6. Tests para PushRegistry

6.1. Introducción

Una vez visto en que consiste *PushRegistry* y cómo es su API, además de los aspectos más importantes de la plataforma J2ME y la tecnología Bluetooth, estamos preparados para describir lo que se ha realizado para llevar estos aspectos a la práctica. Lo que se ha desarrollado ha sido una aplicación cuya función es la de probar estas características en una determinada implementación.

En particular lo que se pretende con esta aplicación es poder ejecutarla en dispositivos de la Serie 40 de Nokia en los que se está implementando la característica *PushRegistry* sobre conexión Bluetooth. Con esto se consigue comprobar si la implementación que realizan los desarrolladores funciona según lo previsto.

La idea no es sólo la de hacer una aplicación que funcione, sino que intente poner a prueba la implementación de todas las formas posibles. De hecho, el objetivo no es ver que la implementación funciona, sino también en que casos y por qué no funciona.

Para el diseño de los tests se parte de la descripción de la arquitectura de *software* de la implementación. Esta documentación describe las especificaciones funcionales a seguir por el *software* en forma de casos de uso. Estos se intentan describir con el mayor detalle posible, para que los desarrolladores se ajusten a ellos a la hora de programar.

En este capítulo veremos primeramente un breve resumen de estos casos de uso, para poderse hacer una idea de cuáles han sido los requerimientos que se han seguido a la hora de diseñar los tests. A continuación se describirá el proceso de pruebas diseñado explicándose además como llevarlo a cabo.

6.2. Escenarios técnicos de uso

En este apartado se describen los diferentes escenarios de uso. La decripción de los posibles escenarios es muy importante, pues los tests se diseñarán en función de estos. Es decir, los tests se diseñarán teniendo en cuenta las distintas circunstancias que se describen en los distintos escenarios. En cada uno se describen distintas operaciones que se pueden llevar a cabo, y cómo debe responder la implementación en cada caso, cosa que deberán comprobar los tests. Lo que aquí se describe es un resumen de la especificación de la arquitectura de *PushRegistry*, que es la que marca con todo detalle como se debería comportar la implementación en cada caso particular.

6.2.1. Registro estático de conexión Push

1. Un MIDlet se instala con atributos *Push* en su fichero JAD o manifiesto del JAR con lo que se realiza el registro estático del mismo:
   - Todas las conexiones registradas se incluyen en el *PushRegistry*.
   - Las entradas de este *PushRegistry* se almacenan en el sistema de ficheros para que se conserven aunque el teléfono esté apagado.
   - Se crea una entrada del servicio en la SDDB.
   - El contenido completo del *Service Record* existe en el sistema de ficheros.
2. El MIDlet podrá permanecer sin iniciar o podrá haberse iniciado ya.
3. El servicio Bluetooth registrado será visible y conectable por dispositivos remotos.
4. El MIDlet puede ser iniciado por conexión *Push*.

6.2.2. Eliminación estática de conexión Push

1. Un MIDlet que previamente ha realizado algún registro de conexión *Push* es desinstalado del teléfono:
   - Se liberan todos los recursos ocupados del *PushRegistry*, sistema de ficheros (correspondientes al *PushRegistry* y *Service Record*), y SDDB.
   - El MIDlet será borrado del sistema.
2. Los servicios Bluetooth relacionados dejarán de ser visibles para los demás dispositivos.

6.2.3. Registro dinámico de conexión Push

1. El MIDlet llama al método `PushRegistry.registerConnection` y se realiza un registro de forma dinámica:
   - Todas las conexiones registradas se incluyen en el *PushRegistry*.
   - Las entradas de este *PushRegistry* se almacenan en el sistema de ficheros para que se conserven aunque el teléfono esté apagado.
   - Se crea una entrada del servicio en la SDDB.
   - El contenido completo del *Service Record* existe en el sistema de ficheros.
2. El MIDlet se puede cerrar o dejar abierto.

3. El servicio Bluetooth registrado será visible y conectable por dispositivos remotos.

4. El MIDlet puede ser iniciado por conexión *Push*.

6.2.4. Eliminación dinámica de conexión Push

1. Un MIDlet que previamente ha realizado algún registro de conexión Push utiliza el método `PushRegistry.unregisterConnection()` para eliminar el registro de la conexión.

