PUNTO DE ENCUENTRO VIRTUAL P2P CON ACCESO MÓVIL

ANEXO IV – "PROGRAMACIÓN CON JXME"

RICARDO ALBERTO CAMACHO GÓMEZ

LUIS ERNESTO GARCÍA MARTÍNEZ

UNIVERSIDAD DEL CAUCA

FACULTAD DE INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES

DEPARTAMENTO DE TELEMÁTICA

LÍNEA DE ÉNFASIS EN INGENIERÍA DE SISTEMAS TELEMÁTICOS

POPAYÁN

TABLA DE CONTENIDO

ANEXO IV. PROGRAMACIÓN CON JXME

A través de este capítulo se presentan los fundamentos para empezar a programar con JXME, clases y métodos disponibles para posteriormente aplicar los elementos vistos en la construcción de dos MIDlets de prueba que ejemplifican lo visto hasta el momento.

1.1. LAS CLASES FUNDAMENTALES EN LA PROGRAMACIÓN JXME

Las clases base para la implementación de soluciones JXTA en dispositivos J2ME son:

- *Element*
- *Message*
- *PeerNetwork*

A continuación se estudiarán en detalle cada una de estas clases.

1.1.1. La clase *Element*

Los mensajes JXME se componen de partes denominadas *Elements* (elementos). La implementación JXME usa la clase *Element* para crear elementos o componentes del mensaje de acuerdo a la aplicación desarrollada.

Cuando el desarrollador desee crear sus propios elementos de mensaje, la declaración para esto se hace de la siguiente forma:

```
Element  e = new Element("message","HelloWorld!".getBytes(),
                              "peerNamespace",null);
```

El primer parámetro (“message”) especifica el nombre del elemento. El segundo parámetro, (“Hola mundo!”.getBytes()) contiene el arreglo de bytes que representa el contenido del elemento. El tercer parámetro (“peerNamespace”) contiene identificador del namespace (cuando este valor es nulo indica que se trata de el namespace por defecto o el namespace vacío). El último parámetro

(null) especifica el MIME type del dato (null indica que se trata del MIME type por defecto o *application/octet-stream*).

La clase *Element* provee 4 métodos para extraer información acerca del elemento:

1. El método *getName(),* retorna el nombre del elemento.
2. El método *getData(),* retorna los contenidos del elemento.
3. El método *get*Namespace*()* retorna el namespace del elemento (aquí puede ir cualquier valor, o también vacío es aceptable).
4. El método *getMimeType()* retorna el MIME type del elemento.

1.1.2. La clase *Message*

La clase *Message* representa un mensaje JXME completo, para crear un mensaje JXME primero se crean los elementos que constituirán el mensaje. Luego se crea un arreglo de elementos y se le pasa al constructor de la clase Message, quien se encargará finalmente de crear el Mensaje. La declaración de la creación de un mensaje es como la siguiente:

```
Message message = new Message(elementsArray);
```

La clase *Message* contiene tres métodos para la obtención de sus parámetros y elementos que lo componen:

1. El método *getElementCount()* retorna un entero que representa el número de elementos que contiene el mensaje.
2. El método *getElement()* toma como argumento un valor entero que indica el índice del elemento dentro del mensaje.
3. El método *getSize()* retorna un entero que representa el tamaño del mensaje en bytes.

1.1.3. La clase *PeerNetwork:*

La clase PeerNetwork contiene los métodos que permiten la comunicación con la red JXTA mediante el servidor relay. La clase PeerNetwork es como el módulo de comunicación que internamente usa diferentes mensajes y elementos para manejar todos los asuntos de comunicación con el servidor relay.

Esta clase gestiona toda la parte de mensajería y otras tareas de soporte, tales como mantener la identidad de varios mensajes intercambiados entre el cliente y el relay.

En términos generales, la clase *PeerNetwork* es la clase más importante en la implementación del lado del cliente para la integración de los terminales J2ME con la red JXTA.

A continuación se especificarán los métodos más importantes de esta clase:

1.1.3.1. El método *createInstance()*

Mediante este método se puede instanciar la clase PeerNetwork a través de cualquiera de sus dos versiones, una que recibe un parámetro u otra que recibe dos ya que los constructores de la clase PeerNetwork son privados y no pueden usarse para instanciación.

> Uno de los dos *createInstance()* toma como parámetro una sencilla cadena, la cual especifica el nombre que se le quiere dar al peer JXME y luego se llama al constructor de la clase PeerNetwork el cual añade el nuevo peer al NetPeerGroup o grupo por defecto. La otra versión del método *createInstance()* toma dos parámetros, el nombre del peer (*peerName),* y el nombre del grupo al cual se desea unir (*peerGroupName*).
>
> Al final de cuentas el que recibe dos parámetros como argumento (peerName y peerGroupName) hace que el nuevo peer resulte unido al

peer group especificado, en tanto que el que recibe solo un parámetro crea el peer y lo ubica en el peer group por defecto o *NetPeerGroup.*

Luego, el constructor internamente reduce el tamaño del identificador del peer group (proceso llamado *trimming*) esto ayuda a optimizar el transporte de tráfico saliente hacia la red inalámbrica.

Finalmente el método *createInstance()* instancia un objeto de la clase HttpMessenger, una clase usada para la comunicación con el relay y que provee a los peers JXME una capa de abstracción entre las clases de alto nivel JXME (tales como PeerNetwork) y las clases J2ME/MIDP de nivel más bajo.

Por ejemplo, JXME viene con dos versiones de la clase HttpMessenger, una para dispositivos CLDC y otra para dispositivos CDC. Esto permite ver que los diseñadores de JXME han construido clases de alto nivel para trabajar sobre diferentes versiones de J2ME.

1.1.3.2. El método *connect()*

Después de instanciar un objeto de la clase *PeerNetwork,* se realiza una llamada a su método *connect()* , como se muestra a continuación:

```
//PeerGroupDemo.java
byte[] persistentState =
peerNetwork.connect("http://172.16.0.37:9700",null);
```

El primer parámetro es una cadena que contiene la URL del relay. Esta es la dirección IP y el número de puerto por el que el host relay sirve a sus clientes. Estos datos deben ser conocidos

El Segundo argumento es el identificador del peer ó como de ahora en adelante en los segmentos de código se conocerá como *peerId*. Como el *peerId* aún no se conoce, el cliente debe pedirle al relay que le asigne

uno. Esta es la razón por la cual se le pasa *null* como segundo parámetro al método *connect(*.

