TP 3 : J2ME

1. Introduction :

Le J2ME Wireless Toolkit (WTK) est une boite à outil qui permet de créer des applications pour des téléphones mobiles et autres systèmes sans fil. Bien qu'il soit basé sur le profile MIDP 2.0, le J2ME Wireless Toolkit supporte également un certain nombre de packages optionnels utiles, lui donnant ainsi les dimensions d'un environnement de développement large et efficace.

Le WTK permet de créer de nouveaux projets, de les compiler et de les exécuter grâce à l'application « ktoolbar », mais pas d'éditer le code source correspondant. La documentation du WTK et de son API se trouvent dans le répertoire « WTK\docs ».

Toutes les applications pour le WTK se trouve dans le répertoire « WTK\apps », y compris les projets que vous allez créer.

2. Utilisation de WTK :

- Ouvrir le projet « Games », compiler le avec la commande « build », puis exécuter le avec la commande « run ». Sélectionner le jeu « PushPuzzle », et démarrer le. Vous devez déplacer le rond bleu pour pousser les carrés rouges dans les emplacements encadrés.
- Vous pouvez essayer également les projets « Demos », « Demo3D », « PDAPDemo », « PhotoAlbum », « UIDemo ».
- Ouvrir maintenant le projet « BluetoothDemo », et après l'avoir construit, exécutez 2 instances de l'émulateur de téléphone. Le premier sera le serveur, le deuxième le client. Sélectionner ensuite les images à publier côté serveur, et les visualiser sur le téléphone client.
- Tester également la demo « NetworkDemo », et exécuter 2 clients en mode « Socket », l'un serveur et l'autre client. Envoyer ensuite du texte d'un téléphone vers l'autre.
- Pour terminer, essayer la demo « WMADemo » qui permet d'envoyer des SMS/MMS. Le premier téléphone sera configuré en mode réception de SMS, alors que le deuxième enverra les SMS. Au moment de l'envoi du SMS, indiquer l'adresse de destination du téléphone destinataire, qui est unique pour chaque téléphone émulé et qui est indiquée dans la barre de titre (exemple : +5550000 pour le premier téléphone émulé).

3. Création d'une mini MIDLET :

- Pour créer votre premier projet, cliquer sur « New Project... », indiquer le nom du projet « Tiny » et le nom de la classe MIDLet correspondante (`TinyMIDlet`)
- Editer le code source de votre application en créant un fichier « WTK22/apps/Tiny/src/TinyMIDlet.java », avec le contenu suivant :

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.MIDlet;