   - Se liberan todos los recursos ocupados del *PushRegistry*, sistema de ficheros (correspondientes al *PushRegistry* y *Service Record*), y SDDB.

   - El MIDlet será borrado del sistema.

2. Los servicios Bluetooth relacionados dejarán de ser visibles para los demás dispositivos.

6.2.5. Inicio del MIDlet mediante conexión entrante

1. Un dispositivo remoto detecta el servicio previamente registrado, y se conecta a él.

2. El MIDlet se iniciará automáticamente y podrá hacer uso de la conexión Bluetooth.

   - Antes de realizar el inicio del MIDlet se comprobará el filtro de la conexión. Es decir, el filtro puede restringir el conjunto de dispositivos autorizados a iniciar automáticamente el MIDlet, de forma que el dispositivo remoto no esté autorizado a ello. En este caso la conexión se rechazará directamente junto con los datos enviados.

   - Además el inicio automático del MIDlet tendrá que ser confirmado por el usuario.

3. El MIDlet hace uso del método `PushRegistry.listConnections(true)` y obtiene la cadena de la conexión entrante.

4. El MIDlet llama a `Connector.open()` y obtiene el objeto `Notifier` correspondiente a la conexión.

5. El MIDlet llama a `notifier.acceptAndOpen()` y obtiene un objeto de conexión válido.

6. El MIDlet abre los flujos `OutputStream` y `InputStream` que usará para enviar y recibir datos, o utiliza los métodos de transmisión de datos L2CAP. Además podrá disponer de los datos que ya hayan sido enviados por el dispositivo remoto, que se encontrarán almacenados en un *buffer*.

6.2.6. Conexión entrante en MIDlet previamente iniciado

1. El MIDlet es iniciado manualmente por el usuario.

2. Al llamar al método `PushRegistry.listConnections(true)` obtiene una lista de conexiones vacía. De esta forma el MIDlet puede saber que fue iniciado manualmente y no por conexión entrante.

3. Si llama al mismo método, pero con `false` como parámetro, obtendrá la lista de las conexiones registradas. De estas seleccionará la que desea utilizar y, por lo tanto, a la que se pondrá a esperar. Es posible utilizar varias.

4. El MIDlet obtendrá el objeto `Notifier` asociado a cada conexión mediante `Connector.open()`.

5. El MIDlet llama a `notifier.acceptAndOpen()` para cada una de las conexiones que quiera aceptar y se pondrá a esperar a que obtenga un objeto de conexión válido.

6. Cuando se produzca una conexión entrante el MIDlet obtendrá el objeto de conexión asociado a ella.

7. Una vez aceptada la conexión, el MIDlet abre los flujos `OutputStream` y `InputStream` que usará para enviar y recibir datos, o utilizará los métodos de transmisión de datos L2CAP.

6.2.7. MIDlet con conexión entrante llama a Notifier.close

1. El MIDlet que utiliza una conexión entrante, llama a `Notifier.close()`.

2. El servicio dejará de ser visible y conectable a partir de entonces, pero la conexión sigue activa pudiendo incluso seguir intercambiando datos. La conexión no se cerrará hasta que no se cierren el objeto de conexión y los flujos de datos.

3. Si se llama a `PushRegistry.listConnections(true)` se obtiene una lista vacía.

4. Si se llama con `false` la lista contendrá todas las conexiones que estén registradas.

5. El MIDlet se cierra.

6. Una vez cerrado el MIDlet, el servicio volverá a ser visible y conectable para los dispositivos remotos.

6.2.8. MIDlet registrado modifica el servicio Push

1. El MIDlet llama a `LocalDevice.getLocalDevice()` para obtener el objeto `LocalDevice`.

2. Suponiendo que existen conexiones registradas, llama a `PushRegistry.listConnections (false)` para obtener una lista de éstas.

