Technical Report:
Bluetooth integration into CLDC/MIDP

Author:

Thomas Strang,
Institute for Communications and Navigation,
German Aerospace Center (DLR)

(thomas.strang@dlr.de)

Filed:

April 20th, 2001

Introduction

This technical report deals with the integration of Bluetooth into the framework of Java2 Micro Edition (J2ME). It should be thought of as a collection of ideas, focussing the special aspects of using Bluetooth [4] and its different profiles in small, resource limited devices equipped with J2ME, defined by the Connected Limited Device Configuration (CLDC [1]) as base for the Mobile Information Device Profile (MIDP [2]). It should not be thought of as concurrent work to the specification, which will be the result of JSR-82 [7] some time. The ideas presented here could be usefull when testing applications built for the MIDP (MIDlets), using one of the currently available MIDP emulators.

Most of the ideas here are already realized as prototype as extension to the MIDP reference implementation and the Wireless Toolkit [3] provided by Sun and Toshiba(Digianswer) Bluetooth devices [5] within the environment of the Heywow project [8].

The connection framework

With CLDC the generic connection framework has been introduced to support I/O in a generalized and extensible fashion. Eg. whereas the CLDC defines the most generic interfaces like `StreamConnection` and its extension `ContentConnection`, MIDP defines the interface `HttpConnection` as extension of `ContentConnection`. To follow that generic approach, a `BluetoothConnection` should extend one of those basic connection types.

Any connection is opened using the static `Connector` class, which returns an instance of an implementation of one of the general connection interfaces when a connection is established. The protocol in use as well as any connection endpoint information (eg. IP-address and port) are provided as parameters to the `Connector` class call following the URI syntax specified in RFC 2396 where applicable.
In the case of Bluetooth, the scheme of the URI should be "bluetooth:". The following, scheme-specific-part of the URI is currently not defined in a public specification. Therefore, the following shall be a solution for being used as a parameter-string in the generic connection framework:

```
bluetooth:[local_service]//[peer_addr]:<profile>[.remote_service]/[remote_object]
```

Based on that syntax, the following examples could be used to establish a connection between a bluetooth equipped mobile information device (MID) and another bluetooth station:

```
Connector.open("bluetooth://:serial")
Connector.open("bluetooth://00.50.cd.11.0a.4f:serial.com6/index.xml")
```

The first example is the most minimal version of providing the necessary information to establish a connection to any station in the range of the MID using one of the predefined profiles, the serial profile.
The latter one provides also some more information, which may be useful for selecting a specific service ("com6") on a specific station ("00.50.cd.11.0a.4f") as well as for protocols being used on-top of the bluetooth connection like HTTP ("/index.xml").

In the syntax specification above, the `peer_adr` is the 6-byte-hardware address identifying each single bluetooth device. If the hereby identified peer is in the reachable range of the MID's bluetooth device, a connection to that device is established. If it is not reachable, or does not support the mandatory `profile` information, a `ConnectionNotFoundException` must be thrown. If `peer_adr` is omitted, a connection to the first reachable station supporting the required `profile` is established. In both cases, bluetooth's lower layers device discovery is used to find out the reachable devices in the local area. The format of the address ("aa.bb.cc.dd.ee.ff") must be the hex-representation of the remote device's address. The dots may be omitted, but must not be replaced by colons (like "aa:bb:cc:dd:ee:ff") as usual for bluetooth addresses because of the possible additional sources of parsing errors.

The optional `remote_service` indicator, separated from the `profile` by a dot, can be used to connect to a specific service implementing the `profile`, eg. when the remote device offers multiple services for the same profile. Although the syntax above is designed to provide also this type of information for the mobile information device as `local_service`, it is questionable if this optional parameter will be used in limited mobile devices.

The `profile` specifier is mandatory, but is required only once, because any bluetooth link between two devices must use the same profile.

The last part of the supposed syntax, the `remote_object` parameter, is not required for establishing the connection itself. But to be conform with HTTP-like connections, this parameter can be used to specify a path and a filename on the remote station already in the open statement. In other circumstances, eg. when using the bluetooth profile `obex`, this parameter can be used to specify the destination path for the exchanged objects.

