*P**. Elle est donc représentée par un type opaque défini dans la partie déclarative du diagramme d'architecture. Ce `GUI` dispose de la même interface que celle déclarée sur le diagramme de classe.

Par ailleurs, on introduit le type `t_message` qui représente les messages échangés par l'application. Ce type dispose d'une interface simple permettant de déterminer la nature du message à transmettre.

La nature d'un message est définie par le type `msg_kind` qui définit trois valeurs correspondant aux commandes disponibles pour l'utilisateur :

- `subscribe` correspond à une commande *join* ;
- `unsubscribe` correspond à une commande *quit* ;
- `diffuse` corresopnd à un message à diffuser à tous les autres utilisateurs ;
- `local_command` correspond à un message ne devant pas être envoyé au serveur de diffusion.

Initialement, le diagramme d'architecture prévoit quatres instances de la classe `sender` (`S0, S1, S2, S3`), ainsi qu'une instance du média `diffusion` (`DIFF`). Le port de réception du média de diffusion est initialisé pour désigner le binder statique initilisé lors de la création des instances de `sender`.

4.2 La classe `sender`

La classe sender doit gérer l'initilisation du client et la lecture des messages à envoyer au service de diffusion. Son diagramme de comportement est représenté sur la figure 4.

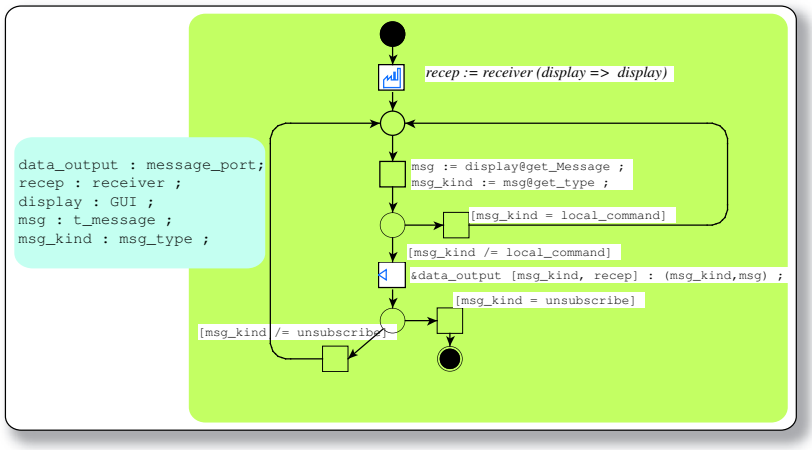

FIG. 4 – Diagramme de comportement de la classe `sender`

Le comportement de cette classe est le suivant :

1. Créer une instance de receiver pour traiter les messages en provenance du service de diffusion ;
2. récupérer le prochain message à envoyer et sa nature ;
3. si le prochain message n'est pas un message local, alors l'envoyer au service de diffusion ;
4. si c'était le dernier message (message de type *unsubscribe*, terminaison du composant ;
5. sinon, retourner au point 2.

Lorsque la classe `sender` veut récupérer le prochain message à envoyer, elle appelle la méthode externe `get_message` sur le composant externe `display`. Cette méthode déclarée dans l'interface du type opaque `GUI` retourne le prochain message à envoyer. Ensuite, pour connaitre le type du message, la classe utilise la méthode `get_type` du type `t_message`.

4.3 La classe `receiver`

La classe receiver s'occupe de la réception des messages en provenance du serveur de diffusion. Le rôle principal de cette classe est de commander leur affichage à l'interface graphique à l'aide de la méthode `display_message`. Le diagramme de comportement de la classe `receiver` est représenté sur la figure 5.

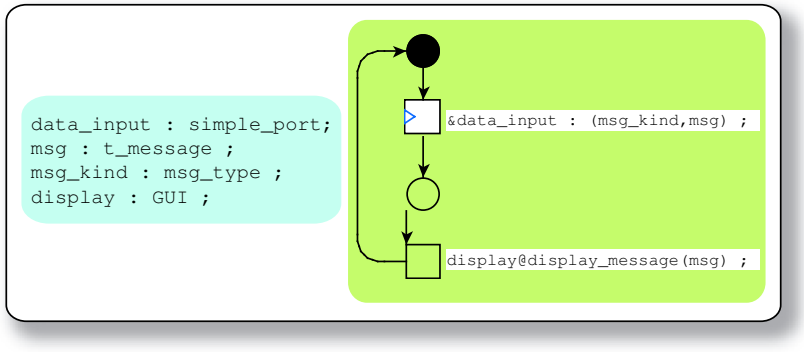

FIG. 5 – Diagramme de comportement de la classe `receiver`

Le comportement de cette classe est relativement simple, il s'agit d'une boucle perpétuelle qui lit les messages en provenance du serveur de diffusion, et demande leur affichage à l'interface graphique.