3. El MIDlet selecciona la conexión a la que le quiere cambiar el contenido del servicio, y obtiene el objeto `Notifier`, que le hará falta para obtener el `ServiceRecord` asociado a la conexión, mediante el método `Connector.open()`.
4. Una vez que tiene el objeto `Notifier` el MIDlet podrá obtener el objeto `ServiceRecord` mediante `localDevice.getRecord()`.
5. El contenido del `ServiceRecord` se modifica con `ServiceRecord.setAttributeValue()`.
6. La SDDB se actualiza con el nuevo `ServiceRecord` con `localDevice.updateRecord()`.
7. El nuevo contenido será visible y conectable por dispositivos remotos.
   - El objeto `ServiceRecord` que se obtiene con `getRecord()` debe contener los datos actualizados.
   - El contenido será actualiza en la SDDB y en el sistema de ficheros.

6.2.9. Bluetooth se activa mientras existen conexiones registradas

1. El MIDlet se encuentra instalado en el teléfono con una o más conexiones registradas, mientras que Bluetooth se encuentra desactivado.
2. El usuario activa Bluetooth mediante el menú del teléfono.
3. El contenido del *Service Record* será leído del sistema de ficheros y se actualizará la SDDB.
4. El contenido actualizado será visible y conectable para los dispositivos remotos.
5. El MIDlet podrá ser iniciado automáticamente por conexión entrante.

6.2.10. Dispositivo con conexiones registradas es apagado y encendido

1. El MIDlet se encuentra instalado en el teléfono una o más conexiones registradas, mientras que Bluetooth se encuentra activado.
2. El usuario apaga y enciende el teléfono.
3. Se leen todas las entradas del sistema de ficheros correspondientes al *Push Registry*.
4. El servidor de JSR82 recibe de nuevo los registros por parte del servidor *Push*, y procede a actualizar sus entradas en el sistema de ficheros y la SDDB.
5. El contenido actualizado será visible y conectable para los dispositivos remotos.
6. El MIDlet podrá ser iniciado automáticamente por conexión entrante.

6.3. Descripción de los tests

A continuación se describe en que consiste el escenario de pruebas desarrollado. Como en cada proceso de *testing* se debe especificar qué es lo que se pretende probar, qué es lo que hace falta para la ejecución del test y cómo se realiza ésta. Por ello cada test es descrito incluyendo los puntos siguientes:

- **Objetivo:** Indica cuál es el método u operación a testear describiendo brevemente que es lo que debería ocurrir si se llevan a cabo con éxito.
- **Condiciones previas:** Se indican cuáles son los dispositivos y MIDlets que se van a utilizar y deben estar instalados, y si previamente se debe haber realizado algún tipo de operación como puede ser la ejecución de otro test.
- **Descripción:** Paso a paso se verá qué es lo que realiza el test y que es exactamente lo que se comprueba lo largo de éste.
- **Ejecución:** A modo de manual orientado al encargado de ejecutar el test, se explican cuáles son los pasos a seguir para realizarlo.

Los tests consisten básicamente en la ejecución de una serie de MIDlets que harán uso de la clase `PushRegistry`. Principalmente se hará uso de un MIDlet denominado `PushRegistryTest` que habrá que ejecutar prácticamente en todas las pruebas, pero además en algunas harán falta `StaticRegistrationTest`, `PushConnectorOpenTest` o `PusUnregistrationTest`.

Generalmente el test comenzará en el lado del servidor, que hará uso de la API de *PushRegistry,* y finalizará en el cliente. Pero esto sólo es una consideración general que no se cumplirá en algunos de los escenarios.

En el dispositivo cliente siempre se ejecutará la aplicación `PushRegistryTest`. Al seleccionar "Client" en el menú inicial se realizará una búsqueda de dispositivos en la que habrá que seleccionar el dispositivo que hará de servidor. A partir de entonces se podrán seleccionar los distintos tests en el menú "Test Case". Esta búsqueda del servidor no es necesario realizarla de nuevo para cada test. Por lo tanto en la descripción de la ejecución de los tests que vienen a continuación se supondrá que el cliente a seleccionado ya el servidor.

Figura 6.1 Inicio del cliente

Otra consideración importante es que los tests están pensados para realizarse en el orden que aquí vienen descritos, pues se va haciendo uso de métodos previamente testeados con éxito. La única excepción es el método `listConnections()`, que se usará desde un principio para obtener las conexiones que hay registradas en el *Push Registry* sin haber sido testeado anteriormente. En principio se supondrá un correcto funcionamiento de este método, cosa que se comprobará en el test que hay destinado a él.

