Implementación de Servidor stub para la ejecución de pruebas unitarias en aplicaciones cliente basadas en XMPP

Pablo Szyrko, Pablo Perotti

Nimbuzz Argentina
Humberto Primo 525. (X5000FAK). Piso 6
Córdoba, Argentina
szyrko@nimbuzz.com.ar, pablo@nimbuzz.com.ar

Resumen. Las pruebas unitarias en la actualidad son consideradas como uno de los elementos claves al momento de plantear una estrategia de verificación y validación efectiva, no sólo aplicables a metodologías de desarrollo software ágil sino a cualquier proceso de desarrollo. Todo aquél que ha trabajo en el desarrollo de aplicaciones sobre la base de XMPP, o incluso en otros protocolos basados en XML o que implique la interacción permanente con un servidor, han experimentado las dificultades y desafíos que plantea la incorporación de prácticas de pruebas unitarias. En el presente trabajo se presenta la experiencia y los resultados obtenidos al implementar un servidor de stub que permita la ejecución de pruebas unitarias de aplicaciones cliente basadas en XMPP, con el fin de ser capaces de verificar las interacciones que se establecen a través de sus módulos con un servidor, de forma de aprovechar las ventajas que las pruebas unitarias proporcionan además de satisfacer requerimientos adicionales que proporcionan valor agregado en el proceso de desarrollo.

Keywords: Prueba unitaria - XMPP - XML – cliente móvil - Android

1 Introducción

El presente trabajo surge como consecuencia de una iniciativa de mejora de las prácticas de validación y verificación en el desarrollo de los clientes móviles de la aplicación Nimbuzz, a través del equipo del cliente Android, focalizado en el desarrollo y ejecución de prueba unitarias en el entorno del protocolo XMPP (eXtensible Messaging and Presence Protocol). Nimbuzz es un mensajero social móvil para múltiples comunidades y proveedor VoIP, que combina VoIP, Mensajería Instantánea y geo presencia. La aplicación está desarrollada sobre la base del protocolo XMPP, un protocolo de red abierto y basado en XML (Extensible Markup Language) para comunicaciones en tiempo real [1]. Al estar basado en XMPP, Nimbuzz está implementado en una arquitectura Cliente y Servidor. El concepto de cliente hace referencia a las diferentes implementaciones de Nimbuzz que están disponibles para que los usuarios utilicen la aplicación en diferentes equipos y plataformas. Entre ellos

se incluyen las implementaciones de Nimbuzz PC, Nimbuzz Web y Nimbuzz Mobile (implementaciones nativas para diferentes plataformas móviles: Android, Blackberry, Symbian, Windows Mobile, IPhone OS, J2ME). El concepto de servidor se aplica al soporte de infraestructura que permite la comunicación entre los clientes, incorporando adicionalmente el soporte para interactuar con las comunidades externas (Skype, Windows Live Messenger, Yahoo! Messenger, ICQ, Google Talk, etc.).

El trabajo se encuentra estructurado de forma de plantear primeramente la problemática que determina la necesidad de generar un esquema automatizado para la ejecución de casos de prueba de clientes XMPP, particularmente para la aplicación Nimbuzz. A continuación se plantea la solución propuesta a través del desarrollo del servidor Stub para la ejecución de pruebas unitarias. Posteriormente se explicitan los resultados obtenidos al utilizar el servidor Stub propuesto y las conclusiones finales.

2 Problemática

2.1 Las pruebas unitarias

El objetivo primario de las pruebas unitarias es el tomar piezas pequeñas de software testeables en la aplicación, aislarlas del resto del código, y determinar si éste se comporta de acuerdo a lo esperado. Cada unidad es testeada en forma separada antes de que sean integradas en módulos con el fin de testear las interfaces entre esos módulos [2].

La ejecución de pruebas unitarias proporciona ventajas al momento de desarrollar software [3], entre las que se destacan:

- Ser capaces de testear partes de un proyecto sin esperar a que las otras partes estén disponibles.
- Lograr paralelismo en la ejecución de las pruebas siendo capaces de testear y corregir problemas simultáneamente por diferentes desarrolladores.
- Ser capaces de detectar y remover defectos a un costo mucho menor comparado con la corrección en etapas posteriores de testeo.
- Simplificar el debugging limitando a unidades pequeñas las áreas de código en donde es posible encontrar bugs.
- Ser capaces de probar condiciones internas que no son fáciles de generar por entradas externas en grandes sistemas integrados.