4.4 Le média `diffusion`

Le média `diffusion` implémente le serveur de diffusion. Son diagramme de comportement est donné sur la figure 6.

La liste des abonnés est conservée dans le tableau `subscribed` défini dans les variables du média. La capacité de ce tableau défini le nombre maximal de composants pouvant être connectés simultanément par une instance de ce média.

Lorsqu'il est instancié, le média de diffusion se met en attente sur le port `input` et attend un message à diffuser. Le message doit être acompagné d'un discriminant dont la structure est la suivante :

– la nature du message (de type `msg_type`) ;
– la référence de l'instance de la classe `receiver` associée au client qui a envoyé le message.

La nature du message déterminera le traitement à appliquer avant sa diffusion à l'ensemble des clients. La référence de l'instance de `receiver` permet de définir le client qui a envoyé le message. Elle est utilisée lorsque l'utilisateur utilise une commande `join` (message `subscribe`) ou *quit* (message `unsubscribe`) pour mettre à jour la liste des clients abonnés.

D'après le comportement de la classe `sender`, trois types de messages peuvent arriver au média diffusion :

– `subscribe` ;
– `unsubscribe` ;
– `diffuse`.

Dans le cas d'un message de type subscribe, le média passe dans la branche la plus à gauche sous l'état `choix_traitement`. Dans ce cas, la référence de la classe `receiver` est utilisée pour désigner le nouveau client qui est ajouté dans le tableau des abonnés. Le message est ensuite diffusé à tous les abonnés pour les avertir de l'arrivée d'un nouveau client.

Dans le cas d'un message de type unsubscribe, le média passe par la branche du milieu sous l'état `choix_traitement`. La position du client dans le tableau des abonnés est recherchée, puis celui-ci est retiré. Enfin tous les clients qui se trouvaient après lui dans le tableau sont décalés d'une case vers la gauche. Enfin, le message est diffusé à tous les abonnés pour les avertir du départ du client.

Dans le dernier cas, le message est de type `diffuse`, et aucun traitement n'est appliqué avant de le diffuser à l'ensemble des abonnés.

Après la boucle de diffusion aux abonnés (branche comprise entre l'état `diffuse` et la transition `TR1`), le média revient dans son état initial (branche à droite sous l'état `diffuse`). Il est alors prêt à traiter un nouveau message.

5 Code java correspondant au modèle

Cette section présente le code java correspondant à l'exemple étudié dans les sections précédentes. Cette section est structurée comme suit :

1. code généré à partir des types de données utilisées dans le modèle **L*f*P** ;
2. code généré à partir des composants **L*f*P** ;
3. code des composants externes.