6.3.1. Registro estático

a) Objetivo

El registro estático que se realiza al instalar un MIDlet con el atributo `MIDlet-Push-<n>` en el archivo JAD o Manifiesto. Una vez realizado el registro deberá existir una entrada en el *Push Registry* correspondiente al MIDlet registrado. Además se deberá haber creado una entrada en la base de datos de servicios de Bluetooth. Esto implica que el servicio debe ser visible por otros dispositivos aún sin iniciarse el MIDlet.

b) Condiciones previas

Será necesario un servidor y un cliente. Este es un caso un poco particular pues en el servidor no hará falta tener ningún MIDlet instalado previamente, pues la instalación del MIDlet `StaticRegistrationTest` formará parte del propio test. En el cliente sin embargo, sí que será necesario tener instalado `PushRegistryTest`.

c) Descripción

1. Al instalarse `StaticRegistrationTest` en el dispositivo servidor se realizará el registro *push*.
2. Con el cliente se realizará una búsqueda de servicios en el dispositivo servidor, para tratar de encontrar el servicio asociado a la conexión registrada.
3. Por otro lado el MIDlet del servidor comprobará que se haya creado una entrada en el *Push Registry*. Esto se realizará con el método `listConnections()`.
4. Una vez iniciada la aplicación del servidor el cliente volverá a realizar la búsqueda de servicios.

d) Ejecución

1. Se instala el MIDlet `StaticRegistrationTest` en el dispositivo servidor. No se iniciará la aplicación todavía.
2. En el otro dispositivo se selecciona "Registration Test" y se espera hasta terminar el test.
3. Se inicia `StaticRegistrationTest` en el dispositivo servidor.

4. En el dispositivo cliente se volverá a ejecutar "Registration Test".

6.3.2. Eliminación estática de registro

a) Objetivo

La eliminación estática de registro que se produce automáticamente al desinstalar el MIDlet registrado. De este modo deberá desaparecer la entrada correspondiente a la conexión registrada del *Push Registry*, así como el servicio de la SDDB.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. Además será indispensable haber realizado antes el test "Registro estático", en el que se habrá instalado `StaticRegistrationTest`, que deberá permanecer todavía instalado en el dispositivo servidor.

c) Descripción

1. Al desinstalarse `StaticRegistrationTest` del dispositivo servidor se eliminará el registro *push*.
2. El servidor comprobará que haya desaparecido la entrada del *Push Registry*. Esto se realizará con el método `listConnections()`.
3. Con el cliente se realizará una búsqueda de servicios en el dispositivo servidor, para confirmar que el servicio no puede ser encontrado.

d) Ejecución

1. Se desinstala el MIDlet `StaticRegistrationTest` del dispositivo servidor.
2. Se inicia `PushRegistryTest` en el servidor, y se selecciona "Static Unregistration Test".
3. En el cliente se selecciona también "Static Unregistration Test" y se espera hasta terminar el test.

6.3.3. Registro dinámico

a) Objetivo

El registro dinámico que se realiza llamando al método `registerConnection()`. Al igual que en el estático, una vez realizado el registro deberá existir una entrada en el *Push Registry*

correspondiente al MIDlet. Además se deberá haber creado una entrada en la SDDB. Esto implica que el servicio debe ser visible por otros dispositivos aún con el MIDlet cerrado.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. Es importante que no haya ninguna conexión registrada asociada a este MIDlet, pues entonces el test fallaría. Esto puede ocurrir, por ejemplo, si se ejecuta este test dos veces seguidas.

c) Descripción

1. Se llama al método `registerConnection()` con un nombre de MIDlet erróneo por lo que el sistema debe lanzar la excepción `ClassNotFoundException`.
2. Se vuelve a llamar al método `registerConnection()` pero con los parámetros adecuados. La conexión debe estar registrada a partir de este momento.
3. Se comprueba la existencia del registro en el *Push Registry* mediante el método `listConnections()`.
4. Se llama de nuevo al método `registerConnection()` con los parámetros adecuados. Esto debe dar lugar a una `IOException`.
5. Se comprueba que se ha creado una entrada en la SDDB y que el servicio es visible. Esto se hará con ayuda del cliente, que realizará una búsqueda de servicios en el dispositivo.

d) Ejecución

1. Se inicia `PushRegistryTest` en el dispositivo servidor y se selecciona "Registration Test". Una vez finalizado el test, se cierra el MIDlet.
2. Se ejecuta "Registration Test" en el cliente.