2.2 Pruebas unitarias para aplicaciones clientes XMPP

El enfoque más común para la ejecución de las pruebas unitarias requiere drivers y stubs para ser escritos. El driver simula una unidad llamadora y el stub simula la unidad llamada. Este concepto se amplía aún más si estamos pensando ejecutar pruebas unitarias en ambientes del tipo cliente-servidor como el existente en Nimbuzz.

Como miembros del equipo de desarrollo del cliente móvil de Android se plantea la problemática de cómo aplicar la práctica de pruebas unitarias de forma que genere un

valor agregado en el producto desarrollado, ampliando el concepto de prueba unitaria previamente planteado. El objetivo es que la prueba unitaria sea capaz de verificar las interacciones que establece un cliente móvil con un servidor, en término del intercambio de stanzas XML a través de XMPP, probando que el comportamiento en el cliente sea el correcto ante diferentes condiciones de respuesta del servidor. De esta forma los componentes que serán probados a través de pruebas unitarias serán los módulos individuales que interactúan contra el servidor. Esto no invalida la creación y ejecución de prueba unitarias bajo el concepto original, sino que lo amplía a un nivel superior.

Este enfoque no es nuevo y ha sido planteado entre otros en el capítulo Beautiful XMPP Testing de [4] y también como parte de las estrategias de testeo de [5]. Sin embargo este enfoque de prueba de comportamiento de los módulos a través de pruebas unitarias no incorpora el desarrollo de stubs para los componentes de servidor con los cuales tienen que interactuar los módulos del cliente, impactando en las ventajas previamente definidas:

- No es posible ejecutar de forma simple pruebas de caja negra a librerías XMPP, probando todas las posibles alternativas que contempla cada protocolo XMPP a través de las interfaces públicas de ese componente. Cada prueba unitaria debería probar un escenario completo del protocolo mediante la validación de su salida correcta a su entrada individual.
- No es posible testear partes de un proyecto sin esperar a que las otras partes estén disponibles, ya que se genera dependencia con el estado de desarrollo de funcionalidad en el servidor. Algunas funcionalidades pueden estar desarrolladas, otras en desarrollo e incluso ni siquiera planteadas, lo cual invalida la posibilidad de realizar pruebas de concepto en los clientes.
- El lograr paralelismo en la ejecución de las pruebas siendo capaces de testear y corregir problemas simultáneamente por diferentes desarrolladores puede verse afectado si las ejecuciones simultáneas no son mutuamente excluyentes en el lado servidor impactando en las respuestas que éste envía al cliente.
- El ser capaces de probar condiciones internas que no son fáciles de generar por entradas externas en grandes sistemas integrados se ve afectado básicamente por el hecho de que puede resultar difícil simular dichas condiciones en el servidor.
- Adicionalmente se genera una gran dependencia con los ciclos de mantenimiento, no disponibilidad y performance por carga de procesamiento del lado servidor, lo que afecta la ejecución de las pruebas unitarias.

2.3 Requerimientos adicionales

De acuerdo a lo planteado surge la necesidad de proporcionar una solución para la ejecución de pruebas unitarias que plantee el desarrollo de algún componente stub que permita obtener las ventajas de la práctica y que cumpla con otros requerimientos adicionales particulares:

