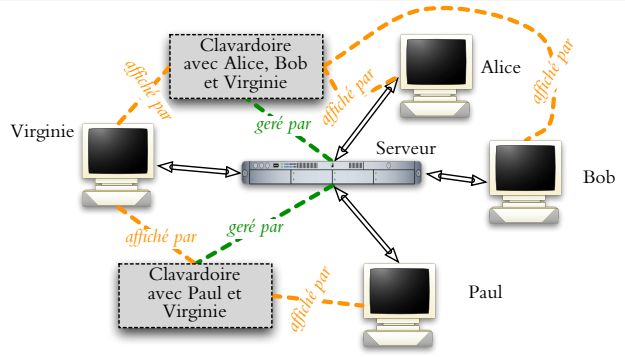
IFT1025 Automne 2009 — Devoir 3

Miklós Csűrös

5 novembre 2009

À remettre avant 23 :59 mercredi le 18 novembre par courrier éléctronique. Ce travail est destiné à des équipes de deux ou trois étudiants. Le but du TP est de se familiariser avec la conception d'un interface graphique, ainsi qu'avec l'interaction de procéssus légers dans une application client-serveur.

1 Application clavardage



Dans ce TP, vous avez à implanter une application de clavardage (*chat*) pour établir des discussions entre plusieurs usagers. Une «salle» de discussion s'appelle un clavardoir (*chat room*). On peut avoir plusieurs participants dans le même clavardoir : tout le monde envoie des messages à tous les participants. L'implantation simule un serveur et les clients à l'aide des *Threads* : dans le prochain TP, on ajoutera la communication propre sur un réseau.

2 Connexions et messages

Le clavardage s'établit par connexions entre le serveur et les clients individuels. Une connexion comprend deux canaux de communication unidirectionnelle : un canal transmet les messages du serveur au client et l'autre transmet les messages du client au serveur.

Un client a un identificateur (id) et un nom ou pseudonyme (nom). Le serveur stocke l'identificateur de chaque client, mais ne le transmet pas parmi eux. Le nom d'un usager est connu par tout le monde. Il est utilisé pour inviter un autre usager, ou pour afficher l'origine d'un message pendant une session de clavardage.

Un message (classe `Message`) est composé de quatre champs : type de message (valeurs possibles DISCONNECT, INVITE, CHAT, ERROR), identificateur de clavardoir (`String`), identité du client (`String`), et d'un texte. Quand le serveur reçoit un message du client u, le troisième champ du message est parfois l'identificateur du client. Dans les messages transmis par le serveur, le troisième champ est toujours un pseudonyme d'usager. Les messages suivants seront envoyés.

Code	Action	Message	Source	Destinataire(s)
Nc	Requête de nouveau clavardoir avec usager v	(INVITE, `null`, v.nom, `null`)	client u	serveur
Ns	OK : Établissement de clavardoir c avec v, demandé par client u	(INVITE, c, u.nom, `null`)	serveur	clients u, v
Cc	Envoi de message au clavardoir c	(CHAT, c, u.id, m)	client u	serveur
Cs	Transmission de message en clavardoir c	(CHAT, c, u.nom, m)	serveur	participants de c
Qc	Usager u quitte clavardoir c	(DISCONNECT, c, u.id, `null`)	client u	serveur
Qs	Notification de départure en clavardoir c	(DISCONNECT, c, u.nom, `null`)	serveur	participants de c
Dc	Déconnexion du client u	(DISCONNECT, `null`, u.id, `null`)	client u	serveur
Qs	Notification de départure du client u	(DISCONNECT, c, u.nom, `null`)	serveur	chaque clavardoir c avec u
E1	Erreur : usager v non-existant (réponse à **Nc**)	(ERROR, `null`, v.nom, `null`)	serveur	client v
E2	Erreur : clavardoire c non-existant (réponse à **Cc**, **Qc**)	(ERROR, c, `null`, `null`)	serveur	client
E3	Erreur : client c ne participe pas à c (réponse à **Cc**, **Qc**)	(ERROR, c, u.nom, `null`)	serveur	client u

Dans cette première version de l'application, on a un modèle simple pour simuler la communication entre des ordinateurs différents. En particulier, une connexion est implanté en utilisant deux `BlockingQueues` pour assurer la synchronisation entre des *threads* du serveur et des clients sur le même ordinateur. La classe `Connection` implante les méthodes suivantes.