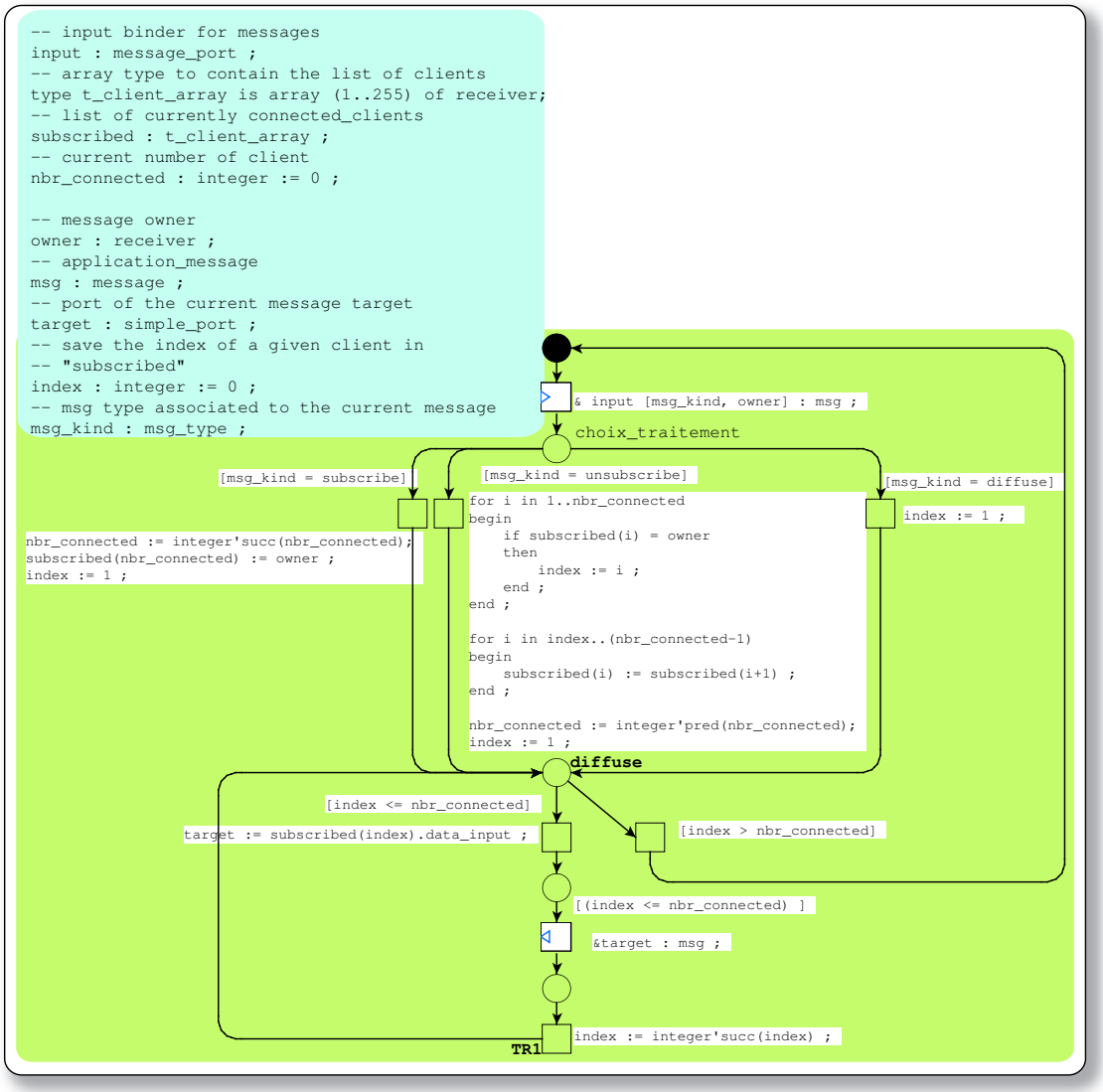

FIG. 6 – Diagramme de comportement du média de diffusion

Dans les deux premiers cas, le code est ré-indenté à la main avant présentation (pour faciliter la lecture).

5.1 Code généré à partir des composants L*f*P

5.1.1 Code de la classe `sender`

```
1  public class SENDER extends LfPClass implements Runnable {
2      static LfPBinderReference DATA_OUTPUT = GlobalDeclarations.
           DATA_OUTPUT ;
3      LfPComponantReference RECEP = null ;
4      GUI DISPLAY = new GUI() ;
5      T_MESSAGE MSG = new T_MESSAGE() ;
6      MSG_TYPE MSG_KIND = new MSG_TYPE() ;
7      /* This class contains no method */
8      /* This class contains no trigger */
```

```
9       /*
10       * main method of the class
11       */
12      public void run ()
13      {
14          try{
15              LfPMessage msg = null ;
16              String next ;
17              String label_4 = "+SENDER+ 15 15";
18              String label_5 = "+SENDER+ 14 14";
19              String label_6 = "+SENDER+ 39 39";
20              String label_7 = "+SENDER+ 23 23";
21              String label_8 = "+SENDER+ 7 7";
22              String label_9 = "+SENDER+ 19 19";
23              String label_10 = "+SENDER+ 18 18";
24              String label_11 = "+SENDER+ 2 2";
25              String label_12 = "+SENDER+ 17 17";
26              String label_13 = "+SENDER+ 32 32";
27              String label_14 = "+SENDER+ 16 16";
28              /* sub_diagram for  */
29              next  = label_11 ;
30              while (true) {
31                  if (next == label_4) {
32                      MSG = (T_MESSAGE) DISPLAY.GET_MESSAGE();
33                      MSG_KIND = new MSG_TYPE(MSG.GET_TYPE().value);
34                      next = label_7;
35                  }
36                  if (next == label_5) {
37                      next = label_4;
38                  }
39                  if (next == label_6) {
40                      next = label_5;
41                  }
42                  if (next == label_7) {
43                      if ( (MSG_KIND.intValue() == new MSG_TYPE( 4).value
                            )) {
44                          next = label_6;
45                      }
46                      if ( (MSG_KIND.intValue() != new MSG_TYPE( 4).value
                            )) {
47                          next = label_14;
48                      }
49                  }
50                  if (next == label_8) {
51                      String[] Attr_List35 = {"DISPLAY"};
52                      Object[] inital_values36 = {DISPLAY};
53                      RECEP = runtime.newInstance("RECEIVER", Attr_List35
                            , inital_values36);
54                      next = label_5;
55                  }
56                  if (next == label_9) {
57                      next = label_13;
58                  }
59                  if (next == label_10) {
60                      next = label_5;
61                  }
```

```
62                  if (next == label_11) {
63                      next = label_8;
64                  }
65                  if (next == label_12) {
66                      if ( (MSG_KIND.intValue() == new MSG_TYPE( 2).value
                            )) {
67                          next = label_9;
68                      }
69                      if ( (MSG_KIND.intValue() != new MSG_TYPE( 2).value
                            )) {
70                          next = label_10;
71                      }
72                  }
73                  if (next == label_13) {
74                      break ;
75                  }
76                  if (next == label_14) {
77                      msg = new LfPMessage();
78                      msg.setBinder(DATA_OUTPUT) ;msg.setDiscriminant(
                            MSG_KIND, 1) ;
79                      msg.setDiscriminant(RECEP, 2) ;
80                      msg.setParameter(MSG_KIND, 1) ;
81                      msg.setParameter(MSG, 2) ;
82                      runtime.sendMessage(msg);
83                      next = label_12;
84                  }
85              }
86          }catch (Exception e) {
87              e.printStackTrace() ;
88              System.exit(1);
89          }
90          try {
91              runtime.removeComponant(this.selfReference) ;
92          }
93          catch (LfPRuntimeError lrExcp) {
94              lrExcp.printStackTrace() ;
95          }
96      }
97  }
```