- Flexibilidad: se debe estar en condiciones de incorporar nuevas implementaciones del protocolo XMPP, incorporando o modificando los XEPs (XMPP Extension Proposal), sin que implique un costo grande de retrabajo de implementación.
- Modularidad: implica que sea posible ejecutar pruebas sobre implementaciones particulares de XMPP, ya sea un XEP completo, parte de uno o combinaciones de los mismos, sin que implique efectuar una carga completa del soporte del servidor a dicho protocolo.
- Multiplataforma: el cliente Android está desarrollado utilizando el Software development kit Android, que determina que el código sea escrito en lenguaje Java. Esto permite que diversas herramientas de ejecución de pruebas unitarias para el dicho lenguaje puedan ser utilizadas: Jtest, JUnit, JWalk, TestNG, entre otros. Resulta tentador el disponer de estas herramientas sin embargo uno de los objetivos que se persiguen es que la solución planteada pueda ser utilizada en el desarrollo de clientes implementados para diversas plataformas escritas en otros lenguajes: Symbian (Symbian C++), Windows Mobile (Visual C++), IPhone (Objetive C) además de los escritos también en Java (Blackberry y J2ME).
- Proporcionar soporte a transacciones de stanzas: el protocolo XMPP y sus XEPs en determinados casos requieren el intercambio sucesivo de stanzas XML, generándose muchos de los datos intercambiados en tiempo de ejecución, como el Id de las stanzas enviadas por el cliente o el SID generado por el servidor. De esta forma es fundamental ser capaces de gestionar este intercambio dinámico de datos.
- Posibilidad de simular acciones generadas por el servidor: la herramienta no sólo debe permitir que el cliente reciba respuestas ante stanzas enviadas por el módulo cliente. También debe ser capaz de generar acciones que impliquen el envío de stanzas originados en el servidor bajo condiciones controladas (programadas en tiempo) al ejecutar el caso de prueba u otro tipo de acciones no asociadas a stanzas, como el cierre de un socket que genera una desconexión.
- Posibilidad de ejecutar pruebas funcionales con la aplicación cliente desarrollado: es esperable que la herramienta sea capaz de permitir ejecutar pruebas funcionales acotadas con el cliente ya implementado. Esto implica que no sólo debe ser capaz de interactuar con unidades de código sino también debe interactuar con el build de la aplicación generado e instalado en el dispositivo destino. Por ejemplo para una aplicación Android, el archivo apk debe haber sido generado y estar instalado en un dispositivo Android, permitiendo ejecutar la aplicación y validando su funcionamiento.
- Extensibilidad: es deseable que la solución pueda ser extendida para dar soporte a otros protocolos basados en XML como XNMP (XML Based Network Management Protocol over VoIP), XFDL (Extensible Forms Description Language), entre otros.

3 Desarrollo del servidor Stub para la ejecución de pruebas unitarias

En pos de cumplimentar los requerimientos previamente explicados se plantea el desarrollo de un Servidor Stub multiplataforma que sea capaz de ser utilizado para la ejecución de pruebas unitarias en aplicaciones clientes basadas en XMPP. El servidor stub está implementado en el lenguaje de Java utilizando principalmente las siguientes librerías: Parsing SAX, XML Unit, expresiones regulares y networking para Java.

El servidor stub es capaz de:

- Emular un servidor: un módulo cliente puede establecer conexión el stub a través de la apertura de un socket, ya sea durante la ejecución de un caso de prueba unitaria o al conectarse con la aplicación instalada en un cliente móvil.
- Emular implementaciones del protocolo XMPP y sus XEPs: esto implica participar en el intercambio de stanzas XMLs con el cliente, basados en la configuración definida al momento de inicializar el servidor. Cada configuración representa la implementación completa o parcial de diferentes implementaciones del protocolo XMPP y sus XEPs en el lado servidor.
- Generar acciones programadas: el concepto de acción ya ha sido explicado en la sección Requerimientos adicionales. Es posible configurar acciones que serán ejecutadas una vez que la conexión con el módulo cliente ha sido realizada, después de n cantidad de tiempo (en segundos).
- Generar acciones ante la recepción de stanzas enviadas por el módulo cliente: una acción puede ser generada al recibir una stanza particular y por ende no ser programada.
- Gestionar variables locales y globales en el intercambio de stanzas: esto permite ejecutar pruebas de transacciones completas entre el cliente y el servidor, en la que participan múltiples y sucesivos intercambios de stanzas. A través de esta gestión de variables el servidor stub es capaz de enviar stanzas de respuesta cuyos valores de atributo están basados en datos contenidos en las stanzas de requests, mucho de los cuales son generados aleatoriamente en tiempo de ejecución.

La figura 1 proporciona una representación de alto nivel de cómo interactúa el servidor stub al ejecutar un caso de prueba con JUnit.

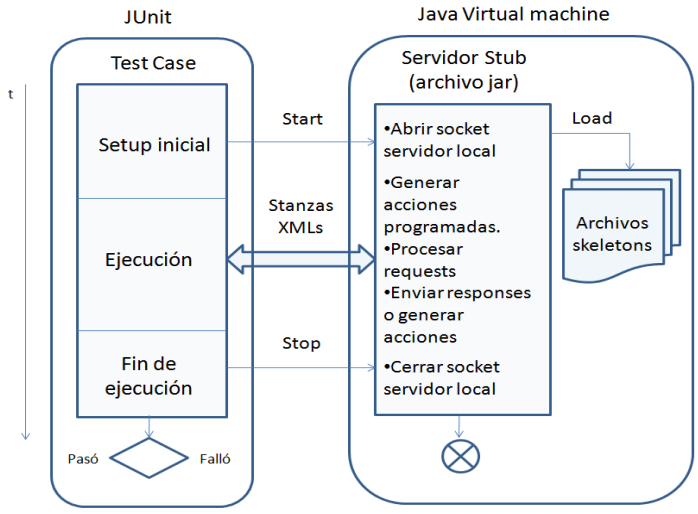

Fig. 1. Interacción entre el módulo cliente y el servidor stub al ejecutar un caso de prueba

El servidor stub está empaquetado en un archivo JAR, que puede ser ejecutado en diferentes plataformas, a través de la máquina virtual Java. De esta forma al iniciar el caso de prueba es necesario ejecutar el proceso que ejecute el servidor stub.