public class TinyMIDlet extends MIDlet implements CommandListener {
```

```
	public void startApp() {
		Display display = Display.getDisplay(this);
		Form mainForm = new Form("TinyMIDlet");
		mainForm.append("Welcome to the world of MIDlets!");
		Command exitCommand = new Command("Exit", Command.EXIT, 0);
		mainForm.addCommand(exitCommand);
		mainForm.setCommandListener(this);
		display.setCurrent(mainForm);
	}
	public void pauseApp () {}
	public void destroyApp(boolean unconditional) {}
	public void commandAction(Command c, Displayable s) {
		if (c.getCommandType() == Command.EXIT)
		notifyDestroyed();
	}
}
```

- Compiler et exécuter votre programme.
- Créer une nouvelle commande dans votre MIDlet pour afficher un nouvel écran avec juste le message « Hello World ! ». Pour cela, créer une commande « hwCommand » qui sera une instance de la classe Commande, avec l'identifieur « Command.ITEM ». Dans la méthode « commandAction », vous pourrez alors tester si la référence de la commande passée en paramètre correspond à la commande « hwCommand », ou comparer les labels correspondants.
- Ajouter une nouvelle commande « back » lorsque le message « Hello world! » est affiché, de façon à pouvoir revenir à l'écran initial de l'application.
- Avant l'exécution de votre application, vous pouvez activer un des outils de monitoring permettant d'analyser l'utilisation de la mémoire, du réseau ou le temps passé à exécuter les différentes méthodes de votre application (profiler). Pour activer ces outils, aller dans « Edit/preferences » puis onglet « Monitor », activer ensuite « Memory Monitor », « Network Monitor » ou « Profiler » pour récupérer ces informations.

4. Création d'une application client/serveur utilisant le bluetooth :

Dans cet exemple, nous allons établir une connexion entre deux appareils bluetooth : un serveur qui met à disposition un service spécifique identifié par une chaîne hexadécimale, et un client qui utilise ce service spécifique en spécifiant la même chaîne hexadécimale. Le serveur se met à l'écoute d'un envoi de message et l'affiche à l'écran dès qu'il reçoit quelque chose. De son côté le client a pour objectif d'effectuer l'envoi du message au serveur.

4.1. Création du serveur

- Créer une première application TP3-BTServer qui supporte le bluetooth (paramètres du projet), qui hérite de MIDlet et qui implémente l'interface Runnable de façon à pouvoir exécuter une tâche de fond tout en interagissant avec le téléphone.
- Afficher le nom et l'adresse de l'appareil bluetooth local dans la méthode run() (cf partie 5 pour les explications sur le fonctionnement du bluetooth avec j2me). Cet affichage pourra se faire dans la console de WTK par « System.out.println » ou dans l'écran du téléphone portable en utilisant la méthode « append » de la Forme associée au Display. Insérer les méthodes bluetooth dans des blocs de capture d'exception et tester votre application.

- Rendre l'appareil local découvrable par la méthode `setDiscoverable(DiscoveryAgent.GIAC).`
- Créer un objet de connexion `StreamConnectionNotifier` pour le serveur qui accepte des

connexions clients par la méthode `open` de `Connector` :
`(StreamConnectionNotifier)Connector.open("btspp://localhost:86b4d")`

- Accepter une connexion d'un client `StreamConnection` par un appel de la méthode `acceptAndOpen()` sur le `StreamConnectionNotifier` précédemment créé.
- Ouvrir alors le flux d'entré des données en créant un `InputStream` par un appel de la méthode `openInputStream` sur le précédent `StreamConnection`.
- Lire chaque octet du flux d'entrée par la méthode `(int)read` jusqu'à ce que la valeur -1 qui signale la fin du flux soit rencontrée. Pour transformer ces octets lus dans le flux d'entrée en une chaîne de caractères, utiliser un `ByteArrayOutputStream` qu'il faudra alimenter avec chaque octet lu par un appel à la méthode `write(int)`. Utiliser alors la méthode `toString` pour convertir le `ByteArrayOutputStream` en String affichable.
- Fermer ensuite toutes les ressources qui ont été ouvertes par un appel à la méthode `close` pour le `InputStream` , le `StreamConnection` et le `StreamConnectionNotifier`.

4.2. Création du client

- Créer une deuxième application TP3-BTClient qui supporte le bluetooth et implémente également Runnable.
- Définir un `DiscoveryAgent` associé à l'appareil local, par un appel à `getDiscoveryAgent` (sur l'instance du `LocalDevice`).
- Sélectionner un service pour cet agent par la méthode `selectService` méthode, en passant les 3 paramètres :
  - `new UUID("86b4d",false)`
  - `ServiceRecord.NOAUTHENTICATE_NOENCRYPT`
  - `false`
- Mémoriser la chaîne de connexion (String) associé à ce service, qui est renvoyée par la méthode `selectService`.
- Une fois la connexion établie, il ne reste plus qu'à envoyer un message. Pour cela, ouvrir un `StreamConnection` par la méthode `open()` de `Connector` en passant en paramètre la chaîne de connexion précédente. Il faut également caster le retour de `open()` en un `StreamConnection`.
- Associer un flux de sortie `OutputStream` au `StreamConnection` précédent par un appel à `openOutputStream()`.
- Utiliser la méthode `write(String)` pour écrire dans le flux de sortie de la façon suivante : `monOutputStream.write(monMessage.getBytes())` où `monMessage` est un `String`.
- Fermer le flux de sortie et le `StreamConnection` avec la méthode `close`.
- Tester votre application.

4.3. Améliorations de l'application Client/Serveur

- Transformer vos 2 applications client/serveur en une seule application qui propose à l'utilisateur de jouer le rôle du client ou du serveur.

- Modifier votre application pour qu'il soit possible de dialoguer entre les 2 appareils bluetooth par envoi de messages.
- Faire en sorte qu'il ne soit plus nécessaire de spécifier qui joue le rôle de client ou de serveur, chaque application jouera simultanément le rôle de client ou de serveur, suivant le contexte. Exemple : Lorsqu'une instance de l'application se lance, elle se met en mode client et essaie de se connecter à un serveur, si il n'existe pas de serveur disponible, alors elle se met en mode serveur et attend des requêtes de clients.

- Etudier l'exemple BluetoothDemo du WTK et comparer les différences par rapport à l'application que vous venez de réaliser.

5. Description du fonctionnement du bluetooth sur J2ME :

La structure d'une application Bluetooth est constituée de 5 parties : initialisation de la pile, gestion des devices, découverte des devices, découverte des service et communication.

Initialisation de la pile

La pile bluetooth contrôle les appareils bluetooth et doit donc être initialisée. L'initialisation comprend un certain nombre d'étapes permettant de rendre disponible l'appareil pour des communications sans fil. Malheureusement, les spécifications bluetooth laissent les vendeurs implémenter le Bluetooth Command Center (BCC qui permet de paramétrer les communications bluetooth et de gérer l'accès concurrentiel des applications aux services bluetooth), et différents vendeurs gèrent l'initialisation de la pile différemment : sur certains appareils, cela peut être une application avec une interface graphique, sur d'autres cela peut être une série de réglages qui ne peuvent pas être changés par l'utilisateur.

Exemple de réglage pour les solutions Atinav's Java Bluetooth :