5.1.2 Code de la classe `receiver`

```
1   public class RECEIVER extends LfPClass implements Runnable {
2       LfPBinderReference DATA_INPUT = runtime.newBinderInstance( 20, "
            FIFO") ;
3       T_MESSAGE MSG = new T_MESSAGE() ;
4       MSG_TYPE MSG_KIND = new MSG_TYPE() ;
5       GUI DISPLAY = new GUI() ;
6       /* This class contains no method */
7       /* This class contains no trigger */
8       /*
9        * main method of the class
10       */
11      public void run ()
12      {
13          try{
```

```
14              LfPMessage msg = null ;
15              String next ;
16              String label_0 = "+RECEIVER+ 13 13";
17              String label_1 = "+RECEIVER+ 10 10";
18              String label_2 = "+RECEIVER+ 3 3";
19              String label_3 = "+RECEIVER+ 2 2";
20              /* sub_diagram for  */
21              next  = label_3 ;
22              while (true) {
23                  if (next == label_0) {
24                      DISPLAY.DISPLAY_MESSAGE(MSG);
25                      next = label_3;
26                  }
27                  if (next == label_1) {
28                      next = label_0;
29                  }
30                  if (next == label_2) {
31                      msg = runtime.getSimpleMessage(DATA_INPUT);
32                      MSG_KIND = (MSG_TYPE) msg.getParameter( 1);
33                      MSG = (T_MESSAGE) msg.getParameter( 2);
34                      next = label_1;
35                  }
36                  if (next == label_3) {
37                      next = label_2;
38                  }
39              }
40          }catch (Exception e) {
41              e.printStackTrace() ;
42              System.exit(1);
43          }
44          try {
45              runtime.removeComponant(this.selfReference) ;
46          }
47          catch (LfPRuntimeError lrExcp) {
48              lrExcp.printStackTrace() ;
49          }
50      }
51  }
```

5.1.3 Code du média `diffusion`

La première partie du code concerne les déclarations de variables et de types faites dans le diagramme principal du média. On pourra notamment remarquer les lignes 3 à 26 qui correspondent au type `t_client_array` qui est le type tableau nécessaire pour stocker l'ensemble des clients connectés. On y trouve l'ensemble des méthodes et constructeurs nécessaires à la manipulation des tableaux **L*f*P**.