6.3.4. Eliminación dinámica de registro

a) Objetivo

La eliminación dinámica de registro que se produce al llamar al método `unregisterConnection()`. Al eliminar el registro deberá desaparecer la entrada correspondiente a la conexión registrada del *Push Registry*, así como el servicio de la SDDB.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`, pero además en el servidor hará falta ejecutar `UnregistrationTest`. Además será indispensable haber realizado antes el test "Registro dinámico", en el que se habrá registrado la conexión a eliminar.

c) Descripción

1. El MIDlet `UnregistrationTest` llama a `unregisterConnection()` lo que debe provocar una `SecurityException` ya que la conexión no fue registrada por él.
2. `PushRegistryTest`, que fue el MIDlet que realizó el registro, llama ahora al método `unregisterConnection()` con los parámetros adecuados.
3. El MIDlet del servidor comprobará que no se encuentre la entrada en el Push Registry. Esto se realizará con el método `listConnections()`.
4. Con el cliente se realizará una búsqueda de servicios en el dispositivo servidor, para confirmar que el servicio no puede ser encontrado en la SDDB.

d) Ejecución

1. Se ejecuta `UnregistrationTest`. Una vez finalizado se cierra el MIDlet.

1. Se inicia `PushRegistryTest` en el dispositivo servidor y se selecciona "Dynamic Unregistration Test".
2. Una vez finalizado el test en el servidor, se ejecuta "Dynamic Unregistration Test" en el cliente.

6.3.5. MIDlet iniciado automáticamente por conexión entrante

a) Objetivo

Inicio automático del MIDlet mediante la conexión entrante de un dispositivo remoto.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada. Si no está registrada habrá que ejecutar el test "Registro dinámico" antes de llevar a cabo éste.

c) Descripción

1. Suponiendo la conexión registrada y el MIDlet cerrado en el servidor, el cliente trata de establecer una conexión con éste, buscando previamente el servicio.
2. El MIDlet del servidor se iniciará automáticamente.

d) Ejecución

1. Se ejecuta "Incoming Connection Test" en el cliente.
2. El MIDlet del servidor se iniciará automáticamente. Si esto ocurre el test se habrá pasado.

6.3.6. Uso de la conexión entrante por MIDlet iniciado manualmente

a) Objetivo

Uso de la conexión entrante registrada por el MIDlet, que ha sido previamente iniciado de forma manual.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada. Si no está registrada habrá que ejecutar el test "Registro dinámico" antes de éste.

c) Descripción

1. El MIDlet del servidor será iniciado.
2. El MIDlet llama al método `Connector.open()`, con la conexión registrada y obtiene un objeto `Notifier`.
3. El MIDlet llama a `Notifier.acceptAndOpen()` y espera a obtener un objeto de conexión.
4. El cliente trata de establecer una conexión con el servidor, buscando previamente el servicio.
5. El MIDlet del servidor obtiene ahora el objeto de conexión, y podrá abrir los flujos `InputStream` y `OutputStream` para enviar y recibir datos.

d) Ejecución

1. Se ejecuta "Use Connection Test" en el servidor.
2. Cuando la aplicación del servidor lo pida, comezar "Use Connection Test" en el cliente.

6.3.7. Uso de la conexión entrante por MIDlet iniciado automáticamente

a) Objetivo

Uso de la conexión entrante registrada por el MIDlet, que ha provocado el inicio automático del MIDlet.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada. Si no está registrada habrá que ejecutar el test "Registro dinámico" antes de éste.

c) Descripción

1. El cliente trata de establecer una conexión con el servidor, buscando previamente el servicio.
2. El MIDlet del servidor se iniciará automáticamente.
3. El servidor llama al método `Connector.open()`, con la conexión registrada y obtiene un objeto `Notifier`.
4. Seguidamente llama a `Notifier.acceptAndOpen()` y espera a obtener un objeto de conexión.
5. Cuando el cliente se conecte el servidor obtiene el objeto de conexión, y podrá abrir los flujos `InputStream` y `OutputStream` para enviar y recibir datos.

d) Ejecución

1. Se ejecuta "Use Connection Test" en el cliente.
2. El MIDlet del servidor se iniciará automáticamente. Una vez iniciado, ejecutar "Use Connection Test".