```
...
// set the port number
BCC.setPortNumber("COM1");
// set the baud rate
BCC.setBaudRate(50000);
// set the connectable mode
BCC.setConnectable(true);
// set the discovery mode to Limited Inquiry Access Code
BCC.setDiscoverable(DiscoveryAgent.LIAC);
...
```

Gestion du « device »

L'API Java Bluetooth contient les classes `LocalDevice` et `RemoteDevice`, qui permettent d'accéder à la gestion de l'appareil défini dans le profil d'accès générique. `LocalDevice` dépend de la classe `javax.bluetooth.DeviceClass` pour récupérer le type de l'appareil et les services qu'il offre. La classe `RemoteDevice` représente un appareil distant et fournit des méthodes pour récupérer des informations sur l'appareil, notamment son nom et son adresse bluetooth. Le code suivant permet de récupérer les informations de l'appareil local :

```
...
// retrieve the local Bluetooth device object
LocalDevice local = LocalDevice.getLocalDevice();
// retrieve the Bluetooth address of the local device
String address = local.getBluetoothAddress();
// retrieve the name of the local Bluetooth device
String name = local.getFriendlyName();
...
```

On peut également récupérer les informations à partir d'un appareil distant :

```
...
// retrieve the device that is at the other end of
// the Bluetooth Serial Port Profile connection,
// L2CAP connection, or OBEX over RFCOMM connection
RemoteDevice remote =
    RemoteDevice.getRemoteDevice(
        javax.microedition.io.Connection c);
// retrieve the Bluetooth address of the remote device
String remoteAddress = remote.getBluetoothAddress();
// retrieve the name of the remote Bluetooth device
String remoteName = local.getFriendlyName(true);
...
```

La classe `RemoteDevice` fournit des méthodes pour authentifier, autoriser ou crypter les données à transférer entre l'appareil local et distant.

Découverte de « device »

Les appareils étant mobiles, ils ont besoin d'un mécanisme leur permettant de découvrir d'autres appareils pour pouvoir profiter de leurs services. La classe de l'API Bluetooth `DiscoveryAgent` et l'interface `DiscoveryListener` fournissent les services de recherche nécessaires.

Un appareil Bluetooth peut utiliser un `DiscoveryAgent` pour obtenir une liste des appareils accessibles de 3 manières différentes :

- La méthode `DiscoveryAgent.startInquiry` met l'appareil en mode recherche de service. Dans ce mode, l'application doit spécifier un écouteur d'événement qui répondra aux événements de découverte d'appareils bluetooth. La méthode `DiscoveryListener.deviceDiscovered` est appelée à chaque fois qu'une recherche trouve un appareil. Quand la recherche est terminée, `DiscoveryListener.inquiryCompleted` est invoquée.

- Si l'appareil ne veut pas attendre que les appareils soient découverts, il peut uiliser la méthode `DiscoveryAgent.retrieveDevices` pour récupérer une liste existante. En fonction des paramètres passés, la méthode renverra soit une liste des appareils trouvés lors de précédentes recherches, ou une liste d'appareils pré-enregistré (*pre-known*) que l'appareil local aura indiqué au BCC (contact fréquent).

Les 3 morceaux de code suivants illustrent les 3 possibilités :