La deuxième partie du code correspond à la méthode `run` du média qui correspond à l'automate défini dans la spécification **L*f*P**. Par exemple, l'état `choix_traitement` correspond aux lignes 100 à 110 ; la transition de réception du message correspond aux lignes 134 à 140 du code.

```
1   public class DIFFUSION extends LfPMedia implements Runnable {
2       LfPBinderReference INPUT = null ;
3       private class T_CLIENT_ARRAY implements java.io.Serializable {
4           int first1 =  1;
5           int last1 =  255;
6           LfPComponantReference array[] = new LfPComponantReference[last1
```

```
            -first1+1];
7           public void initialize(LfPComponantReference value)
8           {
9               for (int l1 = 0 ; l1 <= last1- first1 ; l1++) {
10                  array[l1] = value ;
11              }
12          }
13          public T_CLIENT_ARRAY( T_CLIENT_ARRAY value) {
14              for (int l1 = 0 ; l1 <= last1- first1 ; l1++) {
15                  this.array[l1] = value.array[l1];}
16          }
17          public T_CLIENT_ARRAY()
18          {
19              super ();
20              for (int l1 = 0 ;
21                   l1 <= last1 - first1 ;
22                   l1++) {
23                  this.array[l1] = new LfPComponantReference() ;
24              }
25          }
26      }
27
28
29      T_CLIENT_ARRAY SUBSCRIBED = new T_CLIENT_ARRAY() ;
30      INTEGER NBR_CONNECTED = new INTEGER( 0);
31      LfPComponantReference OWNER = null ;
32      LfPMessage MSG = new LfPMessage() ;
33      LfPBinderReference TARGET = null ;
34      INTEGER INDEX = new INTEGER( 0);
35      MSG_TYPE MSG_KIND = new MSG_TYPE() ;
36      /* This class contains no method */
37      /* This class contains no trigger */
38      /*
39       * main method of the class
40       */
41      public void run ()
42      {
43          try{
44              LfPMessage msg = null ;
45              String next ;
46              String label_15 = "+DIFFUSION+ 15 15";
47              String label_16 = "+DIFFUSION+ 31 31";
48              String label_17 = "+DIFFUSION+ 14 14";
49              String label_18 = "+DIFFUSION+ 13 13";
50              String label_19 = "+DIFFUSION+TR1 12";
51              String label_20 = "+DIFFUSION+ 10 10";
52              String label_21 = "+DIFFUSION+CHOIX_TRAITEMENT 9";
53              String label_22 = "+DIFFUSION+ 8 8";
54              String label_23 = "+DIFFUSION+ 7 7";
55              String label_24 = "+DIFFUSION+DIFFUSION 5";
56              String label_25 = "+DIFFUSION+ 4 4";
57              String label_26 = "+DIFFUSION+ 3 3";
58              String label_27 = "+DIFFUSION+ 2 2";
59              /* sub_diagram for  */
60              next  = label_27 ;
61              while (true) {
```

```
62                  if (next == label_15) {
63                      if ( (INDEX.intValue() <= NBR_CONNECTED.intValue())
                            ) {
64                          next = label_25;
65                      }
66                  }
67                  if (next == label_16) {
68                      next = label_27;
69                  }
70                  if (next == label_17) {
71                      TARGET = (LfPBinderReference) SUBSCRIBED.array[(
                            INDEX.intValue()) - ( 1)].getBinder("DATA_INPUT
                            ");
72                      next = label_15;
73                  }
74                  if (next == label_18) {
75                      next = label_19;
76                  }
77                  if (next == label_19) {
78                      INDEX = new INTEGER((new INTEGER().SUCC(INDEX.
                            intValue()).intValue()));
79                      next = label_24;
80                  }
81                  if (next == label_20) {
82                      for( int prefix37 =  1 ;
83                           prefix37 <= NBR_CONNECTED.intValue() ;
84                           prefix37++) {
85                          INTEGER I = new INTEGER(prefix37) ;
86                          if ( (SUBSCRIBED.array[(I.intValue()) - ( 1)].
                                isEqual(OWNER))) {
87                              INDEX = new INTEGER(I.intValue());
88                          }
89                      }
90                      for( int prefix38 = INDEX.intValue() ;
91                           prefix38 <= (NBR_CONNECTED.intValue() -  1) ;
92                           prefix38++) {
93                          INTEGER I = new INTEGER(prefix38) ;
94                          SUBSCRIBED.array[(I.intValue()) - ( 1)] = (
                                LfPComponantReference) SUBSCRIBED.array[((I
                                .intValue() +  1)) - ( 1)];
95                      }
96                      NBR_CONNECTED = new INTEGER((new INTEGER().PRED(
                            NBR_CONNECTED.intValue()).intValue()));
97                      INDEX = new INTEGER( 1);
98                      next = label_24;
99                  }
100                 if (next == label_21) {
101                     if ( (MSG_KIND.intValue() == new MSG_TYPE( 3).value
                            )) {
102                         next = label_22;
103                     }
104                     if ( (MSG_KIND.intValue() == new MSG_TYPE( 2).value
                            )) {
105                         next = label_20;
106                     }
107                     if ( (MSG_KIND.intValue() == new MSG_TYPE( 1).value
```

```
                        )) {
108                         next = label_23;
109                     }
110                 }
111                 if (next == label_22) {
112                     INDEX = new INTEGER( 1);
113                     next = label_24;
114                 }
115                 if (next == label_23) {
116                     NBR_CONNECTED = new INTEGER((new INTEGER().SUCC(
                            NBR_CONNECTED.intValue()).intValue()));
117                     SUBSCRIBED.array[(NBR_CONNECTED.intValue()) - ( 1)]
                            = (LfPComponantReference) OWNER;
118                     INDEX = new INTEGER( 1);
119                     next = label_24;
120                 }
121                 if (next == label_24) {
122                     if ( (INDEX.intValue() <= NBR_CONNECTED.intValue())
                            ) {
123                         next = label_17;
124                     }
125                     if ( (INDEX.intValue() > NBR_CONNECTED.intValue()))
                            {
126                         next = label_16;
127                     }
128                 }
129                 if (next == label_25) {
130                     MSG.setBinder(TARGET) ;
131                     runtime.sendMessage(MSG) ;
132                     next = label_18;
133                 }
134                 if (next == label_26) {
135                     MSG = runtime.getMessage(INPUT) ;
136                     MSG_KIND = (MSG_TYPE) MSG.getDiscriminant( 1);
137                     OWNER = (LfPComponantReference) MSG.getDiscriminant
                            ( 2);
138                     runtime.commit (msg) ;
139                     next = label_21;
140                 }
141                 if (next == label_27) {
142                     next = label_26;
143                 }
144             }
145         }catch (Exception e) {
146             e.printStackTrace() ;
147             System.exit(1);
148         }
149         try {
150             runtime.removeComponant(this.selfReference) ;
151         }
152         catch (LfPRuntimeError lrExcp) {
153             lrExcp.printStackTrace() ;
154         }
155     }
156 }
```

