Type document : *SRS*	*Projet RNTL MORSE*	**Date :** 13/06/2004
Sous-projet n° 2		**numéro de doc :** MORSE-SRS-041126-V0.1-FGI
tâche : 2.1		**rédacteur (s)** Frédéric Gilliers

Etude de cas : un système de «messagerie instantannée»

Table des matières

Table des figures

Table des matières

L'objectif de ce chapitre est de présenter une étude de cas permettant d'illustrer l'utilisation du langage **L*f*P** dans un cas concrêt. L'étude présentée est un système de messagerie instantannée de type IRC. Il s'agit donc d'une application capable de gérer plusieurs groupes de discution en parallèle.

Cette application a été choisie car elle représente un bon compromis entre la taille du modèle qui doit rester compatible avec ce mémoire d'une part, et la couverture des fonctionnalités du langage **L*f*P** et du générateur de code d'autre part. Les études de cas plus volumineuses seront réalisées dans le cadre du projet MORSE qui prévoit l'étude d'exemples industriels beaucoup plus volumineux et complexes.

0.1 Présentation de l'exemple

Le principe du système de messagerie modélisé ici est le suivant :

- un serveur attend les connexions de nouveaux clients ;
- un client peu se connecter sur un serveur soit pour se joindre à un groupe existant, soit pour créer son propre groupe de discution ;
- un client peut quitter le groupe auquel il est connecté à tout instant.

Du côté serveur, l'application de messagerie instantannée gère un ensemble de groupes de discution indépendant les uns des autres. Chaque serveur doit pouvoir gérer plusieurs groupes de discution simultanément. Il gère les arrivées et départ des utilisateurs, ainsi que leurs changements de groupes. Chaque client de l'application doit être capable de lire gérer les interactions avec l'utilisateur pour lui permettre de s'abonner à un groupe ou de le quitter, lire les messages qu'il souhaite envoyer, et afficher les messages qu'il reçoit.

Afin de conserver un modèle d'une complexité raisonnable pour cet exemple, nous feront les hypothèses simplificatrices suivantes :

- chaque client n'est connecté qu'à un seul groupe de discution à un instant donné ;
- les liaisons de communication sont fiables ;
- les clients sont *fiables* et respectent toujours le protocole d'interaction prévu par le serveur.

La figure 1 présente le diagramme de classe UML simplifié du système. Un utilisateur ou *client* du système de messagerie est constitué d'une instance des trois classes suivantes :

- `receiver` qui réceptionne les messages en provenance du service de diffusion ;
- `sender` qui envoie les messages à destination du service de diffusion ;
- `GUI` l'interface graphique du client qui gère les interactions avec l'utilisateur.

La partie serveur du système est assurée par deux classes : la classe `groupe` implémente le comportement d'un groupe de discution. la classe `server` gère l'ensemble des groupes actifs sur le serveur.

La classe `server` assure les tâches suivantes :

- gestion des demandes de connexion et deconnexion des clients ;

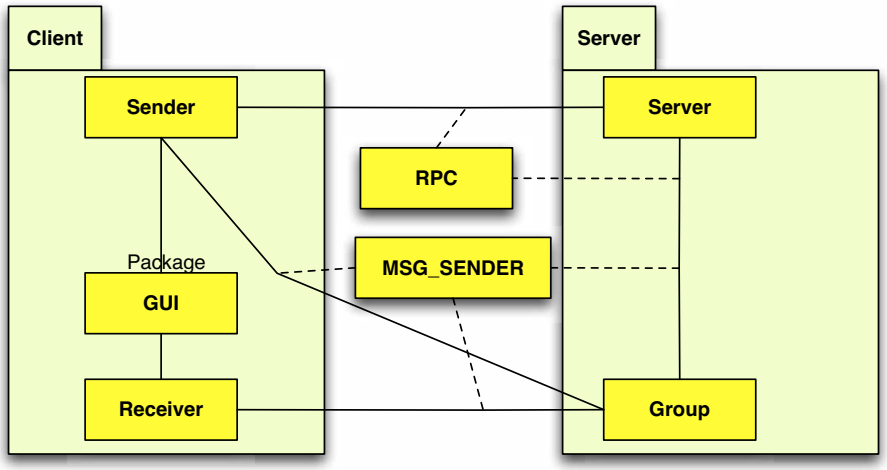


FIG. 1 – Diagramme de classe simplifié de l'étude de cas

– gestion de la liste des groupes actifs ;

La classe `group` assure les tâches suivantes :

– enregistrement des nouveaux clients ;
– diffusion à tous les participant de chacun des messages soumis par un client ;
– gestion des départs des clients ;
– terminaison lorsque le groupe est vide.

Les communications entre les clients et le serveur sont assurées par la classe d'interaction `RPC` qui permet d'assurer une interaction de types *appel de procédure distant*. Tous les appels de méthodes passant par ce média doivent fournir une valeur de retour.

Les communications entre le client et le groupe auquel il appartient sont assurées par la classe d'interaction `MSG_SENDER` qui permet une communication de type *envoi de message* entre deux classes.

Les interactions entre les groupes et le serveur sont assurés par deux cannaux distincts : l'un est de type `RPC` pour les appels de procédures synchrones, l'autre est de type `MSG_SENDER` pour les appels de méthode sans valeur de retour (équivalent à un envoi de message).

0.2 Partie statique du modèle : le diagramme d'architecture

La figure 2 présente le diagramme d'architecture du système considéré. Il définit la vue statique du modèle.

0.2.1 Définition des composants

La partie graphique du diagramme d'architecture définit les composants du système. On y retrouve les composants vus sur le diagramme de classe. Les classes `sender`, `receiver`, `server`, et `group` sont implémentées sous forme de classes **L*f*P**. En revanche, les classes d'interaction `RPC`, et `MSG_SENDER` ont été traduites sous forme de médias. Ces composants définissent la partie controle de l'application, leur comportement sera donc complètement modélisé dans la spécification **L*f*P**.