```
...
// retrieve the discovery agent
DiscoveryAgent agent = local.getDiscoveryAgent();
// place the device in inquiry mode
boolean complete = agent.startInquiry();
...
```

```
...
// retrieve the discovery agent
DiscoveryAgent agent = local.getDiscoveryAgent();
// return an array of pre-known devices
RemoteDevice[] devices =
   agent.retrieveDevices(DiscoveryAgent.PREKNOWN);
...
```

```
...
// retrieve the discovery agent
DiscoveryAgent agent = local.getDiscoveryAgent();
// return an array of devices found in a previous inquiry
RemoteDevice[] devices =
    agent.retrieveDevices(DiscoveryAgent.CACHED);
...
```

Découverte de service

Une fois que l'appareil local a découvert au moins un appareil bluetooth, il peut commencer à rechercher des services disponibles : des applications bluetooth qu'il peut utiliser pour accomplir des tâches utiles. Etant donné que la découverte de service est similaire à la découverte d'appareils bluetooth, `DiscoveryAgent` fournit également des méthodes de découverte de service sur un appareil Bluetooth serveur, et d'initialisation des transactions sur ces services. Il est à noter que l'API fournit des mécanismes de recherche de service sur les périphériques distants, mais pas sur le périphérique local.

Enregistrement d'un service

Avant qu'un service puisse être découvert, il doit tout d'abord être enregistré sur le périphérique serveur bluetooth. Le serveur est responsable de :

- Créer un enregistrement de service qui décrit ce service offert.
- Ajouter cet enregistrement de service à la Base de Données de Découverte de Service du Serveur (Service Discovery DataBase : SDDB), de façon à ce qu'il soit visible et disponible pour des clients potentiels.
- Enregistrer les mesures de sécurité Bluetooth associées avec le service.

- Accepter les connexions des clients
- Mettre à jour l'enregistrement du service dans la SDDB quand les attributs du service changent.
- Supprimer ou désactiver l'enregistrement du service dans la SDDB quand le service n'est plus disponible.

Les fragments de code suivants illustrent les efforts nécessaires à l'enregistrement d'un service par l'utilisation de l'API java pour le Bluetooth:

1. Pour créer un nouvel enregistrement de service représentant le service, invoquer la méthode `Connector.open` avec un argument de type URL de connexion pour un serveur, et caster le résultat en un `StreamConnectionNotifier` qui représente le service :

```
...
StreamConnectionNotifier service =
    (StreamConnectionNotifier) Connector.open("someURL");
```

2. Récupérer l'enregistrement de service créé par le serveur bluetooth :

```
ServiceRecord sr = local.getRecord(service);
```

3. Indiquer que le service est prêt à accepter des connexions clients :

```
StreamConnection connection =
    (StreamConnection) service.acceptAndOpen();
```

   Remarque : `acceptAndOpen` est bloquant jusqu'à la connexion d'un client.

4. Quand le serveur est prêt à se terminer, fermer la connexion et supprimer l'enregistrement du service.