5.1.4 Code du type `msg_type`

Ce code est généré à partir du type global `msg_type` défini dans la partie déclarative du diagramme d'architecture.

```
1  public class MSG_TYPE implements java.io.Serializable {
2      public int SUBSCRIBE =  1;
3      public int UNSUBSCRIBE =  2;
4      public int DIFFUSE =  3;
5      public int LOCAL_COMMAND =  4;
6      public int value ;
7      int FIRST = 1;
8      int LAST = 4;
9      public MSG_TYPE LAST ()
10     {
11         MSG_TYPE newInstance = new MSG_TYPE() ;
12         newInstance.value = newInstance.LAST ;
13         return newInstance ;
14     }
15     public MSG_TYPE FIRST ()
16     {
17         MSG_TYPE newInstance = new MSG_TYPE () ;
18         newInstance.value = newInstance.FIRST ;
19         return newInstance ;
20     }
21     public MSG_TYPE SUCC( int val)
22     {
23         MSG_TYPE newInstance = new MSG_TYPE() ;
24         if (val >= LAST) {
25             new LfPRuntimeError().printStackTrace();
26             System.exit(1);
27         }
28         newInstance.value = val + 1 ;
29         return newInstance ;
30     }
31     public MSG_TYPE PRED (int val)
32     {
33         MSG_TYPE newInstance = new MSG_TYPE() ;
34         if (val <= FIRST) {
35             new LfPRuntimeError().printStackTrace();
36             System.exit(1);
37         }
38         newInstance.value = val - 1 ;
39         return newInstance;
40     }
41     public MSG_TYPE( INTEGER val)
42     {
43         this.value = val.intValue() ;
44     }
45
46     public MSG_TYPE (int val)
47     {
48         this.value = val ;
49     }
50     public MSG_TYPE()
51     {
```

```
52          super();
53      }
54      public MSG_TYPE(MSG_TYPE val)
55      {
56          this.value = val.value;
57      }
58      public int intValue()
59      {
60          return this.value ;
61      }
62  }
```

