Mini projet sur les collecticiels

1) Introduction :

La conception des collecticiels (logiciels de collaboration) commence généralement par la spécification suivant le trèfle fonctionnel du système de collaboration à développer. Le trèfle fonctionnel des collecticiels est constitué de 3 espaces : (1) Espace de communication, (2) Espace de coordination, (3) Espace de Production d'objets partagés (appelé également espace de coopération).

Le but de ce mini projet est la concrétisation de l'aspect conceptuel (vu en cours) des espaces du trèfle fonctionnel des collecticiels. Il s'agit plus particulièrement de développer une interface homme machine permettant à deux utilisateurs (ou plus) de communiquer par chat (simple échange de message) et de co-produire sur un même espace (concept du tableau blanc partagé). On supposera que la coordination des tâches et des utilisateurs peut être faite directement par le biais de la communication.

2) Déroulement du mini projet :

Le mini projet se fera en binôme et pendant 3 séances de TP (les étudiants peuvent travailler sur le mini projet en dehors des séances de TP) et la permanence est assurée par un enseignant pendant chaque TP.

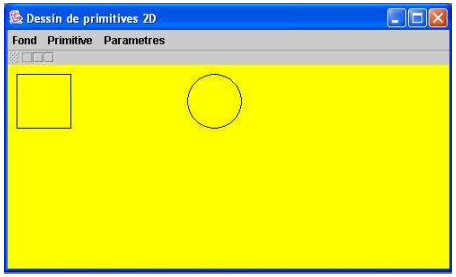
Ce mini projet vous permettra d'utiliser toutes vos connaissances acquises en informatique et les mettre en pratique autour d'un objectif commun qui est la réalisation d'un **collecticiel**. Plus particulièrement les modules suivants sont nécessaires : Le réseau (client - serveur), et la programmation orientée objet en java (vue en II74).

A la fin des 3 séances de TP, chaque binôme devra rendre **un rapport** (un compte rendu) du travail qu'ils ont réalisé, les **codes sources** du collecticiel développé ainsi qu'une **notice d'utilisation**.

3) Existants :

3.1) IHM en java (TP2 du II74)

Utiliser les programmes java que vous avez déjà réalisés en TP2 du module II74 permettant d'afficher une fenêtre de dessin ainsi qu'une fenêtre de saisie de texte (figure ci-dessous). Sinon les programmes sont disponibles sur le serveur **ens-unix (/export/home/otmane/II86/projet/IHM/).**



3.2) Communication Client – Serveur en java

Une application client - serveur en java est constituée de deux programmes suivants (que vous récupérer à l'adresse suivante : **/export/home/otmane/II86/projet/CLIENT_SERVEUR/**). Récupérer ce répertoire dans votre home directory. Les sources ainsi qu'une documentation sont également à votre disposition en Annexe.

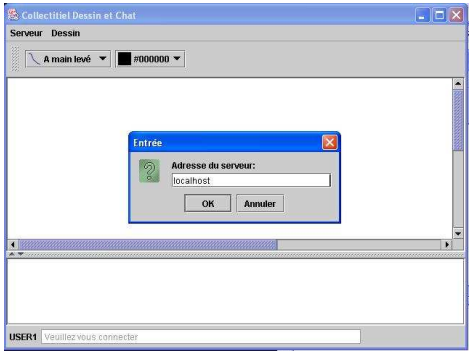
3.3) Un exemple de collecticiel (tableau blanc partagé)

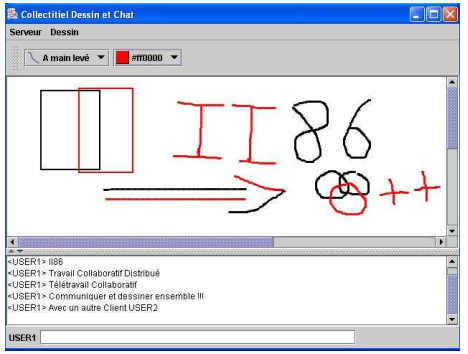
Vous trouverez dans le répertoire **/export/home/otmane/II86/projet/EXEMPLE/** un exemple de collecticiel (sources + exécutables) basé sur le concept de tableau blanc partagé (illustré dans la figure ci-dessous). Récupérez-le dans votre home directory.

a) Lancement du serveur :
- Positionnez-vous dans le répertoire **serveur**
- Compiler le programme serveur (**$ javac NetDrawServerMain.java**)
- Exécuter le programme serveur (**$ java NetDrawServerMain**) : le serveur se met en écoute sur le port « 1515 ».

b) Lancement des clients :
- Positionnez-vous dans le répertoire **client**
- Compiler le programme client (**$ javac NetDrawClientMain.java**)
- Exécuter le programme client (**$ java NetDrawClientMain**)
- Une fenêtre apparaît vous demandant de saisir un *pseudo*

- La fenêtre principale (figure ci-dessous à gauche) apparaît
- Cliquez sur le menu **Serveur/Connecter** afin de spécifier le nom (ou l'adresse IP) de la machine du programme serveur.