La parte fundamental del servidor está definida en los archivos skeletons. Los archivos skeletons son cargados al momento de realizar el setup inicial del caso de prueba, lo cual indica que diferentes casos de pruebas pueden utilizar diferentes archivos skeletons. Un archivo skeleton indica cómo debe responder el servidor ante las diferentes stanzas XML enviadas durante la ejecución del caso de prueba. De esta forma el servidor al recibir una stanza debe determinar si corresponde enviar una respuesta particular, ejecutar alguna acción o no debe realizar nada.

Adicionalmente los archivos skeletons indican qué acciones programadas deben ejecutarse mientras el caso de prueba se encuentre activo (hasta que éste finalice).

Durante la ejecución del caso de prueba el módulo cliente envía stanzas XML, procesa las respuestas XML enviadas por el servidor (ya sea como respuesta a una stanza o por ejecución de una acción) de acuerdo a su implementación. Una vez que el caso de prueba finaliza a través del mecanismo de Assertion se determina si el caso de prueba fue exitoso o no.

3.1 Configuración de los archivos skeletons

1. **Acciones programadas:** se definen en la cabecera de los archivos de testing utilizando el tag [SCHEDULED-ACTIONS]. En el siguiente ejemplo el servidor stub ejecutará la acción disconnect después de 30 segundos de iniciada la conexión.

```
[SCHEDULED-ACTIONS]
```

```
disconnect, 30
```

2. **Stanzas de respuesta y acciones:** las stanzas enviadas por el módulo cliente que serán procesadas se definen a través de la especificación de request de control ("control request") utilizando el tag [REQ]. Una request de control representa el esqueleto (skeleton) de una stanza. Define la estructura de la stanza que puede ser procesada sin tomar en consideración los valores de sus atributos y el texto, lo cual se especifica a través de las variables. Cuando el servidor stub recibe una stanza enviada por el módulo cliente determina si alguna request de control se corresponde a la misma, y en caso afirmativo éste puede enviar una stanza de respuesta al cliente o ejecutar una acción.

1) **Enviar una stanza de respuesta al cliente:** se utiliza el tag [RESP]. En el siguiente ejemplo el servidor stub envía el hash de la roster cuando el cliente lo solicita.

```
[REQ]
<message type="chat" to="%CJID%"
from="%%JID%%@%%DOMAIN%%/%%RES%%"
xmlns="jabber:client"><body>%BODY%</body></message>
[RESP]
<message type="chat" to="%%JID%%@%%DOMAIN%%/%%RES%%"
from="%%DOMAIN%%"
xmlns="jabber:client"><body>Received:
%BODY%</body></message>
```

Las variables están definidas utilizando el símbolo porcentaje. Una variable almacena un valor enviado por el módulo cliente en alguna stanza. Este valor es utilizado para ser reemplazado en las stanzas de respuesta enviadas por el servidor stub. Éstas variables pueden ser globales o locales:

- **Variable global:** especificada de la forma %%<VARIABLE_NAME>%%. El ámbito de vida es la sesión entre el cliente y el servidor. Cuando una stanza es recibida por el servidor, éste extrae y almacena las variables globales. Una vez que envió la respuesta o ejecutó la acción estas variables permanecen disponibles para ser utilizadas a lo largo de la sesión. Se utilizan especialmente en las stanzas iniciales de autenticación e inicio de sesión. JID, DOMAIN y RES son ejemplos de variables globales.
- **Variable local:** especificada de la forma %<VARIABLE_NAME>%. El ámbito de vida de una variable local es el par stanza recibida/stanza de respuesta. Cuando una stanza es recibida por el servidor, éste extrae y almacena las variables globales. Una vez que envió la respuesta o ejecutó la acción estas variables son removidas. Se utilizan en casi todos los intercambios de stanzas y son de gran utilidad para que la transacción mantenga la secuencia de datos. BODY es un ejemplo de este tipo de variable.