5.1.5 Code de démarrage

Ce code est exécuté lors du démarrage de chacuns des sites **L*f*P**. Il s'agit du code d'initialisation de chacune des instances locales. Ce code repose entièrement sur les appels au runtime pour déterminer si une instance est locale (et doit être instanciée) ou si une référence distante doit être créée.

```
1   public class GlobalDeclarations{
2       static LfPBinderReference DATA_OUTPUT ;
3       public static LfPComponantReference S0;
4       public static LfPComponantReference S1;
5       public static LfPComponantReference S2;
6       public static LfPComponantReference S3;
7       public static LfPComponantReference DIFF;
8       public static void initialize ()
9       {
10          DATA_OUTPUT = LfPClientRuntime.newStaticBinder ("DATA_OUTPUT",
                20, "FIFO") ;
11          /* initialization of S0 */
12          String[] prefix17 =   { } ;
13          Object []prefix18 = {} ;
14          S0 = LfPClientRuntime.newStaticComponant("S0", "SENDER",
                prefix17, prefix18, true) ;
15          /* initialization of S1 */
16          String[] prefix21 =   { } ;
17          Object []prefix22 = {} ;
18          S1 = LfPClientRuntime.newStaticComponant("S1", "SENDER",
                prefix21, prefix22, true) ;
19          /* initialization of S2 */
20          String[] prefix25 =   { } ;
21          Object []prefix26 = {} ;
22          S2 = LfPClientRuntime.newStaticComponant("S2", "SENDER",
                prefix25, prefix26, true) ;
23          /* initialization of S3 */
24          String[] prefix29 =   { } ;
25          Object []prefix30 = {} ;
26          S3 = LfPClientRuntime.newStaticComponant("S3", "SENDER",
                prefix29, prefix30, true) ;
27          /* initialization of DIFF */
28          String[] prefix33 =   { "INPUT"} ;
29          Object []prefix34 = {GlobalDeclarations.S0.getBinder("
                DATA_OUTPUT")} ;
```

```
30          DIFF = LfPClientRuntime.newStaticComponant("DIFF", "DIFFUSION",
                prefix33, prefix34, false) ;
31      } /* End of initialization function */
32  }
```

5.2 Code des composants externes

5.2.1 Code du type opaque `GUI.java`

Ce code correspond au type opaque `GUI` définit dans le modèle **L*f*P**. Le rôle de cette classe est de fournir une interface graphique à l'utilisateur. Lorsque ce dernier envoie un message, il est rajouté dans la liste des évennements sortants.

Cette liste peut être accédée via la méthode `GET_MESSAGE` qui retourne le premier élément de la liste. Lorsque la liste est vide, cette méthode est bloquante.

Enfin, la méthode `DISPLAY_MESSAGE` sert à afficher un message reçu.

```
1   import java.io.* ;
2   import javax.swing.* ;
3   import java.awt.* ;
4   import java.awt.event.* ;
5
6
7   public class GUI implements Serializable {
8
9       /*
            ************************************************************************
10       * Boolean that must be set to false if the graphical UI canot
11       * be activated.
12       */
13      boolean graphicalUI = true ;
14
15      /*
            ************************************************************************
16       * the frame that contains the graphical elements
17       */
18      JFrame chatFrame = null ;
19
20      /*
            ************************************************************************
21       * The text field that contains the message entered by the user.
22       */
23      JTextField newMsg = new JTextField(15) ;
24
25      /*
            ************************************************************************
26       * The last message entered by the user.
27       */
28      String msg = new String();
29
30      /*
            ************************************************************************
```

```
31        * The list of events generated by the UI
32        */
33       EventList eventList = new EventList() ;
34
35       JEditorPane dialog = new JEditorPane ("plain/text",
36                                             "Conversation => ") ;
37
38       /*
             ************************************************************************
39        * The nickname provided by the user
40        */
41       private String nickname = "";
42
43       private class MyAction implements ActionListener {
44           public void actionPerformed (ActionEvent e) {
45               msg = newMsg.getText() ;
46               msg = msg.trim() ;
47               T_MESSAGE message = new T_MESSAGE() ;
48               MSG_TYPE msgType = null ;
49               /*
50                * First handle the commands
51                */
52               System.out.println("# setting message content : " + msg) ;
53               msg = msg.trim() ;
54               if ( msg.equals("/join") || msg.equals("/subscribe") ) {
55                   System.out.println("# message type : SUBSCRIBE") ;
56                   msgType = new MSG_TYPE (new MSG_TYPE().SUBSCRIBE) ;
57               }
58               else {
59                   if ( msg.equals ("/quit") || msg.equals ("/unsubscribe"
                         ) ) {
60                       System.out.println("# message type : UNSUBSCRIBE")
                             ;
61                       msgType = new MSG_TYPE (new MSG_TYPE().UNSUBSCRIBE)
                             ;
62                   }
63                   else {
64                       if ( msg.indexOf("/nickname ") == 0 ) {
65                           nickname = msg.substring(10).trim() ;
66                           msgType = new MSG_TYPE (new MSG_TYPE().
                                 LOCAL_COMMAND) ;
67                       }
68                       else {
69                           System.out.println("# message type : DIFFUSE")
                                 ;
70                           msgType = new MSG_TYPE(new MSG_TYPE().DIFFUSE)
                                 ;
71                           msg = nickname + " " + msg ;
72                       }
73                   }
74               }
75
76               message.setType (msgType) ;
77               message.setContent(msg) ;
```

```
78              eventList.addEvent(message) ;
79              newMsg.setText("") ;
80          }
81      }
82
83      /*
            ************************************************************************
84       * create the frame that contains the GUI
85       */
86      private void createChatFrame()
87      {
88          Container contentPane ;
89
90          chatFrame = new JFrame ("Chat window...") ;
91          contentPane = chatFrame.getContentPane() ;
92          Box col1 = new Box(BoxLayout.Y_AXIS) ;
93          contentPane.add(col1) ;
94
95          Dimension dim = new Dimension() ;
96
97          dialog.setAutoscrolls(true) ;
98          dialog.setEditable(true) ;
99          dim.setSize(400, 100) ;
100         dialog.setPreferredSize(dim) ;
101         dim = new Dimension(100, 100) ;
102         dialog.setMinimumSize(dim)  ;
103         dim = new Dimension  (800, 600) ;
104         dialog.setMaximumSize(dim) ;
105
106         dialog.setDoubleBuffered(true) ;
107         dialog.setEditable(false) ;
108
109         JScrollPane scrollDialog = new JScrollPane(dialog) ;
110         col1.add(scrollDialog) ;
111
112         Box horizontal = new Box(BoxLayout.X_AXIS) ;
113
114
115         dim = new Dimension (800, 100) ;
116         newMsg.setMaximumSize(dim) ;
117         horizontal.add(newMsg) ;
118         newMsg.addActionListener(new MyAction()) ;
119
120         JButton send = new JButton("send") ;
121
122         send.addActionListener(new MyAction()) ;
123
124         send.setDefaultCapable(true) ;
125
126         horizontal.add(send) ;
127
128         col1.add(horizontal) ;
129         chatFrame.pack() ;
130         chatFrame.setVisible(true) ;
131
```

```
132     }
133
134
135     public T_MESSAGE GET_MESSAGE () throws Exception
136     {
137         T_MESSAGE message = new T_MESSAGE() ;
138
139
140         if ( chatFrame == null ) {
141             createChatFrame() ;
142         }
143
144         if ( graphicalUI == false ) {
145             System.out.println("# Please enter message : ") ;
146             InputStreamReader streamReader = new InputStreamReader(
                    System.in) ;
147             BufferedReader reader = new BufferedReader(streamReader) ;
148
149             message.setContent(msg) ;
150         } else {
151             message = (T_MESSAGE) eventList.getEvent() ;
152             System.out.println("%%% GOT A MESSAGE...") ;
153         }
154
155         return message ;
156     }
157
158     public void DISPLAY_MESSAGE (T_MESSAGE message )
159     {
160
161         if ( graphicalUI ) {
162             String text = dialog.getText() ;
163             text =  message.getContent()  +  "\n" + text ;
164             dialog.setText (text) ;
165         }
166         else {
167             System.out.println("# Message content : ") ;
168             System.out.println("# > " + message.getContent() + " <") ;
169         }
170     }
171
172
173
174 }
```