Le diagramme d'architecture précise également le fonctionnement des liaisons entre les composants en détaillant les files de messages (représentées par les binders), ce qui permet de définir la topologie de l'application. Par exemple, la classe `server` utilise deux files de messages distinctes. Ainsi, pour ses interactions avec les clients, le serveur utilise un binder relié à un média de type

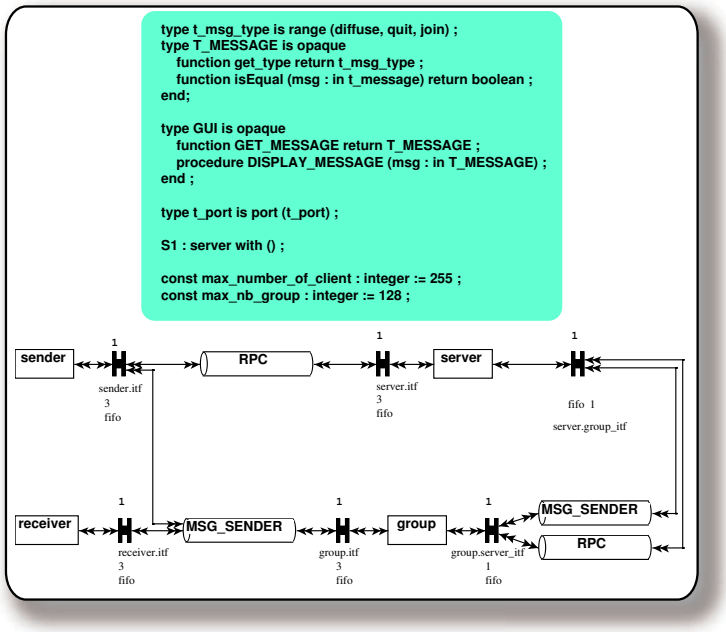


FIG. 2 – Diagramme d'architecture du système de messagerie instantannée

`RPC` correspondant au port `itf` de la classe. Pour ses interactions avec les instances de la classe `group`, le serveur utilise le binder `group_itf`. Ce dernier peut être relié à la classe `group` par deux types de médias : `MSG_SENDER` pour les appels de procédures asynchrones (sans message de retour), et `RPC` pour les appels de méthodes nécessitant une valeur de retour (fonctions). Ces deux médias sont reliés au binder `server_itf` qui définit la file de messages utilisés par le groupe pour traiter les messages du serveur.

Les clients du système de messagerie (classes **L*f*P** `sender` et `receiver`) communiquent avec les groupes par l'intermédiaire d'un média de type `MSG_SENDER`. Chacune de ces classes définit sa file de message gérant ses interactions avec les autres composants du modèle. Tous les binders du modèle sont de multiplicité `1` ce qui signifie qu'une nouvelle instance de la file de messages est créée pour chaque instance de la classe définie dans l'attribut binding.

0.2.2 Définition des types de données

La partie déclarative du diagramme d'architecture de la figure 2 définit les types et constantes partagés par les composants du modèle.

Le type `GUI`

La classe `GUI` est un composant de traitement de données ; son rôle est de traiter les interactions entre l'application et l'utilisateur l'utilisateur, et de les traduire en évènements qui pouront être traités par la partie contrôle de l'application. Pour cette raison, elle est traduite dans la spécification **L*f*P** par un type opaque défini dans le diagramme d'architecture du modèle.

Le type `GUI` définit donc l'interface de la partie controle avec l'interface graphique de l'application. Il fournit deux primitives permettant de modéliser les interactions entre la partie controle de l'application et la partie de traitement de donnée :

– `get_message` retourne le prochain message que le client veut envoyer ;

– `display_message` provoque l'affichage d'un message reçu depuis le groupe de discution courant.

Le type `t_msg_type`

le type énuméré `t_msg_type` définit la nature d'un message de l'application. Les variables de ce type peuvent prendre trois valeurs :

– `diffuse` qui correspond à un message à diffuser à tous les membres du groupe courant, Par convention, le contenu d'un message de type `diffuse` est le texte à diffuser à tous les abonnés du groupe.
– `quit` qui correspond à l'action de quitter le groupe de discution auquel le client participe, le message est alors vide ;
– `join` qui correspond à l'action de créér un groupe de discution ou de le rejoindre s'il existe déjà ; par convention, le contenu d'un message de type `join` est le nom du groupe à rejoindre ou à créer.

Le type `t_message`

le type opaque `t_message` définit l'interface de la partie controle avec un message envoyé par le client au système de messagerie. Le contenu du message n'est pas accessible à la partie controle. En revanche, ce type fournit deux *appels externes* devant respecter le contrat suivant :

– `get_type` retourne le type du message sous forme d'une valeur de type `t_msg_type` ;
– `isEqual` retourne un booléen égal à `true` si le message passé en paramètre a le même contenu que le message sur lequel l'appel est réalisé.

Le type `t_port`

Le type `t_port` correspond à un type de port pour lequel les discrimiants des messages doivent contenir exactement une valeur de type `t_port`. Il est utilisé pour définir les types des ports de toutes les classes du modèle.

Les variables globales du modèle

Les variables globales du système définissent les constantes partagées par tous les composants de l'application, ainsi que les instances statiques des composants.

La constante `max_number_of_client` définit le nombre maximum de personnes pouvant se connecter à un des groupes géré par le serveur.

La constante `max_number_of_group` définit le nombre maximum de groupe de discution que peut héberger un serveur.

La variables `S1` définit l'instance de la classe `server` qui sera créée au démarrage du système de messagerie pour recevoir les messages des clients.

0.3 Partie dynamique du modèle : comportement des composants

Cette section présente les diagrammes de comportement des composants du système de messagerie instantannée. Ils définissent le comportement dynamique de l'application.

0.3.1 Modélisation des médias

Le modèle comporte deux médias : `RPC` et `msg_sender`.

Le média `msg_sender`

Ce média implémente l'envoi d'un message depuis un composant source vers le destinataire. Son diagramme de comportement est donné sur la figure 3.

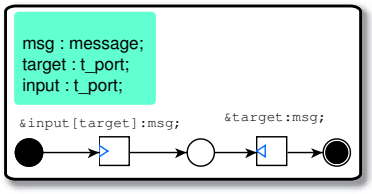


FIG. 3 – Diagramme de comportement du média `msg_sender`

Ce média déclare trois variables :

- `msg` de type message contiendra le message reçu par le média ;
- `input` de type `t_port` est le port reliant le média au composant émeteur du message ;
- `target` également de type `t_port` est le port reliant le média au destinataire du message.

Lors de la création d'une instance de `msg_sender`, l'utilisateur doit initialiser l'attribut `input`. Lorsque son exécution débute, le média se met en attente d'un message sur ce port. Comme le spécifie la définition du type `t_port`, le message doit contenir la référence d'un binder dans son discriminant. Cette valeur est affectée à l'attribut `target`. Le contenu du message est sauvegardé dans l'attribut `msg`. L'attribut `target` est ensuite utilisé comme port de sortie par l'instuction d'envoi de message.

Une fois que le message a été déposé dans le binder cible, le média termine son exécution. Une instance de `msg_sender` ne peut donc servir à envoyer qu'un seul message.