Bluetooth Profiles

The approach shown here may be usefull in all scenarios, where the bluetooth connection can be seen as a channel for input and output streams. This is a much less restrictive requirement and should be fulfilled by at least any bluetooth profile, which is the *serial profile* or is on top of the *serial profile*. A list of those profiles is given below. For use in the parameter-syntax described above, the nomenclature in the second column is proposed:

Profile Name	**Profile Identifier**
Serial Port Profile	`serial`
Dial-up Networking Profile	`dun`
Fax Profile	`fax`
Headset Profile	`headset`
LAN Access Profile	`lan`
Generic Object Exchange Profile	`obex`
File Transfer Profile	`obex-ftp`
Object Push Profile	`obex-push`
Synchronization Profile	`obex-sync`

It is optional to implement services for any of the listed profile identifiers, with the exception of the `serial` profile identifier, which is mandatory. When an unsupported profile identifier is used, `an IOException` must be thrown.
Whereas the approach introduced here can be used for any of the mentioned profiles, we will focus on the serial profile in this technical report.

Protocol Handler

The introduction of the bluetooth scheme ("`bluetooth:`") requires the implementation of a class exactly named `com.sun.midp.io.j2me.bluetooth.Protocol.class`. This enables the MIDP implementation to find, load and instantiate the bluetooth protocol handler at runtime when `Connector.open()` is called.

This class is responsible for parsing the scheme-specific part of the URI as well as providing the system with the protocol specific functions. By implementing this class as an extension to the `ConnectionBase` class, conformance to the `Connection` and some other standard connection interfaces is already ensured. For using bluetooth as a provider for stream-oriented I/O connections, it is also necessary to implement functions to open streams as defined in the CLDC standard `InputConnection` and `OutputConnection` interfaces, which both together build the `StreamConnection`.

Bluetooth Connection Interface

The bluetooth specific functions like device discovery etc. are not accessible by the CLDC or MIDP standard connection interfaces. Thus, it is more than consequent to introduce a new `BluetoothConnection` interface, which specifies the access to the bluetooth specific functions in an implementation independent manner.

The following paragraphs deal with some of the functionality required from a `BluetoothConnection` interface, namely *discovery* and *HTTP connections* on top of a bluetooth connection. But keep in mind, the proposed interface – shown at the end of this chapter – is certainly not complete.

Discovery

Service discovery is a very complex procedure within Bluetooth. We will focus here on the mobile information device user's perspective upon that subject, to keep it as simple as possible.

Device discovery is a procedure, where one bluetooth device tries to find other ones to communicate with in its local area. This procedure may be useful in any situation, where a subsequent peering communication partner is currently unknown. Device discovery can be seen as a part of the service discovery defined within bluetooth (in fact, it is defined in some lower layers than SDP).
Each time the device discovery is performed by calling the function `deviceDiscovery()` of the `BluetoothConnection` interface, a list of currently reachable bluetooth devices within the reception area of the local bluetooth device is created. This list can be accessed by stepping through the Enumeration returned by the `getDiscoveredAddresses()` function. Each element of the Enumeration is a String containing the address of a remote device (like "00.50.cd.11.0a.4f") found during device discovery.

Extensive information about the capabilities of each specific device (like the friendly name or the supported profiles and services) can be accessed by the content of the `BluetoothDeviceParameter` class, returned by the function `getDeviceParameter()`.

HTTP over Bluetooth

One innovation of MIDP on top of CLDC was the introduction of the `HttpConnection` interface. Through a class implementing that interface, a MIDlet is able to exchange data according to HTTP v1.1 [6]. The MIDP `HttpConnection` interface provides the additional functionality needed to deal with HTTP request/response headers as well as other HTTP specific handling, and extends the CLDC `ContentConnection` interface, which is an extension of the CLDC `StreamConnection` interface itself.

Looking for a software architecture design which incorporates both – Bluetooth and HTTP support into CLDC/MIDP offers several different approaches, beside others:

a. Hide anything bluetooth specific "under the hood" of the `HttpConnection` interface. That means extending the classes implementing the `HttpConnection` interface (`com.sun.midp.io.j2me.http.Protocol.class` in this case) by functionality

to handle syntax extensions of the URL containing all the necessary information (profile, services, discovery etc.) for a bluetooth connection.