5.2.2 Code du type opaque `t_message`

Ce code implémente le type opaque `t_message` définit dans le modèle **L*f*P**. Il permet de représenter les message échangés par l'application.

```
1 public class T_MESSAGE  implements java.io.Serializable {
2     private String msgContent = null ;
3     private MSG_TYPE msgType ;
4
5     public T_MESSAGE (String msg)
6     {
```

```
7           msgContent = msg ;
8       }
9
10      public T_MESSAGE()
11      {
12          msgContent = new String() ;
13      }
14
15      public String getContent()
16      {
17          return new String(this.msgContent) ;
18      }
19
20      public MSG_TYPE GET_TYPE ()
21      {
22          return this.msgType ;
23      }
24
25      public void setContent(String msg)
26      {
27
28
29          this.msgContent = msg ;
30      }
31
32      public void setType (MSG_TYPE type)
33      {
34          this.msgType = type ;
35      }
36
37  }
```

5.2.3 Code de la classe java EventList

Cette classe est nécessaire à l'implémentation des deux précédente bien qu'elle n'apparaisse pas dans la spécification **L*f*P**. Elle implémente la file d'évennement nécessaire à l'implémentation de la méthode `get_message` de la classe `GUI`.

```
1   import java.util.* ;
2
3   public class EventList {
4
5       LinkedList events = new LinkedList() ;
6
7
8       public synchronized Object getEvent() throws Exception
9       {
10
11          if ( events.size() == 0 )
12              {
13                  wait () ;
14              }
15          return events.removeLast() ;
16      }
17
```

```
18      public synchronized void addEvent(Object event)
19      {
20          events.addFirst(event) ;
21          System.out.println("% ADDED A MESSAGE ") ;
22
23
24          if ( events.size() == 1 )
25              {
26                  notify() ;
27              }
28      }
29
30  }
```