Le média `RPC`

Le diagramme de comportement de ce média est donné par la figure 4.

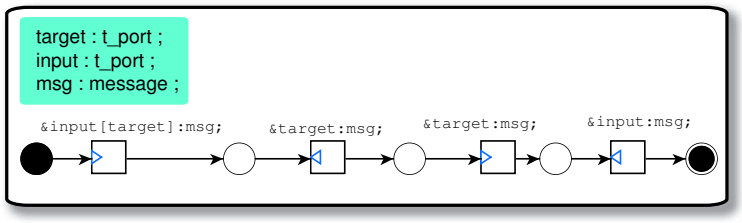


FIG. 4 – Diagramme de comportement du média `RPC`

Le média `RPC` déclare trois attributs de même nom et de même type que `msg_sender`. Le début de l'exécution de `RPC` est similaire : le média se met en attente sur le port `input` initialisé lors de l'instanciation, puis le dépose dans le binder spécifié dans le discriminant du message reçu.

Ensuite, `RPC` se met en attente sur le binder où il a déposé le premier message (troisième transition du média). Lorsqu'il reçoit un message , il le dépose dans le binder désigné par `input` (quatrième transition). Cette série d'opérations permet de transmettre le message d'activation d'une méthode et le message de retour correspondant.

Après avoir traité un appel de méthode, le média termine son exécution. Une nouvelle instance de `RPC` doit donc être créée avant chaque opération d'appel de méthode utilisant ce média.

0.3.2 La réception des messages du côté client

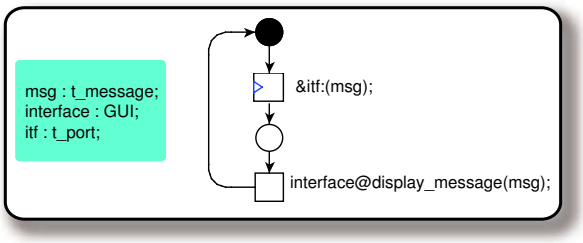


FIG. 5 – Diagramme de comportement de la classe `Receiver`

Ce rôle est assuré par la classe `receiver`. Son comportement est très simple et est présenté sur la figure 5. Cete classe attend un message sur son port `itf`. Ce message doit contenir une seule valeur de types `t_message`. Ce message est ensuite simplement passé à l'interface graphique pour être affiché pour lecture par l'utilisateur. Cette opération est réalisée par l'appel à la méthode externe `display_message` définie sur le type opaque `GUI`. L'instance de `receiver` retourne ensuite dans son état initial pour attendre le prochain message.

0.3.3 L'envoi des messages par le client

L'envoi des messages par le client est assuré par la classe sender dont le diagramme de comportement est présenté sur la figure 6. La première action réalisée par la classe `sender` est de créer l'instance de `receiver` qui lui est associée pour le client. Le composant externe `GUI` est initialisé lors de sa déclaration. Ensuite, le composant entre dans une boucle correspondant à la lecture d'un message entré par l'utilisateur sur l'interface graphique et à son traitement en fonction de son type.

La lecture du message, ainsi que l'identification de son type sont effectués par la transition de sortie de l'état `start_loop` (sur fond vert sur la figure). Le traitement correspond aux trois branches sur fond bleu, jaune et rouge.

Traitement des messages de type `diffuse`

La branche centrale sur fond jaune correspond au traitement des messages de type `diffuse`. Dans ce cas, le message est envoyé directement au groupe pour qu'il le diffuse à tous ses abonnés. Cette branche réalise les opérations suivantes :

- la première transition sert à définir l'alternative menant dans cette branche : `msg_kind` doit contenir la valeur `diffuse` ;
- ensuite le média servant à l'envoi de message est instancié, avec comme binder d'entrée le binder `itf` de l'instance courante ;
- la transition d'envoi de message crée un message d'activation pour la méthode asynchrone `diffuse` avec comme paramètre le message à diffuser, et un discriminant de message contenant la référence du binder où le message doit être déposé.

Le média utilisé est de type `msg_sender` car la méthode `diffuse` est asynchrone et ne donnera pas lieu à un message de retour.

Traitement des messages de type `join`

La branche la plus à gauche, sur fond bleu, correspond au traitement d'un message de type `join`, ce qui signifie que l'utilisateur cherche à rejoindre un groupe dont le nom est spécifié dans le corps du message.