El método connect() lleva a cabo los siguientes pasos:

- Internamente llama al método *connect()* de la clase HttpMessenger y le pasa ambos argumentos.
- El método connect() de HttpMessenger construye la URL basada en la información que se le acaba de pasar y arma una petición de peerId y la envía al relay.
- Si el relay retorna con éxito un identificador de peer (*peerId)*, el método *HttpMessenger.connect()* envía una petición de establecimiento de conexión al relay.
- Después de tener establecida una conexión con el relay, el método *PeerNetwork.connect()* retorna el peerId en un arreglo de bytes. Este *peerId* representa al cliente JXME en esta sesión de comunicación. El cliente JXME usa este identificador (peerId) a través del resto de su comunicación con el relay. Por tanto, es necesario guardar este identificador para su uso posterior.

  Si se pasa un peerId como segundo parámetro al método *PeerNetwork.connect(),* el funcionamiento del método cambia ligeramente. Ahora el método no necesita pedir un peerId, en este caso ya se salta este paso y procede directamente a establecer la conexión.

  Nótese que cuando se pasa *null* como segundo parámetro al método *PeerNetwork.connect()* lo que se hace es que se asigna un nuevo peerId. La próxima vez se debe usar este peerId para conectarse al relay. Por qué?, pues porque si se continúa pasando *null* como segundo parámetro se le asignarán nuevos peerId cada vez. Es como si en el sistema de correo electrónico tradicional se estuviera cambiando al usuario de dirección cada vez que se quieran revisar los

mensajes de entrada, por lo que no se podrían leer los mensajes entrantes.

1.1.3.3. El método *create()*

Este método es utilizado para crear un recurso JXTA (por ejemplo, un pipe o un peer group). Este toma cuatro parámetros:

- El tipo de recurso a crear. Si se quiere crear una pipe, se pasa *PeerNetwork.PIPE* como primer parámetro. Igualmente, si se desea crear un peer group, se pasa *PeerNetwork.GROUP.*
- El nombre que tendrá el recurso. Por ejemplo, si se quiere crear una pipe con el nombre *miPipe,* se pasa esta cadena como segundo parámetro.
- Un identificador predeterminado para el recurso. Se puede pasar *null* como tercer parámetro si que desea que el relay sea quien provea el identificador.
- Un argumento adicional especificando el tipo de pipe a crear. Por ejemplo, si se quiere crear una pipe punto a punto, se pasa *Peernetwork.UNICAST_PIPE* como cuarto argumento, o *Peernetwork.PROPAGATE_PIPE* si es una pipe de difusión, sin embargo funcionalmente esta distinción se hizo hasta hace poco, ya que se puede crear una pipe de difusión (propagate) a partir de varias pipes unicast, por lo que a nivel de programación generalmente se emplea la unicast. Nótese que se pasa *null* como cuarto parámetro si se desea crear un peer group.

El método *create()* resulta en la creación de un mensaje. Se le pasa el mensaje a un ayudante privado, el método *sendMessage(),* quien adiciona el mensaje a la cola de mensajes salientes.

El método retorna un entero, el cual la aplicación puede usar para confrontar las respuestas recibidas y escoger las que le corresponden.

Este mecanismo es necesario, ya que las respuestas del relay pueden llegar en orden aleatorio.

1.1.3.4. El método *join()*

Básicamente este método permite a un peer unirse a un peer group. Para su operación toma dos parámetros, el identificador del peer group al que el peer desea unirse, y el password de autenticación. La implementación actual de JXME no soporta el uso de passwords, los ignora en caso de recibirlos, solo aparece sin funcionalidad como parte de una implementación en constante evolución.

Este método simplemente crea un mensaje de petición de ingreso (*join*) y lo pone en la cola de mensajes salientes. La cola de mensajes contiene todos los mensajes que se hayan enviado al relay, por ejemplo, la petición de ingreso a un grupo, la petición de creación de pipes, o una petición de búsqueda. Los mensajes que se encuentran en cola son enviados al relay cuando se invoca el método *poll().*Este método se describirá con más detalle a continuación.

1.1.3.5. El método *poll()*

Este método permite sondear el relay para obtener los mensajes entrantes, se declara de la siguiente forma:

```
Message message = peerNetwork.poll (2000);
```

El método *poll()* necesita solo un parámetro, un valor de *timeout* o de tiempo de expiración. Este valor se especifica en milisegundos e indica el tiempo entre peticiones de sondeo al relay. Un valor de *0* indica que se queda esperando para siempre un mensaje y nunca va a sondear.

Cuando un cliente JXME sondea el relay, ejecutando el método *PeerNetwork.poll()* primero examina su cola de mensajes no enviados, si

el método encuentra algún mensaje en la cola lo ubica en el cuerpo de la petición HTTP y lo envía al relay usando el método POST.

Desde luego, se espera a que el relay le envíe al peer JXME algún mensaje entrante en respuesta al sondeo que realiza. El relay podría no tener mensajes entrantes para el cliente, en ese caso retornaría *null.* En caso de que el relay tenga un mensaje para el cliente, el método *poll()* internamente adapta la respuesta, crea un objeto de la clase *Message* propio del mensaje entrante y retorna el *Message* a la aplicación que realizó la solicitud. Desde luego, la aplicación que mandó a sondear el relay debe llamar algunos de los métodos de la clase *Message* para extraer la información de este. Más adelante se dará un ejemplo de esto que se acaba de explicar.