6.3.8. Listar conexiones

a) Objetivo

El método `listConnections()` encargado de devolver una lista de las conexiones registradas en el momento. Si recibe `false` como parámetro deberá devolver todas las conexiones registradas, pero si el parámetro es `true` sólo listará las conexiones activas en el momento.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada. Si no está registrada habrá que ejecutar el test "Registro dinámico" antes de éste.

c) Descripción

1. El MIDlet llama al método `listConnections()` con parámetro `false`, con lo que devolverá la conexión registrada.
2. El MIDlet llama al método `listConnections()` con parámetro `true`, pero al no estar la conexión establecida no aparecerá en la lista.
3. El cliente establece la conexión con el MIDlet.
4. El MIDlet llama de nuevo a `listConnections()` con `true` como parámetro, recibiendo ahora la conexión registrada.

d) Ejecución

1. Se ejecuta "List Connections Test" en el servidor.
2. Cuando la aplicación del servidor lo pida, comenzar "List Connections Test" en el cliente.

6.3.9. Actualizar Push Service Record

a) Objetivo

Modificación del *Service Record* asociado a una conexión registrada.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada. Si no está registrada habrá que ejecutar el test "Registro dinámico" antes de éste.

c) Descripción

1. El servidor obtiene el objeto `connectionNotifier` llamando al método `Connector.open()`.
2. Con éste obtiene el objeto `ServiceRecord` asociado a la conexión llamando al método `LocalDevice.getRecord()`.
3. Se modifica el `ServiceRecord` añadiendo algún elemento nuevo.

4. Por último llama a `LocalDevice.updateRecord()` para actualizar el `ServiceRecord`.
5. Una vez actualizado el `ServiceRecord`, lo lee para comprobar que verdaderamente se ha modificado.
6. El cliente remoto buscará el servicio para confirmar que la modificación es visible.

d) Ejecución

1. Se ejecuta "Modify Service Test" en el servidor. Cuando finalice cerrar MIDlet.
2. Comenzar "Modify Service Test" en el cliente.

6.3.10. Excepción en Connector.open

a) Objetivo

El método `Connector.open()` que llamado desde el MIDlet que registró la conexión devolverá el objeto `ConnectionNotifier` asociado a la conexión. Pero este comportamiento positivo del método ya ha sido comprobado en tests anteriores. En éste se realiza un test negativo, que se basa en que si el método trata de abrir la conexión registrada desde otro MIDlet el sistema lanzará `IOException`.

b) Condiciones previas

Solo necesitaremos el servidor en el que se ejecutará el MIDlet `ConnectorOpenTest`. La conexión deberá estar anteriormente registrada.

c) Descripción

1. Un MIDlet distinto al que registró previamente la conexión, trata de abrirla llamando a `Connector.open()`.
2. El sistema lanzará una excepción del tipo `IOException`.

d) Ejecución

1. Se ejecuta el MIDlet `ConnectorOpenTest` en el servidor.

6.3.11. Cerrar conexión

a) Objetivo

El método `close()` del objeto `StreamConnection`. La conexión se cerrará pero la conexión debe seguir registrada y visible, de forma que se puedan seguir aceptando conexiones.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada. Si no está registrada habrá que ejecutar el test "Registro dinámico" antes de éste.

c) Descripción

1. Un cliente establece una conexión *push* con el servidor.
2. El MIDlet que se ejecuta en el servidor llama a `StreamConnection.close()`.
3. El servidor volverá a esperar a clientes haciendo una nueva llamada a `acceptAndOpen()`.
4. El cliente vuelve a realizar la búsqueda del servicio y se volverá a conectar a él.

d) Ejecución

1. En el cliente se selecciona "Connection Close Test".
2. El servidor se iniciará automáticamente. Seleccionar "Connection Close Test". La conexión con el cliente se cerrará y el MIDlet volverá a esperar en `acceptAndOpen()`.
3. Cuando el servidor espera volver a ejecutar "Connection Close Test" en el cliente con lo que se establecerá de nuevo la conexión entre ambos.