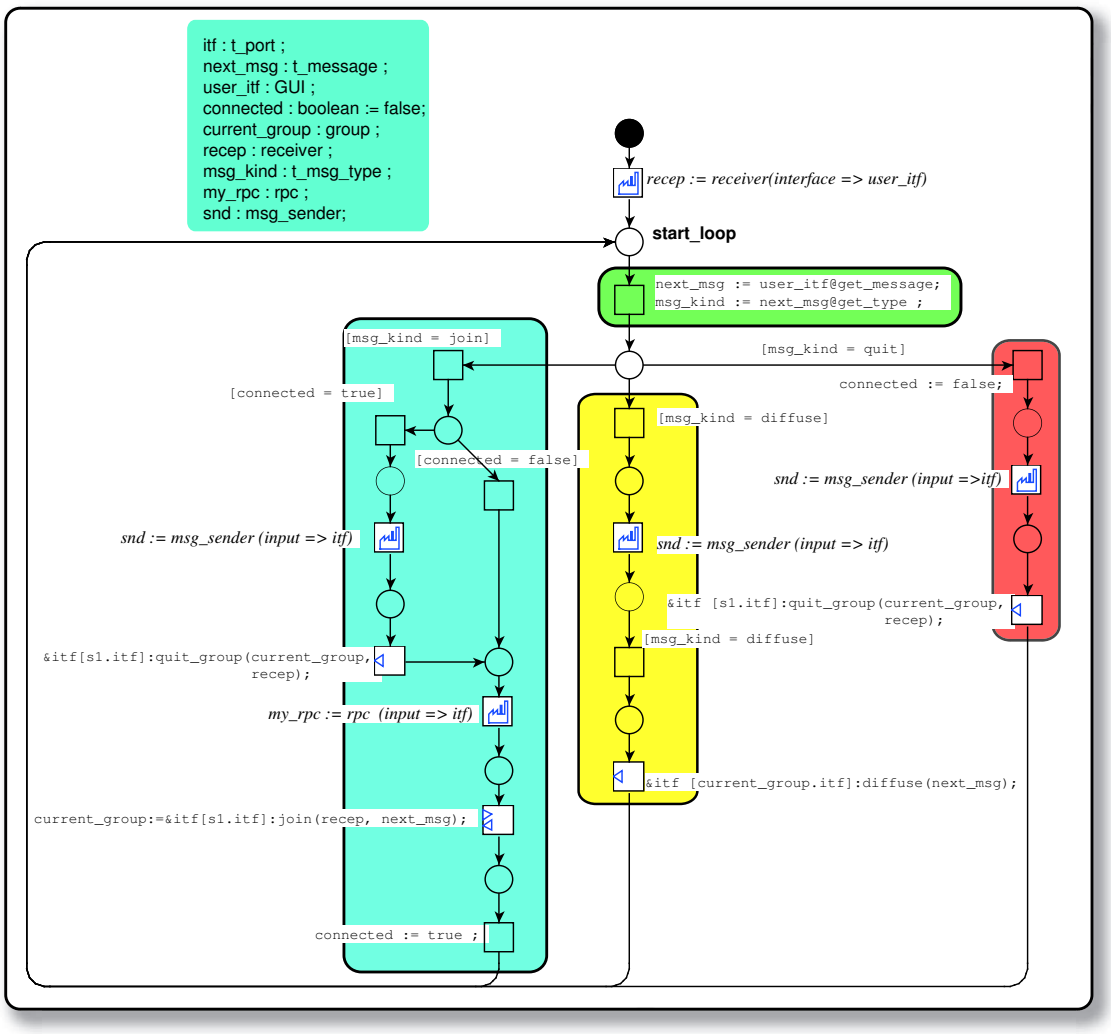


FIG. 6 – Diagramme de comportement de la classe `Sender`

Cette branche réalise les opérations suivantes :

- si le client est déjà connecté à un groupe (la garde `[connected = true]` s'évalue à `true`), il commence par le quitter en appelant la méthode `quit_group` sur le serveur.
- dans tous les cas, le client appel la méthode `join` de la classe `server` pour rejoindre le groupe dont le nom est contenu dans message.

La valeur de retour de la fonction `join` de la classe `server` contient la référence du groupe auquel l'utilisateur appartient après la connexion.

Avant chaque envoi de message, un média approprié est instancié : `MSG_SENDER` pour les appels de méthode asynchrone, et `RPC` pour les appels de méthode synchrones.

Traitement des messages de type `quit`

La branche sur fond rouge correspond au traitement d'un message `quit`, ce qui signifie que le client souhaite quitter le groupe auquel il est actuellement connecté.

Cette branche effectue les opérations suivantes :

- instanciation d'un média de type `MSG_SENDER` ;

– appel de la méthode asynchrone `quit_group` de la classe `server` avec en paramètre la référence du groupe à quitter et la référence du client qui se désabone.

Quelle que soit la branche suivie, le client saute ensuite à l'*état* `start_loop` qui définit le début de la boucle d'interaction avec l'interface graphique.

0.3.4 La gestion du groupe : la classe `group`

Cette classe permet aux abonnés d'un groupe de communiquer par diffusion : chaque message émit par un client est transmis par tous les abonnés du groupe. Les instances de cette classe sont toujours créées par le serveur qui héberge le groupe, à la demande d'un utilisateur.

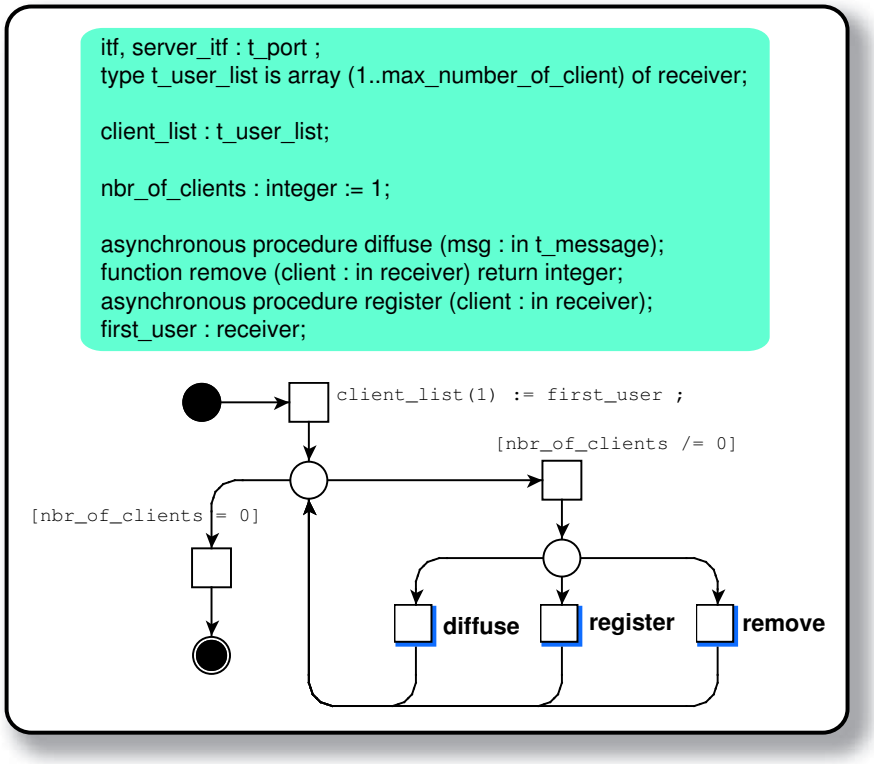


FIG. 7 – Diagramme de comportement de la classe `group`

Cette classe utilise un type tableau `t_user_list` pour conserver la liste des clients connectés. Cette liste est définie par la variable `client_list`. Enfin, `nbr_of_clients` définit le nombre de clients connectés à un instant donné sur le groupe. Cette valeur est initialisée à 1, car un groupe doit au moins avoir un client pour exister. La référence du premier client du groupe est conserveé dans la variable `first_user`. Cette variable est initialisée par la classe `server` avec la référence de l'utilisateur qui provoque l'instanciation du nouveau groupe.

Lors de son instanciation, le groupe commence par insérer le premier utilisateur dans la liste de ses abonnés. Le groupe arrive alors dans son état central : si le nombre d'abonnés est supérieur ou égal à 1, trois méthodes sont activables :

– `diffuse` permet de diffuser un message à l'ensemble du groupe ;
– `register` permet de rajouter un nouveau client dans la liste des abonnés ;
– `remove` retire du groupe le client passé en paramètre et retourne le nombre d'utilisateurs restant.

Si le nombre d'abonné est égal à zéro (0), le groupe n'a plus de raison d'être et l'instance termine son exécution. Elle sera reitirée de la liste des groupes valides par le serveur.