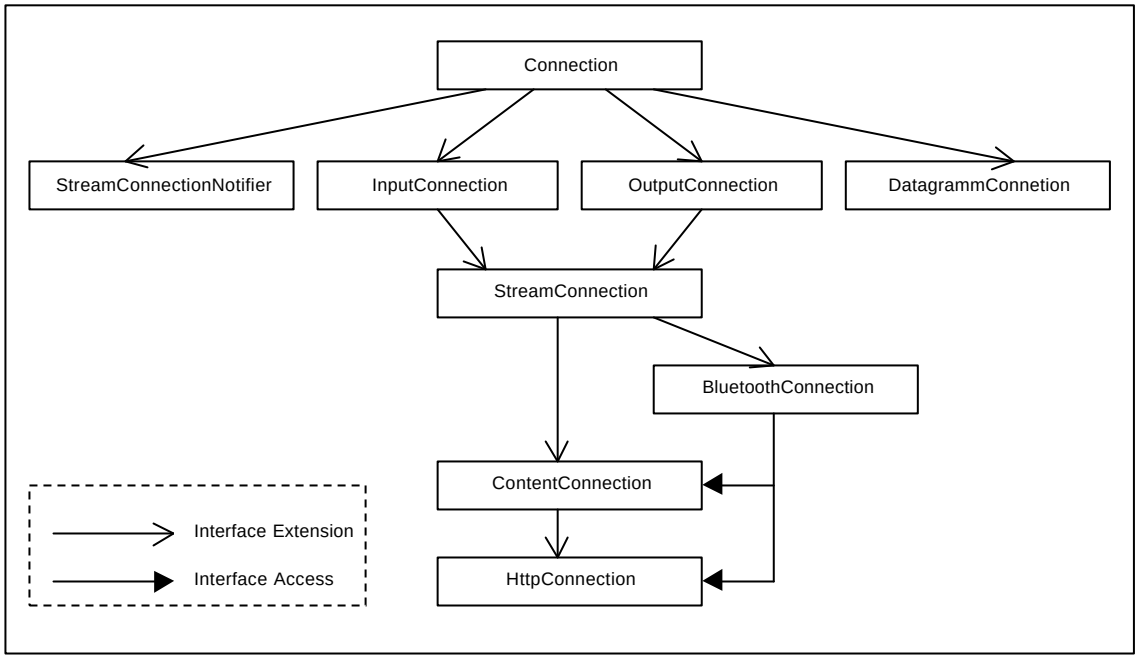
b. Implement the bluetooth specific parts in its own protocol class, which is handled by the execution environment exactly like other protocols in the generic connection framework. That means identifying and loading the protocol handling class at runtime out of the URL used in the `Connector.open()` statement.
Implement the `BluetoothConnection` interface as an extension to the `HttpConnection` interface.

c. Like (b), but implement the `BluetoothConnection` interface as an extension to the Stream`Connection` interface to provide a stream oriented connection behaviour. Enable the use of HTTP by implementing the `HttpConnection` interface beside a bluetooth connection.

d. Like (c), but enable the use of HTTP by providing a function called `openHttpConnection()` to access an instance of a `HttpConnection` on top of the bluetooth connection.

We decided for the latter one for several reasons. It offers a strict separation between the bluetooth connection in the sense of a “transport layer” and the HTTP connection as a “context layer”. This allows to use a bluetooth connection without any relationship to HTTP, eg. when used with the headset profile.

The following `Connection` interface hierarchy illustrates the relationsships:



The `BluetoothConnection` interface can be seen as a separate extension of the CLDC and is therefore usefull in CLDC, but non-MIDP environments (eg. if the PDA profile is used on top of CLDC).

The syntax of the URL passed to `openHttpConnection()` matches the format specified in RFC 2396 with the easement, that the `scheme`-specifier may be omitted (always "`http:`"), as well as the `net_path`, if the accessed resource is a local one at the already connected server.

If no URL is passed to the function or the URL passed is an empty string, the optional `remote_object` part of the the URL passed to the `Connector.open()` function may be used instead.