6.3.12. Notifier.close

a) Objetivo

El método `Notifier.close()` en un MIDlet con conexión *push* establecida. Al llamar a este método la conexión dejará de ser visible y conectable hasta que el MIDlet se cierre.

b) Condiciones previas

Será necesario un servidor y dos clientes. En los tres se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada. Si no está registrada habrá que ejecutar el test "Registro dinámico" antes de llevar a cabo éste.

c) Descripción

1. Un cliente establece una conexión *push* con el servidor.
2. El MIDlet que se ejecuta en el servidor llama a `ConnectionNotifier.close()`.
3. El servicio dejará de ser visible, cosa que se comprobará con el segundo cliente, aunque la conexión seguirá activa.
4. El método `listConnections(true)` no deberá devolver la conexión registrada.
5. El MIDlet termina.
6. El servicio volverá a ser visible, cosa que se comprobará con el otro cliente.

d) Ejecución

1. En el cliente se selecciona "Notifier Close Test".
2. Cuando el servidor se inicie automáticamente ejecutar "Notifier Close Test". Cuando lo indique ejecutar en el otro cliente "Unregistration Test". Este test se debe pasar pues el servicio no debe ser visible. Una vez que termine este test cerrar el MIDlet del servidor.
3. Ejecutar ahora en el segundo cliente "Registration Test". Este test se debe pasar pues el servicio debe ser ahora visible.

6.3.13. Clientes permitidos

a) Objetivo

El filtro que se realiza al registrar la conexión que restringe la conexión al servidor, de forma que sólo los clientes especificados pueden realizar la conexión.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión no debe estar previamente registrada.

c) Descripción

1. El servidor registra la conexión utilizando como filtro cualquier dirección diferente a la del cliente, con lo que tendrá permitido establecer la conexión cuando el MIDlet esté cerrado.
2. El MIDlet del servidor será cerrado.
3. El cliente tratará de conectarse al servidor pero no podrá.
4. El MIDlet del servidor se inicia de nuevo y esperará a clientes llamando a `acceptAndOpen()`.
5. El cliente tratará de nuevo de conectarse al servidor y lo conseguirá, pues el filtro sólo debe funcionar en el caso de que la conexión sea para iniciar el MIDlet y no cuando el servidor espere normalmente a clientes. La conexión se cierra seguidamente

d) Ejecución

1. En el servidor se ejecuta "Not Allowed Test" y se cierra el MIDlet.
2. En el cliente seleccionar "Use Connection Test" con lo que intentará conectarse al servidor pero no debe conseguirlo.
3. Iniciar de nuevo el MIDlet del servidor y ejecutar "Use Connection Test", esperando el servidor así a que el cliente se conecte.
4. Volver a ejecutar "Use Connection Test" en el cliente. La conexión deberá poder establecerse.

6.3.14. Buffer de datos

a) Objetivo

El *buffer* de datos que almacena los datos cuando el cliente se conecta al servidor y le envía datos, pero el MIDlet todavía no ha sido iniciado.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada.

c) Descripción

1. El cliente busca el servicio, se conecta a él. Seguidamente abre el flujo `OutputStream` y envía algunos datos.
2. El MIDlet del servidor será automáticamente iniciado al recibir la conexión entrante.

3. Seguidamente llamará a `acceptAndOpen()` con lo que aceptará la conexión.
4. Abrirá el flujo `InputStream` y leerá los datos. Se comprobará que no haya habido ninguna pérdida de éstos.

d) Ejecución

1. En el cliente se ejecuta "Simple Buffer Test".
2. El MIDlet del servidor se iniciará automáticamente. Ejecutar el mismo test en el servidor.