1.1.3.6. El método *send()*

Este método envía los mensajes de aplicación a determinadas pipes. El método requiere dos parámetros, el mensaje a ser enviado y la pipe destino por la cual debe enviarse. Este método internamente pasa estos parámetros a un método privado de la clase PeerNetwork llamado *pipeOperation(),* el cual ofrece funcionalidades comunes a todos los métodos basados en pipes tal como *listen()* y *send().* El método *pipeOperation()* toma el mensaje y le agrega algunos elementos, tales como identificador de la pipe destino (pipeId), el identificador de petición (requestId), y los elementos de la dirección del destino listados en la figura 3.20, del capítulo 3.

El método *send()* finalmente ubica el mensaje en la cola de envío para despacharlo al relay.

1.1.3.7. El método *search():*

Una aplicación J2ME usa el método *PeerNetwork.search()* para buscar diferentes recursos JXTA, tales como peer groups y pipes. El método *search()* toma cuatro parámetros:

- El tipo de recurso que se va a buscar. Por ejemplo, *PeerNetwork.GROUP* si se quiere buscar un peer group o *PeerNetwork.PIPE* si se quiere buscar una pipe.
- El nombre del atributo del recurso a buscar, es decir el parámetro de búsqueda, por ejemplo, si se quiere buscar algún recurso por su nombre, este campo lleva entonces "Name".
- La cadena a buscar, por ejemplo, "miPipeHola" iría en este campo si se quisiera buscar una pipe llamada así.
- Un valor de límite, conocido como *threshold,* especificando el máximo número de respuestas que se pueden recibir desde un peer remoto.

El método *search()* crea la petición de búsqueda de acuerdo a los parámetros recibidos y finalmente le pasa dicha petición al método *sendMessage()* quien ubica el mensaje en la cola para su posterior envío.

1.1.3.8. El método *listen()*

Este método se utiliza para abrir una pipe de entrada o de lectura. En su invocación toma un parámetro, el identificador de pipe ó más conocido como *pipeId.* Una vez se ha invocado este método, el peer queda listo para escuchar mensajes entrantes por la pipe especificada.

1.2. EJEMPLOS DE PROGRAMACIÓN CON JXME[1]

Hasta ahora se han visto las clases más importantes en la implementación actual de JXME. A continuación se presentan algunos ejemplos de programación en los que se demuestra el uso de estas clases desarrollando algunas de las tareas más comunes en las aplicaciones JXME. En seguida se describirán las siguientes tareas:

- Cómo conectarse al relay
- Cómo crear un peer group
- Cómo buscar un peer group
- Cómo procesar mensajes JXME
- Cómo enviar y recibir mensajes a través de pipes

Esto ayudará a entender más a fondo el funcionamiento del punto de encuentro y de cualquier otra aplicación desarrollada con JXME.

Se desarrollarán dos MIDlets de ejemplo, (PeerGroupDemo y PipeDemo) para estudiar más a fondo el desarrollo del lado del cliente JXME. Estas MIDlets utilizan todas las clases descritas hasta el momento, la aplicación PeerGroupDemo muestra todas las tareas relacionadas con los peer groups y el PipeDemo muestra cómo crear, buscar, y usar pipes. En esta sección solo se muestran los segmentos de código que interesan, el código fuente de la aplicación completa se encuentra en la parte final de este anexo.

1.2.1. Cómo conectarse al relay

Ambas MIDlets necesitan conectarse al relay antes de realizar cualquier cosa. Por lo tanto en ambas MIDlets esta parte del código es igual. A continuación se muestra el código que permite a un cliente J2ME ingresar a la red JXTA a través del relay.

```
// PeerGroupDemoMIDlet.java
// peerNetwork es un objeto de la clase PeerNetwork.
try {
```

[1] Khan Faheem; "Wireless messaging with JXTA, Part 1:Using JXTA Technology"; http://www-106.ibm.com/developerworks/wireless/library/wi-jxta/#9

```
  peerNetwork = PeerNetwork.createInstance ("JxmeTestPeer");
 byte[] persistentState =
peerNetwork.connect("http://localhost:9700", null); }
}
catch (IOException ioEx) {
ioEx.printStackTrace ();
}
```

En este código se muestran los siguientes pasos:

1. Se inicia la plataforma JXTA para el cliente JXME mediante la instanciación de la clase PeerNetwork. Para esto, se usa el método *PeerNetwork.createInstance()* y solo se le especifica el nombre del peer.
2. Se conecta a la red JXTA mediante el relay, ejecutando el método *PeerNetwork.connect()* y especificando la URL del host relay.

Como resultado a una petición de búsqueda exitosa, el peer JXME se une al NetPeerGroup (el grupo por defecto al cual pertenecen todos los peers en la red JXTA), si después de esto el peer quiere unirse a otro grupo debe primero buscarlo y luego unirse a él, más adelante se verá con más detalle este proceso.

1.2.2. Trabajo con Grupos

La MIDlet DemoPeerGroup demuestra el uso de peer groups. La MIDlet contiene tres métodos: createPeerGroup, searchPeerGroup y joinPeerGroup, métodos en los cuales se demuestra cómo crear, buscar y unirse a peer groups, respectivamente. Otro método llamado *processMessage()* demuestra cómo procesar un mensaje entrante JXME y extraer información útil de él.

El constructor de DemoPeerGroup llama estos métodos en una secuencia determinada solo para sacar una demostración. Primero, crea un grupo, luego busca dicho grupo y finalmente se une al mismo.

1.2.2.1. Cómo crear un grupo

El siguiente fragmento de código fuente contiene un método llamado *createPeerGroup(),* el cual demuestra los pasos a seguir para crear un grupo. Este método se encuentra también en la MIDlet *PeerGroupDemo.*

```
//PeerGroupDemo.java
public String createPeerGroup(String groupName)
{
    String newGroupId = null;
    try {
        /***Paso 1***/
        int messageID = peerNetwork.create (
                                PeerNetwork.GROUP,
                                groupName,
                                null,
                                null);
        /***Paso 2***/
        Message msg = peerNetwork.poll (5000);

        /***Paso 3***/
        if (msg == null)
            return null;

        /***Paso 4***/
    newGroupId = processMessage(msg, messageID, "success");
    }
    catch (IOException ie) {

        ie.printStackTrace ();
    }
    return newGroupId;
}
```

El método toma como parámetro una cadena especificando el nombre del nuevo grupo, y se hace necesario llamar al método *PeerNetwork.create()*. Se pasa una constante denominada PeerNetwork.GROUP como primer parámetro al método *create()* y como segundo parámetro el nombre del peer group. Se puede pasar el peerGroupId como tercer parámetro, en caso de conocerlo, sino se pasa *null* como tercer parámetro y dejar que el relay decida el valor del peerGroupId. En cuanto al cuarto parámetro, siempre se debe pasar *null* cuando se estén creando grupos, ya que este parámetro no es necesario para la creación de grupos.