La méthode `diffuse`

La méthode diffuse permet à un abonné de transmettre un message à tous les abonnés du groupe. Cette méthode dont le diagramme de comportement est présenté sur la figure 8 parcours la liste des abonnés au groupe et leur transmet le message passé en paramètre.

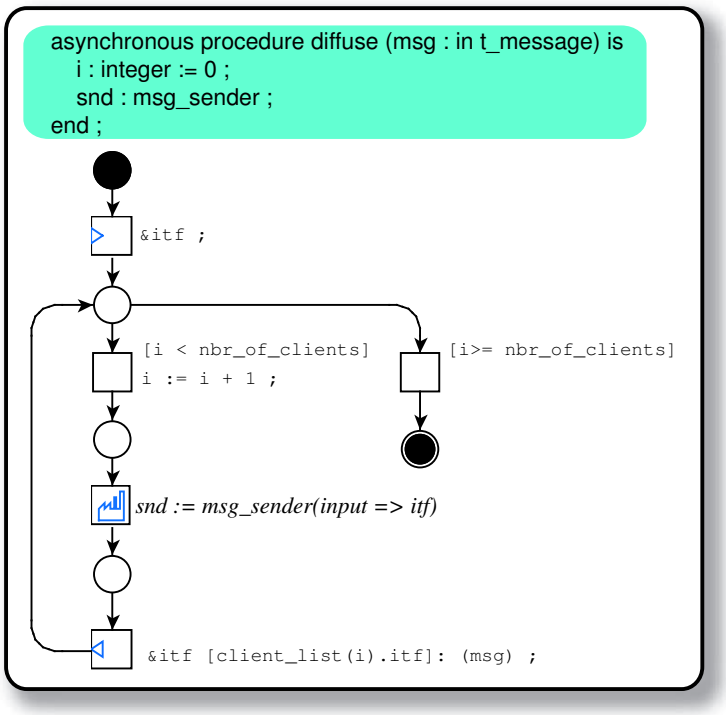


FIG. 8 – Diagramme de comportement de la méthode `diffuse`

Le parcours du tableau est effectué par la boucle gauche du diagramme. Le corps de la boucle incrémente le compteur d'indice, crée une instance du média `msg_sender` pour communiquer avec chaque abonné, et lui envoie le corps du message.

La méthode `register`

La méthode `register` dont le diagramme de comportement est présenté sur la figure 9 ajoute un abonné à un groupe existant. Le paramètre de cette procédure est l'utilisateur à ajouter au groupe. Il est ajouté dans le tableau des abonnés du groupe, et le nombre d'abonné est incrémenté.

La méthode `remove`

La méthode `remove` permet au serveur de retirer un utilisateur du groupe. Le paramètre de la procédure est la référence de l'utilisateur à supprimer ; son diagramme de comportement est présenté sur la figure 10.

La procédure `remove` commence par retirer l'utilisateur du tableau contenant la liste des abonnés et elle décrémente le compteur du nombre d'abonnés. Cette valeur est ensuite envoyée comme valeur de retour de la fonction.

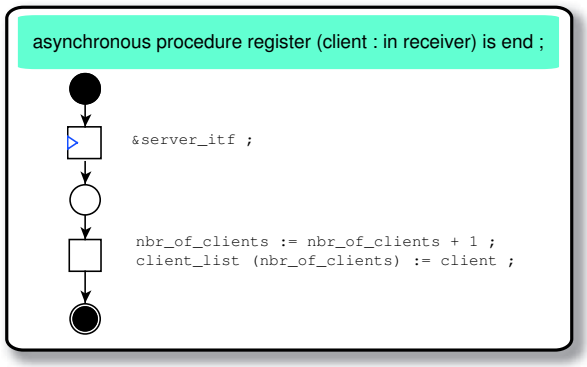


FIG. 9 – Diagramme de comportement de la méthode `register`

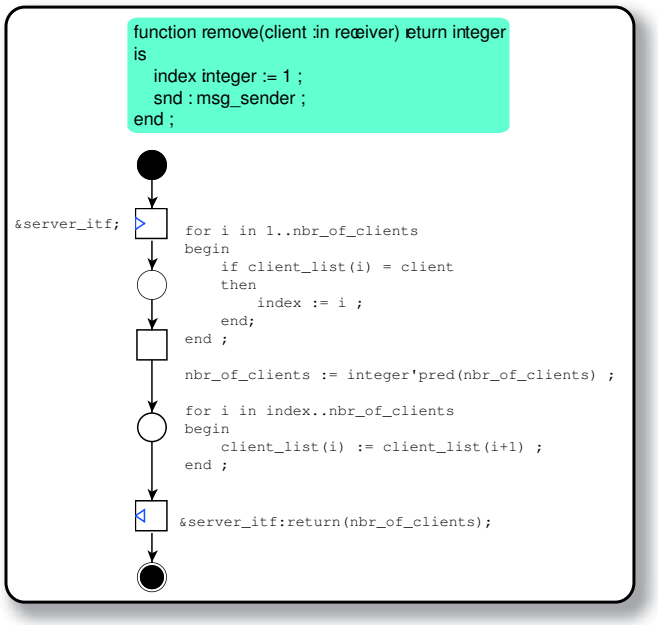


FIG. 10 – Diagramme de comportement de la méthode `remove`

Pour retirer l'utilisateur, le groupe commence par déterminer son indice dans le tableau contenant la liste des utilisateurs, puis à partir de cet indice, il décale toutes les valeurs d'une case vers la gauche.

0.3.5 Le serveur de gestion de groupes

L'objectif de la classe `server` est de générer les différents groupes de discution dont il a la charge. Le serveur offre une interface basée sur deux méthodes :

- la fonction `join` qui permet de se joindre à un groupe existant ou d'en créeer un nouveau, elle retourne la référence du groupe auquel l'utilisateur vient de s'abonner ;
- la procédure asynchrone `quit_group` qui permet à un utilisateur de se désabonner d'un groupe.

Le diagramme principal de cette classe est présenté sur la figure 11. Les isntances de la classe `server` sont en permanence en attente d'activation d'une des deux méthodes vues précédemment.