6.3.15. Solidez del buffer de datos

a) Objetivo

El rebosamiento del *buffer* de datos sólo debe conducir a que se desechen los datos sobrantes.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada.

c) Descripción

1. El cliente busca el servicio, se conecta a él. Seguidamente abre el flujo `OutputStream` y envía una cantidad grande de datos (superior al *buffer* del servidor).
2. El MIDlet del servidor será automáticamente iniciado al recibir la conexión entrante.
3. Seguidamente llamará a `acceptAndOpen()` con lo que aceptará la conexión.
4. Abrirá el flujo `InputStream` y leerá los datos que existan en el *buffer*.

d) Ejecución

1. En el cliente se ejecuta "Robustness Buffer Test".
2. El MIDlet del servidor se iniciará automáticamente. Ejecutar el mismo test en el servidor.

6.3.16. Buffer de datos con acumulación de conexiones

a) Objetivo

El *buffer* debe ser capaz de almacenar los datos de varias conexiones entrantes para poder ser leídos más tarde, pudiéndose además distinguir qué datos corresponden a cada conexión.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada.

c) Descripción

1. El cliente busca el servicio, se conecta a él. Seguidamente abre el flujo `OutputStream` y envía algunos datos.
2. A continuación cierra la conexión, la vuelve a abrir y vuelve a enviar datos.
3. El MIDlet del servidor será automáticamente iniciado al recibir la primera conexión entrante.
4. Seguidamente llamará a `acceptAndOpen()` con lo que aceptará la primera conexión. Abrirá el flujo `InputStream` y leerá los datos que existan en el *buffer* correspondientes a la primera conexión.
5. Volverá a llamar a `acceptAndOpen()` y a abrir `InputStream` con lo que recibirá los datos de la segunda conexión.

d) Ejecución

1. En el cliente se ejecuta "Accumulation Buffer Test".
2. El MIDlet del servidor se iniciará automáticamente. Ejecutar el mismo test en el servidor.

6.3.17. Buffer vacío al cancelar inicio del MIDlet

a) Objetivo

Los posibles datos envíados por una conexión entrante que provoca el inicio automático del MIDlet deben ser descartados si el usuario no aprueba este inicio.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada.

c) Descripción

1. El cliente busca el servicio y se conecta a él. Seguidamente abre el flujo `OutputStream` y envía algunos datos.
2. El MIDlet del servidor comenzará a ejecutarse de manera automática. El usuario lo cancelará.
3. A continuación el MIDlet del servidor será iniciado manualmente y esperará a que se conecte el cliente.
4. El cliente se conectará al servidor y éste abrirá el flujo `InputStream` y leerá a ver si hay datos. El *buffer* deberá estar vacío.

d) Ejecución

1. En el cliente se ejecuta "Discard Buffer Test".
2. El MIDlet del servidor tratará de iniciarse automáticamente pero el usuario lo cancelará.
3. Ejecutar ahora "Discard Buffer Test" en el servidor.
4. Cuando el servidor espere en `acceptAndOpen()` ejecutar "Incoming Connection Test" en el cliente. El servidor aceptará la conexión y si no consigue leer ningún dato el test se habrá pasado.

6.3.18. Buffer vacío al terminar MIDlet

a) Objetivo

Los posibles datos envíados por una conexión entrante que provoca el inicio automático del MIDlet deben ser descartados si el MIDlet se termina, aunque no se hayan leído.

b) Condiciones previas

Será necesario un servidor y un cliente. En los dos se ejecutará `PushRegistryTest`. La conexión debe estar previamente registrada.

c) Descripción

1. El cliente busca el servicio y se conecta a él. Seguidamente abre el flujo `OutputStream` y envía algunos datos.
2. El MIDlet del servidor se ejecutará de manera automática. Una vez abierto, el MIDlet será cerrado por el usuario.
3. A continuación el MIDlet volverá a ser iniciado manualmente y esperará a que se conecte el cliente.
4. El cliente se conectará al servidor y éste abrirá el flujo `InputStream` y leerá a ver si hay datos. El *buffer* deberá estar vacío.

d) Ejecución

1. En el cliente se ejecuta "Discard Buffer Test".
2. El MIDlet del servidor se iniciará automáticamente. Una vez iniciado se cierra.
3. Ejecutar ahora "Discard Buffer Test" en el servidor.
4. Cuando el servidor espere en `acceptAndOpen()` ejecutar "Incoming Connection Test" en el cliente. El servidor aceptará la conexión y si no consigue leer ningún dato el test se habrá pasado.