Se debe guardar el identificador de petición (*requestId*) retornado por el método *create()* para llevar el control de la futura respuesta a esta petición, la cual llegará con un *requestId* aumentado en uno.

El siguiente paso es llamar al método *poll()* del objeto PeerNetwork. El método *poll()* envía la petición de creación de peer group desde la cola de mensajes al relay y también mira si hay mensajes pendientes para el cliente. En caso de que el relay tenga algún mensaje para el cliente, el método *poll()* retorna un objeto de la clase *Message,* de otra forma retorna *null.*

Si el mensaje no es *null,* es necesario verificar si el mensaje está en la respuesta a la petición de creación de grupo o en la respuesta de algún otro mensaje. Para esto se implementó el método *processMessage().*

1.2.2.2. Cómo procesar mensajes JXME

El código presentado a continuación presenta el método *processMessage(),* el cual muestra la lógica de procesamiento para un mensaje JXME y el uso de varios métodos de las clases *Message* y *Element.*

El método *processMessage()* toma tres parámetros: un objeto de la clase Message llamado msg, un identificador llamado reqId, y una cadena llamada *successCriterion (parámetro de búsqueda*).

El método *processMessage()* verifica si el objeto Message es la respuesta a la petición cuyo identificador se pasó como segundo parámetro. Si es así, el método

processMessage() explora el objeto *Message* más a fondo para determinar si la petición (por ejemplo, una petición de búsqueda de peer group) fue realizada exitosamente.

La confirmación de una respuesta exitosa varía de acuerdo a los diferentes tipos de petición. Este es el por qué del paso del tercer parámetro llamado *successCriterion.* El método *processMessage()* verifica si la cadena *successCriterion* concuerda con el contenido de los elementos del mensaje *response.* Si concuerda, el método *processMessage()* extrae los contenidos de un elemento llamado *id,* y retorna los contenidos. El elemento *id* contiene el identificador del recurso buscado (por ejemplo, un *peerGroupId*).

El método *processMessage()* procesa las respuestas de las peticiones crear (*create*), buscar (*search*), unirse a grupo (*join*), escuchar (*listen*) y enviar (*send*). A estas peticiones las respuestas no retornan ningún identificador. En este caso, el método *processMessage()* retorna los contenidos del elemento *response* en vez de los del elemento *id.*

```
//PeerGroupDemo.java
private String processMessage (Message msg, int reqId,
String successCriterion)
{
    int requestIdentifierInResponse = 0;
    String searchedResourceIdentifier = null;
    String responseString = null;

    try
    {
        /***Paso 1***/
        for (int j=0; j<msg.getElementCount(); j++) {
            /***Paso 2***/
            Element  e = msg.getElement(j);
            /***Paso 3***/
            if ((e.getNameSpace()).equals("proxy")) {
```

```
                if
((e.getName()).equals("requestId"))requestIdentifierInRespon
se = Integer.parseInt(new String(e.getData()));
                else if ((e.getName()).equals("response"))
                    responseString     =     new     String
(e.getData());
                else if ((e.getName()).equals("id"))
                    searchedResourceIdentifier = new String
(e.getData());
            }
        } //for (int j=0;)

        /***Paso 4***/
        if (requestIdentifierInResponse == reqId) {
            /***Paso 5***/
            if (responseString.equals (successCriterion)) {
                if (searchedResourceIdentifier != null)
                    /***Paso 6***/
                    return searchedResourceIdentifier;
                else
                    /***Paso 7***/
                    return responseString;
            }
        }//if (requestIdentifierInResponse == reqId)

    }
    catch(NumberFormatException ne){
        ne.printStackTrace();
    }
    return null;
}
```

El código anterior muestra los siguientes pasos:

1. Primero, llama al método *getElementCount()* del objeto *Message,* el cual retorna el número de elementos del mensaje.
2. Ahora se recorren todos los elementos del mensaje ejecutando el método *getElement()* al mensaje entrante obteniendo así el elemento (*Element*) correspondiente al índice.
3. Luego, se procesa el elemento obtenido a través de varias sentencias *if.* En el código de arriba, se han utilizado los métodos *getName(), get*Namespace*() y getData()* para extraer los datos de tres elementos de un mensaje en particular (*requestId*, *response* y *Id*). Todos los tres elementos pertenecen al namespace *proxy.* Se almacena el contenido del elemento *requestId* en una variable llamada *requestIdentifierinResponse*, el del elemento *response* en otra variable llamada *responseString* y el del elemento *id* en *searchedResourceIdentifier.* La variable *requestIdentifierInResponse,* la variable *requestidentifierInResponse* ahora contiene el identificador de la petición correspondiente al mensaje de respuesta *msg.* La variable *searchedResourceIdentifier* contiene el identificador del recurso JXTA (por ejemplo, un peer group o pipe) que alguien retornó en respuesta a la petición de búsqueda. La variable *responseString* indica si la petición fue aceptada exitosamente o si hubo algún fallo.
4. El elemento *requestId* especifica la petición a la cual este *Message* está respondiendo. Por lo tanto, hay que confrontar el valor de la variable *requestidentifierInResponse* con el parámetro *reqId* . Si estos no concuerdan, no es necesario procesar más este mensaje.
5. Si el objeto *msg* de la clase *Message* es realmente la respuesta a la petición identificada con el *reqId*, se confrontan los criterios de éxito con los contenidos del elemento *response*.
6. Si el criterio de búsqueda concuerda, sencillamente se retornan los contenidos del elemento *id.* Este elemento siempre contiene el identificador del recurso buscado en todas las respuestas a la petición de búsqueda.