Proposed interface

```
/**
 * This interface defines some capabilities of a bluetooth connection.
 *
 * @author  Thomas Strang, DLR Oberpfaffenhofen
 * @version 20.04.2001
 */
public interface BluetoothConnection extends StreamConnection
{
    /**
     * Set a timeout value for the duration of following device
     * discoveries. Default is 10 seconds.
     *
     * @param                The new value in seconds
     * @return               The previous value in seconds
     * @exception IOException If an I/O error occurs
     */
    public int setDiscoveryTimeout(int seconds) throws IOException;

    /**
     * Perform a device discovery for the time specified previously set.
     *
     * @return                Number of devices found excl. the local one
     * @exception IOException  If an I/O error occurs
     */
    public int deviceDiscovery() throws IOException;

    /**
     * Get an Enumeration of the discovered devices by their address
     *
     * @return                Enumeration of addresses. Each element is
     *                        a String of the format "aa.bb.cc.dd.ee.ff".
     * @exception IOException  If an I/O error occurs
     */
    public Enumeration getDiscoveredAddresses() throws IOException;

    /**
     * Collect information about a specific device
     *
     * @param     Address of device to request informations about,
     *            specified as a String of the format "aa.bb.cc.dd.ee.ff",
     *            "aa:bb:cc:dd:ee:ff" or "aabbccddeeff". An empty String
     *            ("") can be used to determine the local device.
     * @return                Instance of class
     *                        <code>BluetoothDeviceParameter</code>
     * @exception IOException  If an I/O error occurs
     */
    public BluetoothDeviceParameter getDeviceParameter(String address)
    throws IOException;
```

```
    /**
     * Create a HTTP-over-Bluetooth connection. At the same time
     * give also access to a ContentConnection.
     *
     * @param                  String containing a URL.
     * @return                 Instance of a class implementing the
     *                         <code>HttpConnection</code> interface
     *                         and all superinterfaces.
     * @exception IOException  If an I/O error occurs
     */
    public HttpConnection openHttpConnection(String url)
    throws IOException;
}
```

Sample

Assume a mobile device equipped with CLDC/MIDP, Bluetooth and a library implementing the functionality introduced here. A MIDlet may be downloaded, installed and started by the Java Application Management (JAM). When running, the MIDlet may do the following steps in a loop:

```
while (true)
{
  BluetoothConnection btconn;
  try {
    btconn = (BluetoothConnection)
      Connector.open("bluetooth://:serial/");
  }
  catch (NoSuchElementException e) {} // no peer found, try again
  catch (IOException e) {} // error, sleep some time, try again

  HttpConnection htconn;
  try {
    htconn = btconn.openHttpConnection("/heywow/peerinfo.xml");
  }
  catch (IOException e) {} // error, try from beginning

  InputStream is;
  try {
    is = htconn.openInputStream();
    for (int i=0; i < htconn.getLength(); i++)
      System.out.print( is.read() );
  }
  catch (IOException e) {} // error, try from beginning

  try {
    is.close();
    htconn.close();
    btconn.close();
  }
  catch (IOException e) {} // error, try from beginning
}
```

In this example a MIDlet tries to establish a bluetooth connection by using the Serial Port Profile and any available own matching service to some other device in its local area offering

a remote service for the Serial Port Profile. When the bluetooth connection is successfully established, the HTTP server bound to the remote service is contacted and requested to deliver a specific file (“/heywow/peerinfo.xml” in this case). After transferring the file it may be processed somehow (printed in this case), the connections are closed, and after a short time the loop may be entered again. This sample shows how easy it can be to collect information with a mobile information device via bluetooth while walking along some bluetooth equipped service points.

References

[1] CLDC – http://java.sun.com/products/cldc
[2] MIDP – http://java.sun.com/products/midp
[3] Wireless Toolkit – http://java.sun.com/products/j2mewtoolkit
[4] Bluetooth – http://www.bluetooth.com
[5] Digianswer – http://btsws.digianswer.com/btsws
[6] HTTP – http://sunsite.dk/RFC/rfc/rfc2616.html
[7] JSR-82 – http://java.sun.com/aboutJava/communityprocess/jsr/jsr_082_bluetooth.html
[8] Heywow – http://www.heywow.com