- ⋆ **sendMessageToServer**(`Message` msg)
- ⋆ **sendMessageToClient**(`Message` msg)
- ⋆ `Message` **getMessageFromServer**()
- ⋆ `Message` **getMessageFromClient**()

3 Serveur

Le serveur doit maintenir une connexion à chaque client. Le serveur reçoit des messages pour établir des clavardoirs, pour connecter les usagers aux clavardoirs ou les y déconnecter, et pour transmettre du texte d'un usager à l'autre. Au côté serveur, un procéssus léger est dédié à chaque client pour écouter ses messages. Le procéssus est lancé à l'aide d'un objet `ClientAssistant` qui implante l'interface `Runnable`. Un tel objet est instancié avec la connexion avec un client. La méthode **run**() obsèrve la connexion dans une boucle infinie.

```
public class ClientAssistant implements Runnable
{
    ...
    public void run()
    {
        while (true)
        {
            Message msg = connexion.getMessageFromClient();
            if (msg!=null)
            {
                // gestion de message
            }
            sleep(200L);
            // gestion d'interruptions
        }
    }
}
```

Synchronisation des procéssus. Au serveur, il y a des ressources partagées parmi les procéssus des `ClientAssistants`, comme par exemple le registre de clients, et les associations entre les connexions, identificateurs de clients, et instances de `ClientAssistant`. Il est donc important d'assurer l'exclusion mutuelle quand ces données sont accédées : utilisez la technique de verrouillage avec `synchronized`. En choisissant vos structures de données, notez que les méthodes des classes `Vector` (fontionne comme `ArrayList`) et `Hashtable` sont synchronisées automatiquement.

4 Client

Le client participe au clavardage par un interface graphique. Un procéssus léger est dédié à l'écoute de messages arrivés du serveur : utilisez une extension de `SwingWorker`. La méthode **doInBackground**() obsèrve une connexion dans une boucle infinie. Les messages arrivés sont passés au procéssus d'événements (*Event Dispatch Thread*) par **publish**, où la méthode **process** performe la mise à jour de l'interface graphique.

```
class ... extends SwingWorker<Void,Message>
{
    public Void doInBackground()
    {
        while(true)
        {
            Message msg = connexion.getMessageFromServer();
            if (msg != null) publish(msg);
            sleep(200L);
            // gestion d'interruption: isCancelled()
        }
    }
    public void process(List<Message> messages_arrives)
    {
        // mise à jour du GUI
    }
}
```

Synchronisation des procéssus. Chez le client, la synchronisation de trois procéssus (`main`, EDT [*Event Dispatch Thread*], et `SwingWorker`) ne pose pas trop de problèmes. Avec l'usage de **publish-process**, le GUI n'est jamais accédé à partir d'un autre procéssus que l'EDT, mais il peut être utile de savoir que la méthode **append** de `JTextArea` (utilisé pour afficher les messages dans le clavardoire) peut être appellée de n'importe quel thread.

5 Scénarios

Connexion d'un client. Une nouvelle connexion est établie dans la séquence suivante.

1. Appel de la méthode (définie dans la classe `Server`)
   `public synchronized Connection registerClient(Client client).`
   (Dans une version ultérieure, on ajoutera l'établissement de connexion sur un réseau et pas entre threads...)
2. La méthode **registerClient** crée un objet `ClientAssistant` et lance un nouveau procéssus léger exécutant la boucle de son **run**(). On doit aussi stocker l'assistant associé avec l'identificateur du client (utiliser `Hashtable<String,ClientAssistant>`), et l'association du nom avec l'identificateur (utiliser `Hashtable<String,String>`).

Établissement d'un clavardoir. Un clavardoir est établi quand le `ClientAssistant` reçoit un message

$$(\mathsf{INVITE}, v.\mathsf{nom}, \texttt{null}, \texttt{null})$$

du client u. S'il y a un client v, alors on détermine son objet `ClientAssistant`. Un clavardoire est créé avec les deux connexions (des deux `ClientAssistants`). Il faut envoyer un reçu aux deux clients :

$$(\mathsf{INVITE}, u.\mathsf{nom}, c, \texttt{null}),$$

où c est l'identificateur du clavardoir (choisi par le serveur). Les `ClientAssistants` doivent stocker les clavardoirs actifs, ainsi que leurs identificateurs (utiliser `Hashtable<String,Chat>`). Si le nom v n'est pas connu, alors il faut envoyer un message d'erreur au client u.