7. Finalmente, si el criterio de búsqueda concuerda pero el *id* del elemento no está presente (lo cual indica que la variable *searchedResourceIdentifier* es *null*), se retorna el contenido de los elementos de la respuesta. Este paso garantiza que se puede usar el método *processResponse()* para procesar las respuestas a las peticiones *join, listen, y send* que no contienen el elemento *id.*

1.2.2.3. Cómo buscar un peer group

Ahora se mostrará la forma de encontrar un peer group específico. El código mostrado a continuación posee el método *searchPeerGroup(),* el cual se ha escrito para demostrar cómo es que se buscan grupos.

```
//PeerGroupDemo.java
public String searchPeerGroup(String groupName){
    String newGroupId = null;
    try
    {
        /***Paso 1***/
        int messageID = peerNetwork.search (
                                    PeerNetwork.GROUP,
                                    "Name",
                                    groupName,
                                    1);
        /***Paso 2***/
        Message msg = peerNetwork.poll (5000);

        /***Paso 3***/
        if (msg == null)
            return null;

        /***Paso 4***/
   newGroupId = processMessage(msg, messageID, "result");
    }
    catch (IOException ie) {
```

```
            ie.printStackTrace ();
        }
        return newGroupId;
}
```

El método *searchPeerGroup()* toma solo un parámetro, el nombre del peer group que se desea encontrar. El método ejecuta los siguientes pasos:

1. Primero, llama al método *search()* de la clase PeerNetwork, a este método se le pasan cuatro parámetros, como resultado de la llamada a este método se crea un mensaje de búsqueda y se pone en la cola de espera de mensajes salientes dentro del objeto de la clase PeerNetwork. El método *search()* retorna un identificador para este mensaje de búsqueda. En el código listado arriba se almacenó el identificador en una variable entera llamada *messageID.*
2. Ahora se llama al método *poll()* de la clase PeerNetwork, el cual retorna un objeto de la clase *Message.*
3. Luego, se verifica si el objeto retornado por el método *poll()* es nulo (*null*), en caso de ser diferente de *null* es porque se ha recibido un mensaje desde el relay.
4. Por último se llama al método *processMessage()* para determinar si el mensaje es la respuesta al mensaje de búsqueda que se envió. Si lo es, el método *processMessage()* retorna el identificador del peer group (*peerGroupId*) que se estaba buscando.

1.2.2.4. Cómo unirse a un Peer Group

El código mostrado a continuación muestra un método sencillo llamado *joinPeerGroup* para demostrar cómo un peer se une a un peer group. Este método toma tan solo un parámetro, el identificador del peer group al cual se quiere ingresar.

```
//PeerGroupDemo.java
public String joinPeerGroup(String groupId)
```

```
{
    String joinOk = null;
    try
    {
        /*** Paso 1***/
        int messageID = peerNetwork.join (groupId, null);

        /*** Paso 2***/
        Message msg = peerNetwork.poll (5000);

        /*** Paso 3***/
        if (msg == null)
            return null;

        /***Paso 4***/
        joinOk = processMessage(msg, messageID, "success");
    }
    catch (IOException ie) {
        ie.printStackTrace ();
    }
    return joinOk;
}
```

El código anterior muestra que para unirse a un peer group, simplemente se requiere llamar al método *join()* de la clase *PeerNetwork* pasándole como parámetro el peerGroupId. Después de llamar a este método, se debe hacer un sondeo al relay y posteriormente llamar al método *processResponse()* para verificar si se ha unido al peer group.

Después de que el peer se ha unido al grupo, se debe instanciar otro objeto de la clase PeerNetwork usando el constructor de dos argumentos. Cabe anotar que es muy importante tener en cuenta la necesidad de crear un nuevo objeto de la clase PeerNetwork por cada grupo al cual ingrese el peer. De esta forma, la lógica de la comunicación dentro de la aplicación desarrollada es mucho más simple y fácil de entender.

1.2.3. Trabajo con Pipes

En esta sección se muestra cómo trabajan las pipes JXTA vistas desde la perspectiva de los terminales J2ME. Como ya se sabe, las pipes se usan para la comunicación P2P, la cual involucra el envío y recepción de mensajes sobre pipes. Para dar un mejor entendimiento de la temática se ha escrito una aplicación sencilla llamada *PipeDemo* para así poder visualizar a nivel de programación la comunicación P2P con las pipes JXTA.

Como se mencionó anteriormente, el código completo de la aplicación se encuentra en la parte final de los anexos. A continuación se analizará el constructor de la aplicación PipeDemo:

```
//PipeDemo.java
public PipeDemo(){
    try {
     peerNetwork=PeerNetwork.createInstance("JxmeTestPeer");
       byte[] persistentState = peerNetwork.connect(
     "http://localhost:9700",null);
        String pipeId = createPipe("JxmeTestPipe");
        String listenOk = listenToPipe(pipeId);
        String searchedPipeId = searchPipe("JxmeTestPipe");
        Message msg = authorMessage("Hola mundo!");
        String sendOk = sendMessage (msg, searchedPipeId);
    }
    catch (IOException ie) {
        ie.printStackTrace ();
    }
}
```

Como se puede ver, el constructor de PipeDemo primero instancia el objeto de la clase PeerNetwork y luego llama a su método *connect(),* estos puntos ya han sido discutidos anteriormente.

El constructor luego llama cinco métodos: *createPipe(), listenToPipe(), searchPipe(), authorMessage() y sendMessage().* Se han escrito estos cinco métodos para facilitar el entendimiento de cómo se realizan las diferentes tareas:

- Cómo crear una pipe
- Cómo abrir una pipe de lectura
- Cómo buscar una pipe
- Cómo crear un mensaje
- Cómo enviar un mensaje a través de una pipe

El resultado de la ejecución de esta secuencia de métodos es que se crea una pipe propia y se empieza a escuchar a través de esta, luego se realiza un búsqueda de la pipe creada para luego crear un mensaje y enviarlo por allí, de tal manera que se recibe el mensaje que se acaba de enviar.