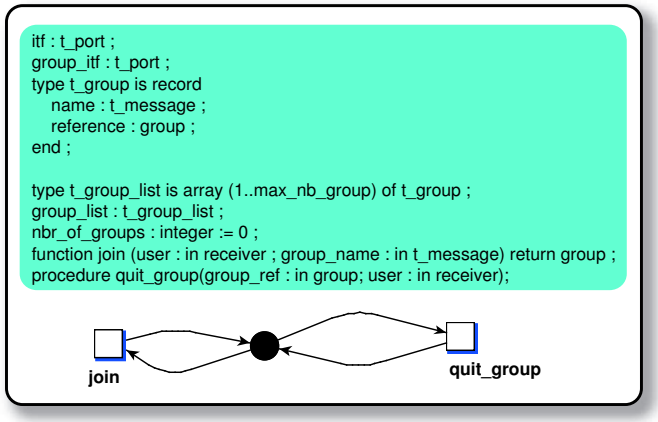


FIG. 11 – Diagramme de comportement de la classe `server`

La méthode `join`

La fonction `join` doit être appelée par un client désirant se joindre à un groupe où le créer ; son diagramme de comportement est donné par la figure 12. Sa valeur de retour est la référence du groupe auquel l'utilisateur vient de s'abonner.

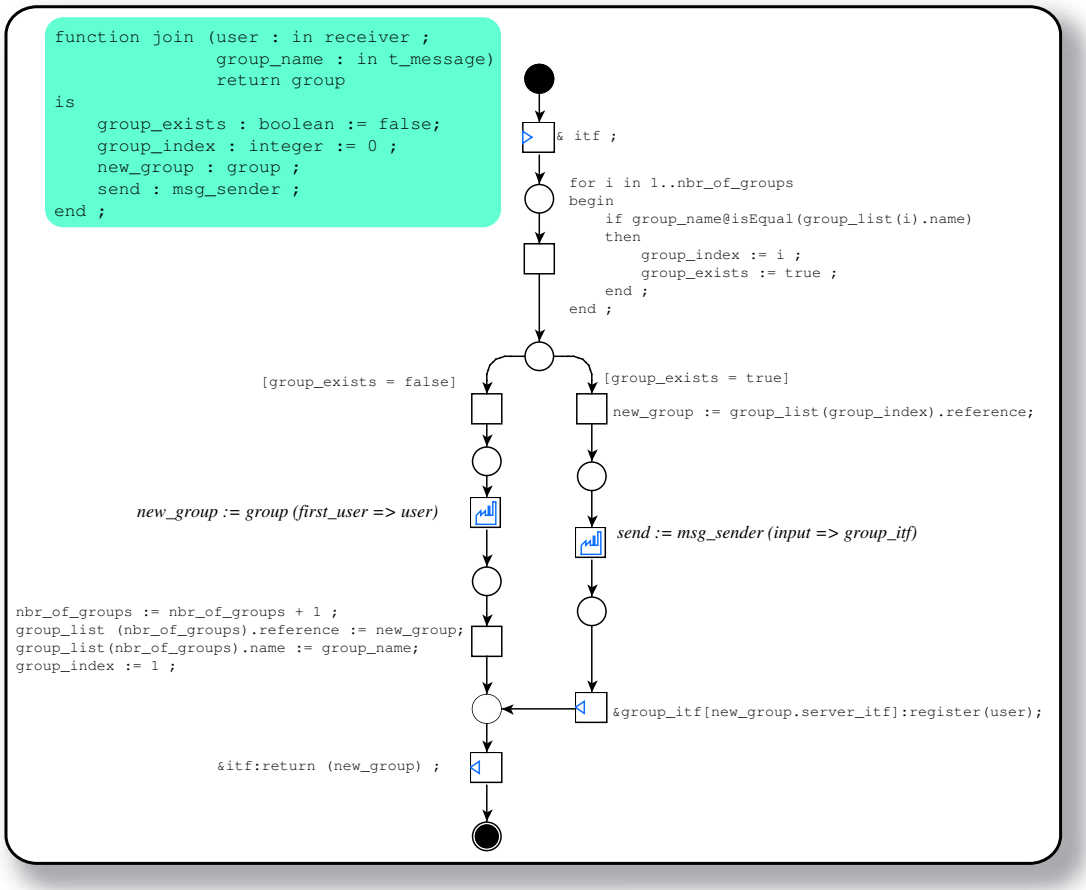


FIG. 12 – Diagramme de comportement de la méthode `join`

Cette méthode a deux paramètres formels :

- `user` est la référence vers le client qui appelle la fonction ;
- `group_name` est le message qui contient le nom du groupe à créer ou à joindre.

Cette méthode commence par déterminer si le groupe auquel l'utilisateur veut s'abonner existe déjà. Cette opération est réalisée par une boucle for qui parcours le tableau contenant la liste des groupes existants. Si le nom d'un des groupes existants est égal au nom du groupe demandé, son indice dans ce tableau est retenu, et le booléen `group_exist` reçoit la valeur true.

Si le groupe existe déjà (branche de droite), une isntance de `msg_sender` est créée pour communiquer avec ce groupe. Puis la procédure asynchrone `register` est appelée pour abonner l'utilisateur au groupe.

Si le groupe n'existe pas encore (branche de gauche), un nouveau groupe est instancié. Lors de l'instanciation, on initialise l'attribut `first_user` avec la référence de l'utilisateur qui provoque la création du groupe. Puis le nouveau groupe est ajouté à la liste des groupes gérés par le serveur.

Enfin, la méthode retourne la référence du groupe auquel l'utilisateur a été abonné. Le message de retour est envoyé sur le port `itf` qui est utilisé pour toutes les communications avec les clients.

La méthode `quit_group`

Cette fonction est appelée par le client lorsqu'il souhaite quitter le groupe de discution auquel il participe. Son diagramme de comportement est donné par la figure 13.