- Répétez la procédure pour les autres clients et vous pouvez dessiner ensemble à distance sur un même tableau blanc.

4) Travail à faire :

*Objectif principale ➔Transformer l'IHM en java donnée en **3.1** en un véritable collecticiel permettant à deux utilisateurs de communiquer et de partager un même espace de dessin.*

Q1 Compiler et exécuter les deux programmes. Commenter les deux programmes **Serveur.java et Client.java** et commenter le résultat obtenu (aidez-vous de la documentation fournie en annexe)

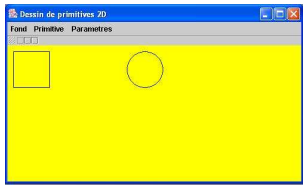
Q2 Dans l'exemple précédent, le client et le serveur communiquent alternativement. Programmer une version où le client et le serveur communiquent simultanément. Chacun envoie à l'autre un message, puis chacun lit le message de l'autre.

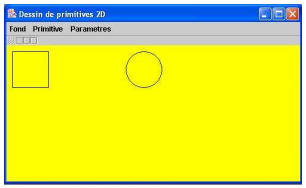
Q3 Utiliser l'interface graphique développée en java (**/export/home/otmane/II86/projet/IHM/**) et incorporer le serveur et le client dans les programmes afin de permettre une communication par **chat** entre deux utilisateurs.

Q4 Développer le serveur de façon à ce qu'il puisse gérer plusieurs clients simultanément.

Q5 Etudier l'exemple de collecticiel (tableau blanc partagé) mis à votre disposition (**/export/home/otmane/II86/projet/EXEMPLE/**) et expliquez brièvement le rôle de chacune des classes décrites dans la partie **serveur** et dans la partie **client.**

Q6 Transformer la fenêtre de « Dessin de primitive 2D » en un espace de dessin partagé (co-production) entre deux utilisateurs.





Annexe :

Les Sources Client - Serveur en java (dans : /export/home/otmane/II86/projet/CLIENT_SERVEUR/)

Serveur.java

```
import java.io.*;
import java.net.*;

public class Serveur {
  static final int port = 8080;

  public static void main(String[] args) throws Exception {
      ServerSocket s = new ServerSocket(port);
      Socket soc = s.accept();

      // Un BufferedReader permet de lire par ligne.
       BufferedReader plec = new BufferedReader( new InputStreamReader(soc.getInputStream()) );

      // Un PrintWriter possède toutes les opérations print classiques.
      // En mode auto-flush, le tampon est vidé (flush) à l'appel de println.
       PrintWriter pred = new PrintWriter(new BufferedWriter(new OutputStreamWriter(soc.getOutputStream())), true);

      while (true) {
        String str = plec.readLine();          // lecture du message
        if (str.equals("END")) break;
        System.out.println("ECHO = " + str);   // trace locale
        pred.println(str);                     // renvoi d'un écho
      }
      plec.close();
      pred.close();
      soc.close();
  }
}
```

Client.java

```
import java.io.*;
import java.net.*;
/** Le processus client se connecte au site fourni dans la commande
 *   d'appel en premier argument et utilise le port distant 8080.
 */
public class Client {
  static final int port = 8080;

  public static void main(String[] args) throws Exception {
      Socket socket = new Socket(args[0], port);
      System.out.println("SOCKET = " + socket);

      BufferedReader plec = new BufferedReader(new InputStreamReader(socket.getInputStream()));

      PrintWriter pred = new PrintWriter(new BufferedWriter(new OutputStreamWriter(socket.getOutputStream())), true);

      String str = "bonjour";
      for (int i = 0; i < 10; i++) {
```

```
        pred.println(str);          // envoi d'un message
        str = plec.readLine();      // lecture de l'écho
      }
      System.out.println("END");     // message de terminaison
      pred.println("END") ;
      plec.close();
      pred.close();
      socket.close();
  }
}
```

Documentation sur la communication client - serveur par les sockets en java

L'interface socket BSD est l'interface de programmation réseau la plus courante. Cette interface permet de façon classique d'utiliser une communication par socket selon le schéma habituel client-serveur. L'interface Java des sockets (package `java.net`) offre un accès simple aux sockets sur IP.