Con esto se tiene ya un repaso general de los conceptos clave en el manejo de pipes, suficiente para empezar a desarrollar la lógica de una aplicación propia.

1.2.3.1. Cómo crear una pipe

El código fuente mostrado a continuación muestra la implementación del método *createPipe()* y demuestra cómo se crea una pipe.

```
//PipeDemo.java
public String createPipe(String pipeName){
    String pipeId = null;
    try{
        /*** Paso 1***/
        int messageID = peerNetwork.create (
                                PeerNetwork.PIPE,
                                pipeName,
                                null,
                                PeerNetwork.UNICAST_PIPE);

        /*** Paso 2***/
```

```
        Message  msg = peerNetwork.poll(5000);

        /*** Paso 3***/
        if (msg == null)
            return null;

        /***Paso 4***/
        pipeId = processMessage(msg, messageID, "success");

        }
        catch (IOException ie) {
            ie.printStackTrace ();
        }
        return pipeId;
    }
```

Se puede ver en el código que el método *createPipe()* es muy parecido al método usado para crear un peer group. Es solo llamar al método *create()* de la clase PeerNetwork y a diferencia de la creación de un peer group en donde se le pasaba como primer parámetro la constante *PeerNetwork.GROUP* ahora se le pasa *PeerNetwork.PIPE* en su lugar.

Por otra parte, nótese que cuando se creó un peer group, no hubo necesidad de especificar el último parámetro al método *create()* , se le pasó *null.* Ahora en la creación de una pipe si es necesario especificar dicho argumento, ya que este representa el tipo de pipe a crear. Si se desea crear una pipe *Unicast* (punto a punto) se pasa *PeerNetwork.UNICAST* o si se desea crear una pipe de tipo *Propagate* (punto a multipunto) se le pasa *PeerNetwork.PROPAGATE_PIPE*, el método *create()* retorna el identificador del mensaje de petición para la creación de pipe (*requestId*).

Después de llamar al método *create(),* se debe llamar al método *poll(),* el cual retorna un objeto de la clase *Message.* Luego se le pasa este *Message* además del identificador de petición para la creación de la pipe (*requestId*) al método

processMessage(), el cual ya se vio anteriomente, y este finalmente retorna el identificador de la pipe (*pipeId*).

Cuando se crea exitosamente una pipe, esta no se abre de inmediato y empieza a escuchar sino hasta que explícitamente se le dé la orden. El siguiente ejemplo, demuestra este proceso de abrir y empezar a escuchar.

1.2.3.2. Cómo abrir una pipe de lectura.

Obsérvese con atención el método *listenToPipe* en el siguiente código:

```
//PipeDemo.java
public String listenToPipe(String pipeId){
    String listenOk = null;
    try{
        /***Paso 1***/
        int messageID = peerNetwork.listen (pipeId);

        /*** Paso 2***/
        Message msg = peerNetwork.poll(5000);

        /*** Paso 3***/
        if (msg == null)
            return null;

        /*** Paso 4***/
       listenOk = processMessage(msg, messageID, "success");

        if (listenOk.equals("success"))
        System.out.println (">>> listening over pipe… ");
        }
    catch (IOException ie) {
        ie.printStackTrace ();
    }
     return listenOk;}
```

Para abrir una pipe de lectura, sobre la cual escuchar y estar pendiente de mensajes entrantes es necesario llamar al método *listen()* de la clase PeerNetwork, y pasarle el identificador de la pipe, el *pipeId* . El método *listen()* devuelve entonces el identificador del mensaje de petición de escucha que se creó.

Después de llamar al método *listen(),* se llama al método *poll(),* este retorna un objeto de la clase *Message* el cual se le pasa al método *processMessage()* junto con el identificador del mensaje de petición de escucha (*requestId*). Si el método *processMessage()* no retorna *null,* esto significa que la solicitud de escucha fue exitosa y que la pipe ha sido abierta para lectura.

1.2.3.3. Cómo buscar una pipe

Si se quiere enviar un mensaje a un peer, es necesario conocer el *pipeId* del peer destino. Por lo tanto, primero hay que buscar una pipe. Después de conocer el *pipeId* del destino, se puede enviar directamente un mensaje a dicho peer a través de la pipe, como se muestra a continuación en el siguiente código:

```
//PipeDemo.java
public String searchPipe(String pipeName)
{
    String pipeId = null;
    try
    {
        /***Paso 1***/
        int messageID = peerNetwork.search (
                                PeerNetwork.PIPE,
                                "Name",
                                "JxmeTestPipe",
                                1);

        /*** Paso 2***/
        Message  msg = peerNetwork.poll(5000);
```

```
            /*** Paso 3***/
            if (msg == null)
                return null;

            /*** Paso 4***/
            pipeId = processMessage(msg, messageID, "result");
        }
        catch (IOException ie) {
            ie.printStackTrace();
        }
        return pipeId;
}
```

El método *searchPipe()* toma solo un parámetro, el nombre de la pipe. Este llama al método *search()* de la clase PeerNetwork, pasándole los cuatro argumentos mencionados previamente. Obsérvese en el código anterior que se le ha pasado la constante *PeerNetwork.PIPE* como primer parámetro del método *search().* Esto significa que se ha realizado una búsqueda de pipe. El método *search()* retorna el identificador del mensaje de búsqueda. Luego se llama al método *poll()* el cual retorna un objeto de la clase *Message.* Ahora se pasa tanto el identificador de petición de envío como el *Message* al método *processMessage(),* el cual verifica si el objeto *Message* es una respuesta al mensaje de búsqueda y extrae el identificador de pipe, *pipeId* del *Message.*

1.2.3.4. Cómo crear un mensaje JXME

Para la creación de mensajes JXME es necesario emplear las clases *Element* y *Message.* A continuación se ha escrito un fragmento de código especial para demostrar el proceso.

```
//PipeDemo.java
public Message authorMessage (String messageString){
    /***Paso 1***/
```

```
    Element[] elements = new Element [1];

    /***Paso 2***/
    elements [0] = new Element("message",
                               messageString.getBytes(),
                               null,
                               null);
    /***Paso 3***/
    Message message = new Message (elements);
    return message;
}
```