```
service.close();
...
```

Communication

Pour qu'un périphérique local utilise un service sur un appareil distant, les 2 appareils doivent partager un protocole de communication commun. De façon à ce que les applications puissent accéder à une large gamme de services Bluetooth, l'API java pour Bluetooth fournit des mécanismes permettant les connexions à n'importe quel service qui utilise RFCOMM, L2CAP ou OBEX comme protocole. Si un service utilise un autre protocole (comme par exemple TCP/IP)

situé au-dessus d'un de ces protocoles, l'application peut accéder au service, mais seulement si le protocole additionnel est implémenté dans l'application, en utilisant le CLDC Generic Connection Framework.

Comme le protocole OBEX peut être utilisé sur différents médias de transmission (filaire, infra-rouge, radio bluetooth, etc...), JSR 82 implémente l'API OBEX (`javax.obex`) indépendamment du coeur de l'API Bluetooth (`javax.bluetooth`). L'API OBEX est un package optionnel séparé, utilisable seul ou avec l'API Bluetooth.

Le Profile Port Série

Le protocole RFCOMM, qui est situé au-dessus du protocole L2CAP protocol, émule une connexion série RS-232. Le Serial Port Profile (SPP) facilite les communications entre les appareils Bluetooth en fournissant une interface basée sur les flux au protocole RFCOMM. Les caractéristiques sont les suivantes :

- Deux appareils peuvent partager une seule session RFCOMM à un moment donné.
- Jusqu'à 60 connexions séries logiques peuvent être multiplexées sur cette session.
- Un seul périphérique Bluetooth device peut avoir au plus 30 services RFCOMM actifs .
- Un périphérique peut supporter une seule connexion client à n'importe quel service à un instant donné.

Pour qu'un serveur et un client puisse communiquer en utilisant le SPP, chacun doit réaliser quelques actions simples.

Comme le montre le code suivant, le serveur doit :

1. Construire l'URL qui indique comment se connecter au service, et la stocker dans l'enregistrement de service.
2. Rendre l'enregistrement de service disponible au client
3. Accepter une connexion du client.
4. Envoyer et recevoir des données au client.

L'URL placée dans l'enregistrement de service peut ressembler à :

```
btspp://102030405060740A1B1C1D1E100:5
```

La signification est la suivante : le client doit utiliser le Bluetooth Serial Port Profile pour établir une connexion à ce service, qui est identifié avec le port 5 du serveur, sur un appareil dont l'adresse est : 102030405060740A1B1C1D1E100.

```
...
// assuming the service UID has been retrieved
String serviceURL =
    "btspp://localhost:"+serviceUID.toString());
// more explicitly:
String ServiceURL =
    "btspp://localhost:10203040607040A1B1C1DE100;name=SPP
        Server1";
try {
    // create a server connection
    StreamConnectionNotifier notifier =
       (StreamConnectionNotifier) Connector.open(serviceURL);
    // accept client connections
    StreamConnection connection = notifier.acceptAndOpen();
    // prepare to send/receive data
    byte buffer[] = new byte[100];
    String msg = "hello there, client";
    InputStream is = connection.openInputStream();
    OutputStream os = connection.openOutputStream();
    // send data to the client
    os.write(msg.getBytes());
    // read data from client
    is.read(buffer);
    connection.close();
} catch(IOException e) {
  e.printStackTrace();
}
...
```

De l'autre côté, pour que le client établisse une connexion RFCOMM, il doit (cf code suivant) :

1. Lancer une découverte de service pour récupérer l'enregistrement de service
2. Construire une connexioin URL en utilisant l'enregistrement de service.
3. Ouvrir une connexion sur le serveur
4. Envoyer et recevoir des données au serveur.

```
...
// (assuming we have the service record)
// use record to retrieve a connection URL
String url =
    record.getConnectionURL(
        record.NOAUTHENTICATE_NOENCRYPT, false);
// open a connection to the server
StreamConnection connection =
    (StreamConnection) Connector.open(url);
// Send/receive data
try {
    byte buffer[] = new byte[100];
    String msg = "hello there, server";
    InputStream is = connection.openInputStream();
    OutputStream os = connection.openOutputStream();
    // send data to the server
    os.write(msg.getBytes);
    // read data from the server
    is.read(buffer);
    connection.close();
} catch(IOException e) {
  e.printStackTrace();
}
...
```