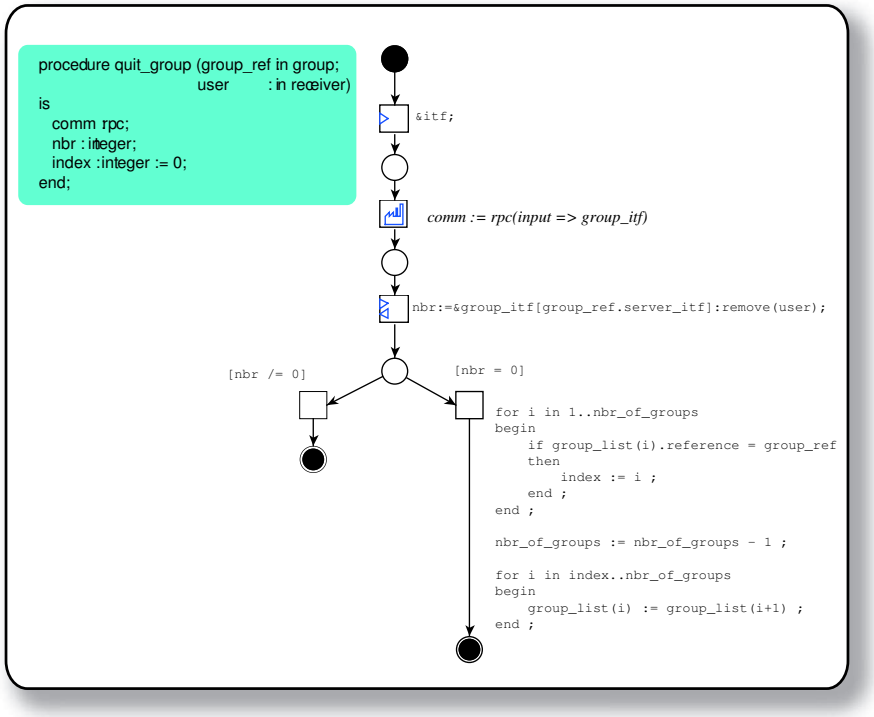


FIG. 13 – Diagramme de comportement de la méthode `quit_group`

La méthode `quit_group` prend deux paramètres

- `group_ref` : la référence du groupe que l'utilisateur veut quitter ;
- `user` : la référence de l'utilisateur qui se désabonne.

`quit_group` commence par appeller la méthode `remove` sur le groupe passé en paramètre. Cette méthode retire du groupe l'utilisateur qui lui est fourni en paramètre et retourne le nombre

d'abonnés restant. Si ce nombre est supérieur à zéro, l'exécution de `quit_groupe` est terminée (branche de gauche). Si ce nombre est égal à zéro, il faut supprimer le groupe de la liste des groupes du serveur (branche de droite).

Pour supprimer le serveur de sa liste, le serveur commence par déterminer son indice dans cette liste (première boucle for), puis à partir de cet indice, il décale toutes les valeurs du tableau d'une case vers la gauche. Une fois ces opérations effectuées, la méthode termine son exécution.

0.4 Déploiement de l'exemple

L'exemple a été déployé à l'aide du fichier de configuration de la figure 14.

```
<instances>
  <instance name="S1" hostName="glaucos.lip6.fr" port="12345" instanceNumber=" 1" />
</instances>
```

FIG. 14 – Fichier de déploiement de l'étude de cas

L'unique instance statique spécifiée est le serveur de messagerie. C'est ce serveur qui recevra les messages de type `join` des clients et qui créera dynamiquement les instances de la classe `group` requises.

Les instances de clients seront créées à la demande en utilisant les fonctionnalités de déploiement dynamique du runtime.

0.5 Présentation du code généré

Le prototype de générateur de code a permis de produire le code correspondant à la spécification **L*f*P** que nous venons de voir ; il représente 1135 lignes de java. Les composants externes de l'application ont été produits manuellement ; ils représentent 270 lignes de code pour obtenir une interface de *chat* minimaliste.

Le code généré pour le modèle de diffusion implémente le modèle et a rendu la spécification **L*f*P** exécutable dans un environnement distribué. Cette section présente quelques extraits significatifs de ce code afin d'illustrer par un exemple concrêt les règles présentées au chapitre **??**.

0.5.1 Implémentation des appels de fonction

La figure 15 présente un extrait du code généré pour la classe serveur (15(b)), ainsi que la transition correspondante (15(a)). Elle est extraite de la méthode `quit_group` de la classe serveur dont le diagramme de comportement a été présenté sur la figure 13. Le code de la figure 15(b) implémente l'appel de la méthode `remove` de la classe `group`. Cette méthode est appelée par le serveur lorsqu'un client souhaite se retirer du groupe.

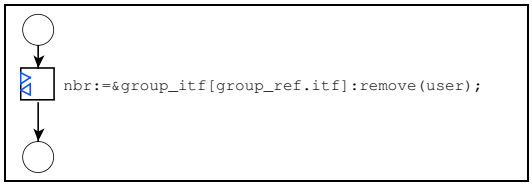
Les caractéristiques de cet appel sont les suivantes :

- Le message d'activation doit êtredéposé dans le port `group_itf` de la classe `serveur` ;
- la méthode appelée est une fonction (cf. figure 10) dont le prototype est le suivant :
- la valeur de retour est stockée dans la variable `nbr`.

La figure 15(b) montre le code correspondant à cette transition. La variable `next` est un marqueur qui définit la prochaine transition à exécuter. Le test de la première ligne permet donc de s'assurer que la classe `server` est prête à exécuter l'appel de méthode.

Les lignes 3 à 6 construisent le message d'activation de la méthode appelée :

- la ligne 3 crée l'instance qui contiendra le message d'activation
- la ligne 4 insère dans le message le nom de la méthode à activer ;



(a) Un exemple de transition d'appel de fonction

```
if (next == label_74) {
    /* METHOD_CALL BEGINS */
    msg = new LfPMessage();
    msg.setMethodName("REMOVE");
    msg.setBinder(GROUP_ITF);
    msg.setDiscriminant(GROUP_REF.getBinder("ITF"), 1);
    msg.setParameter(USER, 1);
    runtime.sendMessage(msg);
    msg = runtime.getSimpleMessage (GROUP_ITF);
    NBR = (INTEGER) msg.getReturnValue();
    /* METHOD_CALL ENDS */
    next = label_70;
}
```

(b) Code correspondant à la transition

FIG. 15 – Transition d'appel de fonction **L*f*P** et le code d'implémentation

– la ligne 5 insère le binder cible dans le message (binder dans lequel il sera déposé) ;
– la ligne 6 insère le premier (ici le seul) élément du discriminant. Ce dernier est déterminé par dé-référenciation de la variable `group_ref`. Cette opération est implémentée par l'appel à la méthode `get_binder` de la class `LfPReference`. Le paramètre de cette méthode est le nom du port dont la référence est recherchée (ici, `ITF`).
– enfin la ligne 7 insère le paramètre effectif de la fonction, c'est à dire le paramètre `user`.

La ligne 8 envoie le message à l'aide de l'appel à la méthode `sendMessage` du runtime associé à cette instance de serveur. la ligne 9 est un appel bloquant au runtime qui retourne le message de retour de la fonction. Une fois ce message reçu, la valeur de retour effective est extraite et affectée à la variable prévue à cet effet dans la spécification **L*f*P**.

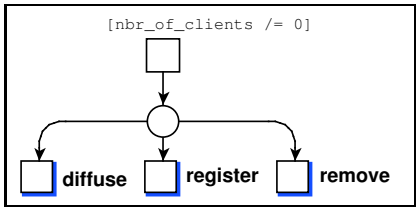
Enfin, la ligne 12 termine l'exécution de la transition en affectant le label de l'état de sortie à la variable next.