El método *authorMessage()* toma una cadena como parámetro llamada *messageString,* la cual representa el contenido del mensaje que se quiere enviar por la pipe. En general, se han seguido los siguientes pasos:

1. Se crea un arreglo de elementos.
2. Se instancia un objeto de la clase *Element*, obsérvese que se ha pasado una cadena (por ejemplo, "*message*") como primer parámetro. Esta cadena especifica el nombre del elemento, el cual envuelve el contenido actual del mensaje. El segundo parámetro especifica el contenido del mensaje (el parámetro *messageString*), el cual debe ser pasado como un arreglo de bytes, por ejemplo *messageString.getBytes().* El tercer elemento especifica el namespace del elemento, el cual, si se pasa *null* indica que se quiere crear un elemento sin namespace. El último parámetro especifica el MIME type del mensaje (se pasa *null* si se quiere especificar el MIME type por defecto).
3. Finalmente se utiliza el arreglo de bytes del paso 1, el cual contiene solo un objeto *Element,* para construir el objeto *Message.*

El método *authorMessage()* retorna el objeto *Message,* que corresponde al mensaje que se enviará por la pipe.

1.2.3.5. Cómo enviar un mensaje a través de una pipe

El siguiente código muestra cómo enviar un mensaje por una pipe usando el método *sendMessage().*

```
//PipeDemo.java
public String sendMessage(Message message, String pipeId){
    String sentOk = null;
    try{
        /***Paso 1***/
        int messageID = peerNetwork.send (pipeId, message);

        /***Paso 2***/
        Message  msg = peerNetwork.poll(5000);

        /*** Paso 3***/
        if (msg == null)
            return null;

        /*** Paso 4***/
        sentOk = processMessage(msg, messageID, "success");
    }
    catch (IOException ie){
        ie.printStackTrace();
    }
    return sentOk;
}
```

El método *sendMessage()* toma un objeto de la clase *Message* y un identificador de pipe (*pipeId*), y envía el *Message* a través de la pipe especificada por dicho *pipeId.* Internamente, el método *sendMessage()* llama al método *send()* de la clase PeerNetwork, le pasa el mensaje y el *pipeId* y finalmente este se encarga de enviar este mensaje por esa pipe. Luego de hacer esto, se llama al método *poll()* para obtener el mensaje que se creó y se envió previamente.

1.3. CÓDIGO FUENTE DE LA APLICACIÓN *PeerGroupDemo*

```
//PeerGroupDemo.java

import net.jxta.j2me.*;
import java.io.IOException;
import javax.microedition.midlet.MIDlet;

public class PeerGroupDemo extends MIDlet
{
    private PeerNetwork peerNetwork;

    public PeerGroupDemo()
    {

       try {
 peerNetwork = peerNetwork.createInstance ("JxmeTestPeer");

            //You can change Relay_URL address from
localhost to your own.
            byte[] persistentState = peerNetwork.connect
("http://localhost:9700", null);
            System.out.println(">>> connected to relay.. ");

            String groupId =
createPeerGroup("JxmeTestGroup");
            String searchedGroupId =
searchPeerGroup("JxmeTestGroup");
            String joinRequestOk =
joinPeerGroup(searchedGroupId);
        }
        catch (IOException ie) {
            ie.printStackTrace ();
        }

    }


    public String createPeerGroup(String groupName)
    {

        String newGroupId = null;
        try
        {
            /***Paso 1***/
            int messageID = peerNetwork.create
(PeerNetwork.GROUP, groupName, null, null);
            System.out.println(">>> creating peer group...
");
```

```
            /***Paso 2***/
            Message msg = peerNetwork.poll (5000);

            /***Paso 3***/
            if (msg == null)
                return null;

            /***Paso 4***/
            newGroupId = processMessage(msg, messageID,
"success");
            System.out.println(">>> new peergroup id :
"+newGroupId);

        }

        catch (IOException ie) {
            ie.printStackTrace ();
        }
        return newGroupId;
    }


    public String searchPeerGroup(String groupName)
    {

        String newGroupId = null;
           try
        {
            /***Paso 1***/
            int messageID = peerNetwork.search
(PeerNetwork.GROUP, "Name", groupName, 1);
            System.out.println(">>> group search requet id :
"+messageID);

            /***Paso 2***/
            Message msg = peerNetwork.poll (5000);

            /***Paso 3***/
            if (msg == null)
                return null;

            /***Paso 4***/
            newGroupId = processMessage(msg, messageID,
"result");

            System.out.println(">>> searched peergroup id :
"+newGroupId);

        }

        catch (IOException ie) {
```

```
            ie.printStackTrace ();
        }
        return newGroupId;
    }

    public String joinPeerGroup(String groupId)
    {
        String joinOk = null;

        try
        {
            /***Paso 1***/
            int messageID = peerNetwork.join (groupId,
null);
            System.out.println(">>> join peerGroup request..
" );

            /***Paso 2***/
            Message msg = peerNetwork.poll (5000);

            /***Paso 3***/
            if (msg == null)
                return null;

            /***Paso 4***/
            joinOk = processMessage(msg, messageID,
"success");

            System.out.println(">>> join peergroup request
succeeded.. ");
        }

        catch (IOException ie) {
            ie.printStackTrace ();
        }
        return joinOk;
    }



    private String processMessage (Message msg, int reqId,
String successCriterion)
    {
        int requestIdentifierInResponse = 0;
        String searchedResourceIdentifier = null;
        String responseString = null;

        try
        {
            /***Paso 1***/
            for (int j=0; j < msg.getElementCount(); j++) {
                /***Paso 2***/
```

```
                Element e = msg.getElement(j);
                /***Paso 3***/
                if ((e.getNameSpace()).equals("proxy")) {
                    if ((e.getName()).equals("requestId"))
                        requestIdentifierInResponse        =
Integer.parseInt(new String(e.getData()));
                    else                                    if
((e.getName()).equals("response"))
                        responseString    =    new    String
(e.getData());
                    else if ((e.getName()).equals("id"))
                        searchedResourceIdentifier   =   new
String (e.getData());
                }
            } //for (int j=0;)

            /***Paso 4***/
            if (requestIdentifierInResponse == reqId) {
                /***Paso 5***/
                if                    (responseString.equals
(successCriterion)) {
                    if (searchedResourceIdentifier != null)
                        /***Paso 6***/
                        return searchedResourceIdentifier;
                    else
                        /***Paso 7***/
                        return responseString;
                }
            }//if (requestIdentifierInResponse == reqId)

        }
        catch(NumberFormatException ne){
            ne.printStackTrace();
        }

        return null;
    }

    public void startApp()    {
    }

    protected void pauseApp() {
    }

    protected void destroyApp(boolean unconditional){
    }
}
```