Les classes

Plusieurs classes interviennent lors de la réalisation d'une communication par sockets. La classe `java.net.InetAddress` permet de manipuler des adresses IP. La classe `java.net.SocketServer` permet de programmer l'interface côté serveur en mode connecté. La classe `java.net.Socket` permet de programmer l'interface côté client et la communication effective par flot via les sockets. Les classes `java.net.DatagramSocket` et `java.net.DatagramPacket` permettent de programmer la communication en mode datagramme.

1. La classe `java.net.InetAddress`

Cette classe représente les adresses IP et un ensemble de méthodes pour les manipuler. Elle encapsule aussi l'accès au serveur de noms DNS.

```
public class InetAddress implements Serializable
```

1.1. Les opérations de la classe `InetAddress`

Conversion de nom vers adresse IP :

Un premier ensemble de méthodes permet de créer des objets adresses IP.

```
public static InetAddress getLocalHost() throws UnknownHostException
```

Cette méthode renvoie l'adresse IP du site local d'appel.

```
public static InetAddress getByName(String host) throws UnknownHostException
```

Cette méthode construit un nouvel objet `InetAddress` à partir d'un nom textuel de site. Le nom du site est donné sous forme symbolique (`bach.enseeiht.fr`) ou sous forme numérique (147.127.18.03).

Enfin,

```
public static InetAddress[] getAllByName(String host) throws UnknownHostException
```

permet d'obtenir les différentes adresses IP d'un site.

Conversions inverses

Des méthodes applicables à un objet de la classe `InetAddress` permettent d'obtenir dans divers formats des adresses IP ou des noms de site. Les principales sont :

`public String getHostName()` obtient le nom complet correspondant à l'adresse IP

`public String getHostAddress()` obtient l'adresse IP sous forme %d.%d.%d.%d
`public byte[] getAddress()` obtient l'adresse IP sous forme d'un tableau d'octets.

2. La classe `ServerSocket`

Cette classe implante un objet ayant un comportement de serveur via une interface par socket.

```
public class java.net.ServerSocket
```

Une implantation standard du service existe mais peut être redéfinie en donnant une implantation explicite sous la forme d'un objet de la classe `java.net.SocketImpl`. Nous nous contenterons d'utiliser la version standard.

2.1 Le constructeur `ServerSocket`

```
public ServerSocket(int port) throws IOException
public ServerSocket(int port, int backlog) throws IOException
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws
IOException
```

Ces constructeurs créent un objet serveur à l'écoute du port spécifié. La taille de la file d'attente des demandes de connexion peut être explicitement spécifiée via le paramètre `backlog`. Si la machine possède plusieurs adresses, on peut aussi restreindre l'adresse sur laquelle on accepte les connexions.

Appels système :

Ce constructeur correspond à l'utilisation des primitives `socket`, `bind` et `listen`.

2.2 Les opérations de la classe `ServerSocket`

Nous ne retiendrons que les méthodes de base. La méthode essentielle est l'acceptation d'une connexion d'un client :

```
public Socket accept() throws IOException
```

Cette méthode est bloquante, mais l'attente peut être limitée dans le temps par l'appel préalable de la méthode `setSoTimeout`. Cette méthode prend en paramètre le délai de garde exprimé en millisecondes. La valeur par défaut 0 équivaut à l'infini. À l'expiration du délai de garde, l'exception `java.io.InterruptedIOException` est levée.

```
public void setSoTimeout(int timeout) throws SocketException
```

La fermeture du socket d'écoute s'exécute par l'appel de la méthode `close`.

Enfin, les méthodes suivantes retrouvent l'adresse IP ou le port d'un socket d'écoute :

```
public InetAddress getInetAddress()
public int getLocalPort()
```

3 La classe `java.net.Socket`

La classe `java.net.Socket` est utilisée pour la programmation des sockets connectés, côté client et côté serveur.

```
public class java.net.Socket
```

Comme pour le serveur, nous utiliserons l'implantation standard bien qu'elle soit redéfinissable par le développement d'une nouvelle implantation de la classe `java.net.SocketImpl`.

3.1 Constructeurs

Côté serveur, la méthode `accept` de la classe `java.net.ServerSocket` renvoie un socket de service connecté au client.
Côté client, on utilise:

```
public Socket(String host, int port) throws UnknownHostException, IOException
public Socket(InetAddress address, int port) throws IOException
public Socket(String host, int port, InetAddress localAddr, int localPort)
            throws UnknownHostException, IOException
public Socket(InetAddress addr, int port, InetAddress localAddr, int localPort)
            throws IOException
```

Les deux premiers constructeurs construisent un socket connecté à la machine et au port spécifiés. Par défaut, la connexion est de type TCP fiable. Les deux autres interfaces permettent en outre de fixer l'adresse IP et le numéro de port utilisés côté client (plutôt que d'utiliser un port disponible quelconque).