1. Usager u demande un nouveau clavardoir en invitant usager v, à l'aide de l'interface graphique.
2. Événement capturé (**actionPerformed**) : envoi de message **Nc**.

3a. Message arrive au serveur : `ClientAssistant` pour u identifie `ClientAssistant` pour v et envoie les messages **Ns**.

3b. Message arrive au serveur : `ClientAssistant` pour u envoie le message d'erreur **E1**.

4a. Message arrive au client u : `SwingWorker` utilise **publish**()

4b. En **process**() chez client u : nouvelle fenêtre (si **Ns**), ou message d'erreur (si **E1**).

5a. Message **Ns** arrive au client v : `SwingWorker` utilise **publish**()

5b. En **process**() chez client u : nouvelle fenêtre

Joindre un clavardoir. Dans cette implantation, il n'est pas possible de joindre un clavardoir après son établisssement. Comme un développement futur, on prévoit la participation d'autres usagers.

Quitter un clavardoir. Le client u peut quitter le clavardoir en envoyant le message

$$(\mathsf{DISCONNECT}, u.\mathsf{id}, c, \texttt{null}).$$

Quand ce message est reçu par le `ClientAssistant`, il doit enlever cette connexion du clavardoire c, et envoyer un message

$$(\mathsf{DISCONNECT}, u.\mathsf{nom}, c, \texttt{null})$$

à tous les autres usagers (dans cette version, un autre usager au plus) qui restent.

Déconnexion d'un usager. Le client u peut se déconnecter du serveur en envoyant le message

$$(\mathsf{DISCONNECT}, u.\mathsf{id}, \texttt{null}, \texttt{null}) :$$

il faut quitter tous ses clavardoirs actifs. Quand `ClientAssistant` reçoit ce message, il transmet un message de déconnexion

$$(\mathsf{DISCONNECT}, u.\mathsf{nom}, c, \texttt{null})$$

à tous les clavardoirs c où ce client participe. Ensuite, le thread de `ClientAssistant` pour u se termine chez le serveur.

Participation dans le clavardoir. Le client u participe à la discussion en envoyant le message

$$(\mathsf{CHAT}, u.\mathsf{id}, c, \mathit{message}).$$

Quand ce message est reçu par le `ClientAssistant`, il transmet le message

$$(\mathsf{CHAT}, u.\mathsf{nom}, c, \mathit{message})$$

à tous les participants du clavardoir, étant donné que u fait parti du clavardoire c.

Erreurs. Si le client u essaie de déconnecter d'un clavardoir c qui n'existe pas ou dont il ne fait pas partie, le serveur répond avec un message d'erreur

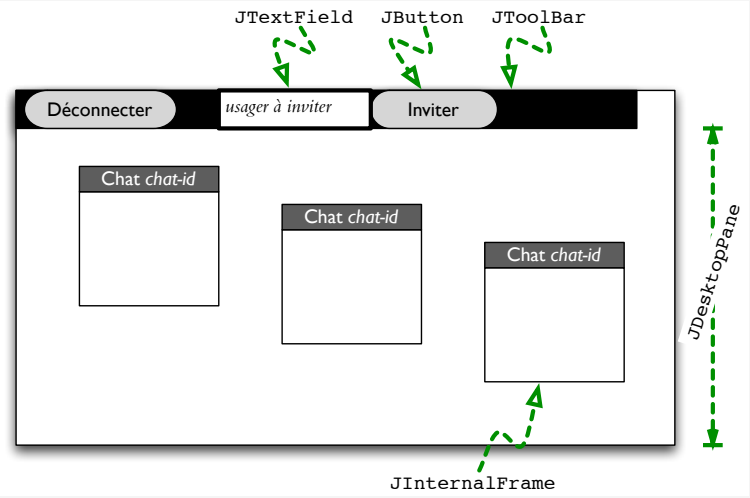
$$(\mathsf{ERROR}, \texttt{null}, c, \texttt{null})$$

ou

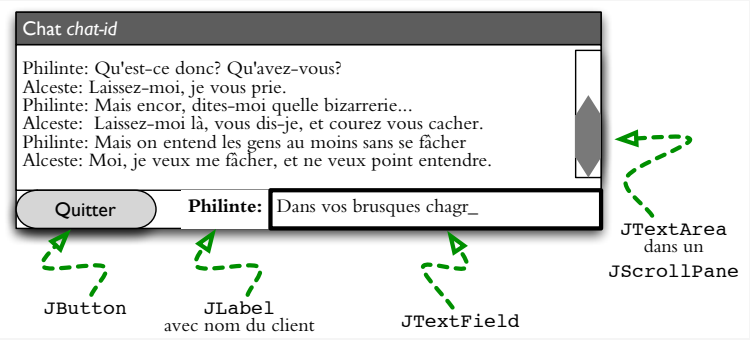
$$(\mathsf{ERROR}, u.\mathsf{nom}, c, \texttt{null})$$

.