1.4. CÓDIGO FUENTE DE LA APLICACIÓN *PipeDemo*

```
//PipeDemo.java

import net.jxta.j2me.*;
import java.io.IOException;
import javax.microedition.midlet.MIDlet;


public class PipeDemo extends MIDlet
{
    private PeerNetwork peerNetwork;

    public PipeDemo ()
    {
        try {
            peerNetwork     =     PeerNetwork.createInstance
("JxmeTestPeer");
// You may change Relay_URL (i.e. http://localhost:9700) to
//your own Relay IP.
            byte[]  persistentState  =  peerNetwork.connect
("http://localhost:9700", null);

            String pipeId = createPipe("JxmeTestPipe");
            String listenOk = listenToPipe(pipeId);
            String             searchedPipeId             =
searchPipe("JxmeTestPipe");
            Message msg = authorMessage("Hello World!");
            String    sendOk    =    sendMessage    (msg,
searchedPipeId);

        }
        catch(IOException ie){
            ie.printStackTrace();
        }
    }


    public String createPipe(String pipeName)
    {
        String pipeId = null;
        try
        {
            /***Paso 1***/
            int     messageID     =     peerNetwork.create
(PeerNetwork.PIPE,             pipeName,             null,
PeerNetwork.UNICAST_PIPE);
            System.out.println(">>> create pipe request ...
");

            /***Paso 2***/
```

```
            Message  msg = peerNetwork.poll(5000);

            /***Paso 3***/
            if (msg == null)
                return null;

            /***Paso 4***/
            pipeId    =    processMessage(msg,    messageID,
"success");

            System.out.println(">>>   create   pipe   request
succeed, id : "+pipeId );

        }
        catch (IOException ie) {
            ie.printStackTrace ();
        }

        return pipeId;

    }//createPipe()


    public String listenToPipe(String pipeId)
    {
        String listenOk = null;
        try
        {
            /***Paso 1***/
            int messageID = peerNetwork.listen (pipeId);

            /***Paso 2***/
            Message msg = peerNetwork.poll(5000);

            /***Paso 3***/
            if (msg == null)
                return null;

            /***Paso 4***/
            listenOk    =    processMessage(msg,    messageID,
"success");

            System.out.println (">>> listening succeed over
pipe... ");
        }
        catch (IOException ie) {
            ie.printStackTrace ();
        }

        return listenOk;
    }
```

```
    public String searchPipe(String pipeName)
    {
        String pipeId = null;
        try
        {
            /***Paso 1***/
int   messageID   =   peerNetwork.search   (PeerNetwork.PIPE,
"Name", "JxmeTestPipe", 1);

            /***Paso 2***/
            Message  msg = peerNetwork.poll(5000);

            /***Paso 3***/
            if (msg == null)
                return null;

            /***Paso 4***/
            pipeId    =    processMessage(msg,    messageID,
"result");

            System.out.println(">>> search pipe id found :
"+pipeId );

        }
        catch (IOException ie) {
            ie.printStackTrace();
        }

        return pipeId;
    }


    public Message authorMessage (String messageString)
    {
            /***Paso 1***/
            Element[] elements = new Element [1];

            /***Paso 2***/
            elements    [0]    =    new    Element("message",
messageString.getBytes(), null, null);

            /***Paso 3***/
            Message message = new Message (elements);

            return message;
    }

    public   String   sendMessage(Message   message,   String
pipeId)
    {
        String sentOk = null;
```

```
        try
        {
            /***Paso 1***/
            int   messageID   =   peerNetwork.send   (pipeId,
message);

            /***Paso 2***/
            Message  msg = peerNetwork.poll(10000);

            /***Paso 3***/
            if (msg == null)
                return null;

            /***Paso 4***/
            sentOk    =    processMessage(msg,    messageID,
"success");

            System.out.println(">>> jxme message sent... "
);
        }
        catch (IOException ie){
            ie.printStackTrace();
        }

        return sentOk;
    }


    private String processMessage (Message msg, int reqId,
String successCriterion)
    {
        int requestIdentifierInResponse = 0;
        String searchedResourceIdentifier = null;
        String responseString = null;

        try
        {
            /***Paso 1***/
            for (int j=0; j < msg.getElementCount(); j++) {
                /***Paso 2***/
                Element e = msg.getElement(j);
                /***Paso 3***/
                if ((e.getNameSpace()).equals("proxy")) {
                    if ((e.getName()).equals("requestId"))
                  requestIdentifierInResponse              =
Integer.parseInt(new String(e.getData()));
                  else                                    if
((e.getName()).equals("response"))
                      responseString    =    new    String
(e.getData());
                  else if ((e.getName()).equals("id"))
```

```
                        searchedResourceIdentifier   =   new
String (e.getData());
                }
            } //for (int j=0;)

            /***Paso 4***/
            if (requestIdentifierInResponse == reqId) {
                /***Paso 5***/
                if                    (responseString.equals
(successCriterion)) {
                    if (searchedResourceIdentifier != null)
                        /***Paso 6***/
                        return searchedResourceIdentifier;
                    else
                        /***Paso 7***/
                        return responseString;
                }
            }//if (requestIdentifierInResponse == reqId)

        }
        catch(NumberFormatException ne){
            ne.printStackTrace();
        }

        return null;
    }

    public void startApp() {
    }

    protected void pauseApp() {
    }

    protected void destroyApp(boolean unconditional) {
    }
}
```