Appels système : Ces constructeurs correspondent à l'utilisation des primitives `socket`, `bind` (éventuellement) et `connect`.

3.2 Opérations de la classe `Socket`

La communication effective sur une connexion par socket utilise la notion de flots de données (`java.io.OutputStream` et `java.io.InputStream`). Les deux méthodes suivantes sont utilisées pour obtenir les flots en entrée et en sortie.

```
public InputStream  getInputStream()  throws IOException
public OutputStream getOutputStream() throws IOException
```

Les flots obtenus servent de base à la construction d'objets de classes plus abstraites telles que `java.io.DataOutputStream` et `java.io.DataInputStream` (pour le JDK1), ou `java.io.PrintWriter` et `java.io.BufferedReader` (JDK2) (cf exemple 2).

Une opération de lecture sur ces flots est bloquante tant que des données ne sont pas disponibles. Cependant, il est possible de fixer une délai de garde à l'attente de données (similaire au délai de garde du socket d'écoute : levée de l'exception `java.io.InterruptedIOException`) :

```
public void setSoTimeout(int timeout) throws SocketException
```

Un ensemble de méthodes permet d'obtenir les éléments constitutifs de la liaison établie :

`public InetAddress getInetAddress()`	fournit l'adresse IP distante
`public InetAddress getLocalAddress()`	fournit l'adresse IP locale
`public int getPort()`	fournit le port distant
`public int getLocalPort()`	fournit le port local

L'opération `close` ferme la connexion et libère les ressources du système associées au socket.

4 Socket en mode datagramme `DatagramSocket`

La classe `java.net.DatagramSocket` permet d'envoyer et de recevoir des paquets (datagrammes UDP). Il s'agit donc de messages non fiables (possibilités de pertes et de duplication), non ordonnés (les messages peuvent être reçus dans un ordre différent de celui d'émission) et dont la taille (assez faible -- souvent 4Ko) dépend du réseau sous-jacent.

```
public class java.net.DatagramSocket
```

4.1 Constructeurs

```
public DatagramSocket() throws SocketException
public DatagramSocket(int port) throws SocketException
```

Construit un socket datagramme en spécifiant éventuellement un port sur la machine locale (par défaut, un port disponible quelconque est choisi).

4.2 Émission/réception

```
public void send(DatagramPacket p) throws IOException
public void receive(DatagramPacket p) throws IOException
```

Ces opérations permettent d'envoyer et de recevoir un paquet. Un paquet est un objet de la classe `java.net.DatagramPacket` qui possède une zone de données et (éventuellement) une adresse IP et un numéro de port (destinataire dans le cas `send`, émetteur dans le cas `receive`). Les principales méthodes sont :

```
public DatagramPacket(byte[] buf, int length)
public DatagramPacket(byte[] buf, int length, InetAddress address, int port)

public InetAddress getAddress()
public int getPort()
public byte[] getData()
public int getLength()

public void setAddress(InetAddress iaddr)
public void setPort(int iport)
public void setData(byte[] buf)
public void setLength(int length)
```

Les constructeurs renvoient un objet pour recevoir ou émettre des paquets. Les accesseurs `get*` permettent, dans le cas d'un `receive`, d'obtenir l'émetteur et le contenu du message. Les méthodes de modification `set*` permettent de changer les paramètres ou le contenu d'un message pour l'émission.

4.3 Connexion

Il est possible de « connecter » un socket datagramme à un destinataire. Dans ce cas, les paquets émis sur le socket seront toujours pour l'adresse spécifiée. La connexion simplifie l'envoi d'une série de paquets (il n'est plus nécessaire de spécifier l'adresse de destination pour chacun d'entre eux) et accélère les contrôles de sécurité (ils ont lieu une fois pour toute à la connexion). La « déconnexion » enlève l'association (le socket redevient disponible comme dans l'état initial).

```
public void connect(InetAddress address, int port)
public void disconnect()
```

4.4 Divers

Diverses méthodes renvoient le numéro de port local et l'adresse de la machine locale (`getLocalPort` et `getLocalAddress`), et dans le cas d'un socket connecté, le numéro de port distant et l'adresse distante (`getPort` et `getInetAddress`). Comme précédemment, on peut spécifier un délai de garde pour l'opération `receive` avec `setSoTimeout`. On peut aussi obtenir ou réduire la taille maximale d'un paquet avec `getSendBufferSize`, `getReceiveBufferSize`, `setSendBufferSize` et `setReceiveBufferSize`.

Enfin, n'oublions pas la méthode `close` qui libère les ressources du système associées au socket.