6 Interface usager du client



En haut, il se trouve une barre d'outils (`JToolBar`) avec des boutons pour se déconnecter du serveur (Déconnecter) et pour commenceur un nouveau clavardoir (Inviter), ainsi qu'un champ de texte pour y mettre le nom de l'usager à inviter. Chaque clavardoir actif est affiché par un `JInternalFrame` dans un `JDesktopPane`.



Un clavardoir est affiché dans un `JInternalFrame` dont le titre contient l'identificateur du clavardoir. Les composants principaux sont les suivants : un `JTextArea` qui affiche les messages, un `JTextField` où l'usager peut taper son message, et un bouton pour quitter le clavardoir.

La fenêtre d'un clavardoire doit être enlevé du GUI quand l'usager quitte le clavardoire ou se déconnecte du serveur. Lors de déconnexion il faut aussi désactiver les boutons Inviter et Déconnecter (utiliser **setEnabled**).

Les messages sont envoyés vers le serveur en réponse d'actions de l'usager.

- ⋆ bouton Inviter : **actionPerformed** du listener attaché envoie un message au serveur (**Nc**) — le nom du client invité est le contenu du champ de texte à côté du bouton.
- ⋆ bouton Déconnecter : **actionPerformed** du listener attaché envoie un message au serveur (**Dc**) et ferme toutes les fenêtres associées avec le clavardoir.
- ⋆ `JTextField` du clavardoir : `actionPerformed` du listener attaché envoie un message au serveur (**Cc**), et efface le contenu.
- ⋆ bouton Quitter : **actionPerformed** du listener attaché envoie un message au serveur (**Qc**) et ferme la fenêtre associée avec le clavardoir.

Le GUI est mis à jour en réponse de messages arrivés à partir du serveur.

- ⋆ **Ns** : il faut ouvrir une fenêtre pour un nouveau clavardoir.
- ⋆ **Cs** : il faut ajouter (**append**) le message au `JTextArea` du clavardoir, en le précédant avec le nom de l'interlocuteur.
- ⋆ **Qs** : ajouter un message «Client xxx a quitté le clavardoir.» au `JTextArea` du clavardoir.
- ⋆ **E1**, **E1**, **E3** : il faut afficher un message d'erreur (utiliser `JOptionPane`).

7 Tests

L'application est lancé à l'aide de la classe `ChatTester`. Les arguments de la ligne de commande sont les noms des usagers. Cette application montre les interfaces usagers par un `JTabbedPane`, et lance les procéssus légers du serveur.

8 Travail et soumission

Une partie du code vous est fournie (`Chat.jar` sur le site Web du cours). Vous devez développer la reste, en définissant de nouvelles classes quand nécessaire. Vous devez travailler avec les classes suivantes.

- ⋆ La classe `ChatTester` sert à tester votre implantation : il est complètement développé.
- ⋆ Les classes `Message` et `Connection` sont développées complètement.
- ⋆ La classe `Client` est développée complètement, mais si vous voulez, vous pouvez changer l'implantation.
- ▶ La classe `Server` est partiellement développé
- ▶ Il faut écrire la classe `ClientAssistant` pour gérer les messages au côté serveur.
- ▶ Il faut écrire la classe `ClientGUI` qui est l'interface usager du client.
- ▶ Vous aurez probablement besoin d'autres classes (p.e., pour représenter un clavardoir) : probablement, juste deux ou trois.

Travail estimé : 400–600 lignes de code (incluant commentaires et lignes vides pour lisibilité).

Compilez un archive jar nommé `Chat.jar` qui contient les classes du package `chat`. L'archive doit inclure les sources `.java` et les classes compilées `.class`. Dans chaque fichier de source, indiquez le membre de l'équipe qui l'a développé, en javadoc (sous `@author` dans la documentation de la classe). Dans votre courrier éléctronique, donnez le nom de tous les membres de l'équipe.

Envoyez le fichier comme attachement dans un courriel à `dift1025@iro.umontreal...`. Votre programme doit être fonctionnel, clair, commenté et original (plagiat=0).