0.5.2 Code généré pour une instruction select

La figure 16 présente le code produit pour l'état central de la classe `group` dont le diagramme de comportement était présenté sur la figure 7. La configuration de l'état représenté est rappelée sur la figure 16(a). Lorsque le groupe atteint cet état, il attend un message d'activation pour l'une des trois méthodes suivantes :

– `diffuse` ;
– `remove` ;
– `join` ;

Le code correspondant à cette configuration est présenté sur la figure 16(b). Le code est inséré dans une alternative dont la condition est évaluée à true lorsque l'instruction peut être exécutée. Les lignes 2 et 3 du code généré instancient un tableau qui contient la liste des binders sur lesquels les méthodes sont en tattente : les méthodes `register` et `remove` sont activées par le port `server_itf`, alors que la méthode diffuse est activée par le port `itf`. Ce tableau sera passé en



(a) Etat d'attente d'activation de la classe `group`

```
if (next == label_4) {
    LfPBinderReference[] prefix22 =
        new LfPBinderReference[] {SERVER_ITF, ITF, SERVER_ITF};
    String savedNext23 = next;
    do {
        msg = runtime.getMessage(prefix22);
        if ((msg.getMethodName().equals("REGISTER") ) &&
            (SERVER_ITF.isEqual(msg.getBinder()))) {
            next = label_1;
            runtime.commit (msg);
        }
        if ((msg.getMethodName().equals("DIFFUSE") ) &&
            (ITF.isEqual(msg.getBinder()))) {
            next = label_6;
            runtime.commit (msg);
        }
        if ((msg.getMethodName().equals("REMOVE") ) &&
            (SERVER_ITF.isEqual(msg.getBinder()))) {
            next = label_5;
            runtime.commit (msg);
        }
    } while ( savedNext23 == next );
}
```

(b) Code généré pour une instruction select

FIG. 16 – Instruction select et code généré correspondant

paramètre lors de l'appel au runtime chargé d'attendre le message d'activation.

La ligne 4 sauvegarde le label de l'instruction courante dans une valeur intermédiaire qui sera utilisée pour détecter la réception d'un message d'activation valide.

Le corps de l'instruction *select* est réalisé à l'intérieur de la boucle do { /*...*/} while (); . Cette boucle commence par attendre un message sur la liste de binder qui vient d'être construite grace à la méthode `getMessage` du runtime. Lors de son premier appel, celle-ci va initier les transaction avec les binders qui lui sont passés en paramètre, et retourner le premier message reçu.

Ensuite, le contenu du message est testé : On teste si le nom de la méthode activée par le message est celui d'une méthode activable pour cette instruction[1], et que le binder sur lequel le message a été reçu est bien celui sur lequel la méthode activée était en attente.

Si le message n'est pas un message d'activation compatible avec une des méthodes activable, la valeur de `next` n'est pas changée, et une nouvelle itération de la boucle commence. Cette fois-ci, la méthode `getMessage` retourne le prochain message fourni par les binders.

Si le message est un message d'activation compatible avec une des méthodes en activables :

- le marqueur `next` reçoit le label de la transition qui exécute la méthode correspondante.
- les transactions en cours avec les ports de la classe sont terminées par l'appel au runtime `commit` : ce dernier
  - signifie au binder qui a fourni le message que ce dernier est accepté,
  - annules toutes les autres requêtes de lecture,
  - ré-initialise les structures du runtime liées aux transactions ;
- le flot d'exécution sort de la boucle do { /*...*/} while (); car la valeur de `next` a été modifiée.

Le label associé à une méthode correspond à une transition de l'automate qui va effectuer les opérations suivantes :

- lecture des paramètres du message ;
- appel de la méthode correspondante ;
- envoi de la valeur de retour si applicable.

Une fois ces opérations effectuées, l'appel est cloturé du point de vue de la classe appellée. L'automate saute alors à l'état de sortie de la transition modélisant la méthode appellée.

[1] Si le message n'est pas un message d'activation, l'appel à `getMethodName()` retourne une chaine vide qui peut être comparée au nom des méthodes activables

0.6 Exécution du code généré

REMARQUE: Comme c'est asynchrone et qu'on ne sait pas vraiment quelle séquence d'évènements est exécutée, je ne sais pas si c'est si interessant que ça. Peut être pour dire qu'on peut faire transiter x msg en x secondes

Compte tenu de la nature fondamentalement interactive de ce type d'application, il est nécessaire de mettre en place un environnement de test approprié. Afin de tester le code généré, une deuxième version des composants externe a été développée. Elle permet de *siumler* le comportement des utilisateurs, et de soumettre le serveur à une charge élevée et bien définie.

Afin de tester le code généré, une version spécifique du composant `GUI` a été implémentée. Lors de son instanciation, ce composant demande à l'utilisateur de choisir un fichier de configuration. Ce fichier contient un ensemble de commandes exprimées en XML permettant de définir le comportement d'un utilisateur fictif. Cela permet de "simuler" des conversations entre plusieurs utilisateurs et de tester les réponses des composants générés. De plus, cette méthode permet de provoquer facilement une forte charge en terme de nombre de message et d'observer les réactions du serveur.

REMARQUE: Je ne sais pas trop quoi dire => ajouter quelques captures d'écran pour faire joli. Si possibles multi-plateformes

0.7 Conclusion

Cette étude de cas a permis de tester le générateur de code sur un exemple de modèle raisonnable. Il a permis de valider plusieurs points de la méthodologie vue au chapitre **??**.

Le langage **L*f*P** s'est montré apte à la description de cet exemple, et a permis de le modéliser en peu de temps. De plus, le découpage entre la partie traitement des données et la partie contrôle s'est faite sans problème. L'interface entre ces deux aspects a pu être spécifiée sans problème bien qu'elle comprenne la gestion d'évennements déclenchés par les utilisateurs (envois de messages). Cela permet de valider l'approche par *types opaques* préconisée par **L*f*P**.

A partir de cette description, le générateur de code a produit rapidement un programme exécutable. La mise au point de cette application de messagerie instantannée a été très peu coûteux en terme de développement. Seuls les composants externes ont du être développés à la main. Sur cet exemple, ils ne représente que 270 lignes de java. La partie *contrôle* de l'application représente quant à elle 1135 lignes de java produites par le générateur de code.

Enfin, il faut remarquer que cette messagerie instantannée a pu être entièremnet produite en déchargeant le programmeur des aspects liés à la distribution. La totalité du code produit de manière "traditionnelle" par un codage manuel est centralisé. Les aspects liés au réseau sont entièrement encapsulé par la description en **L*f*P** de la partie contrôle de l'application. Néanmoins, certains points peuvent encore être améliorés ; ils seront développés dans le chapitre suivant.