2) **Ejecutar una acción:** se utiliza el tag [ACTION]. En el siguiente ejemplo se ejecutará la acción disconnect cuando el módulo cliente envía un mensaje de chat.

```
[REQ]
<message type="chat" to="%CJID%"
from="%%JID%%@%%DOMAIN%%/%%RES%%"
xmlns="jabber:client"><body>jjj</body></message>
[ACTION] disconnect
```

3.2 Procesamiento de stanzas enviadas por el módulo cliente

Los siguientes pasos son ejecutados para procesar una stanza enviada por el módulo cliente:

1. El modulo cliente envía una stanza al servidor stub.

```
<message type="chat" to="jabber.org"
  from="sender@jabber.org/RES1"
  xmlns="jabber:client"><body>Hi!</body></message>
```

2. El servidor stub verifica si alguna de las request de control se corresponde (matching) con la stanza recibida. Matching significa que su estructura y namespace se correspondan. Se utiliza la API proporcionada por la librería XML Unit para ejecutar estas validaciones [XML-Unit].

```
[REQ]
<message type="chat" to="%CJID%"
from="%%JID%%@%%DOMAIN%%/%%RES%%"
xmlns="jabber:client"><body>%BODY%</body></message>
[RESP]
<message type="chat" to="%%JID%%@%%DOMAIN%%/%%RES%%"
  from="%%DOMAIN%%"
  xmlns="jabber:client"><body>Received:
  %BODY%</body></message>
```

3. Si alguna de las request de control se corresponde el servidor stub envía la stanza de respuesta (definida con el tag [RESP]) reemplazando las variables de acuerdo a los valores recibidos o ejecuta la acción especificada (definida con el tag [ACTION]).

```
<message type="chat" to="sender@jabber.org/RES1"
from="jabber.org" xmlns="jabber:client"><body>
Received: Hi!</body></message>
```

3.3 Un ejemplo de caso de prueba en utilizando JUnit

A continuación se presenta un ejemplo de un caso de prueba planteado sobre JUnit. Sólo es necesario instanciar el servidor stub, definir un directorio que contiene los

archivos skeletons y ejecutar las diferentes acciones para validar los módulos. En este caso de prueba el logueo de un usuario y la ejecución de una llamada.

```
public void testPerformCallAfterSuccessfulLogin() {
      mockServidor.addStanzasPath(userDir + "/res/
      PerformCall/SuccessOutgoingCall");
      mockServidor.start();
          JBCController.getInstance().performLogin("pdp2",
      "12345");
      int result =
      JBCController.getInstance().performCall("pdp1@
      jabber.org/android123");

      assertEquals(Constants.RETURN_CODE_OK, result); }
```

4 Resultados obtenidos

La incorporación de esta herramienta al momento de ejecutar las pruebas unitarias ha permitido enfrentar exitosamente aquellos puntos que se veían afectados al intentar probar las interacciones contra el servidor. Su implementación ha sido probada inicialmente en equipo de desarrollo del cliente Android:

- Permitió testear los módulos que interactúan con el servidor sin depender de su estado actual y disponibilidad.
- Permitió efectuar pruebas sobre condiciones internas difíciles de reproducir. Esto ha sido especialmente útil al momento de verificar las respuestas de los módulos ante desconexiones.
- Permitió ejecutar pruebas en paralelo.

Respecto al cumplimiento de los requerimientos adicionales planteados se presentan los siguientes resultados:

- Flexibilidad: fue posible probar módulos que deben interactuar contra implementaciones de XEPs que aún estaban pendientes de ser completadas en el servidor real.
- Modularidad: el modelo de carga de archivos skeletons ha resultado altamente modular para probar los diferentes módulos de la aplicación.
- Multiplataforma: el servidor stub fue implementado activamente por el equipo Android. Fue probado para los módulos clientes de Blackberry, IPhone y J2ME funcionando correctamente. De todas formas es un punto pendiente lograr un uso más masivo del mismo.
- Soporte a transacciones de stanzas: a través de la gestión de variables fue posible ejecutar pruebas de todas las transacciones sin que se presentaran problemas.
- Posibilidad de simular acciones generadas por el servidor: este punto ha sido ampliamente cubierto y es un factor que fue de gran ayuda al momento de probar ciertas condiciones particulares.

- Posibilidad de ejecutar pruebas funcionales con la aplicación cliente desarrollado: es posible realizar las pruebas funcionales, sin embargo el principal uso del servidor stub está orientado a las pruebas unitarias. Estas pruebas funcionales también funcionaron correctamente para de Blackberry, J2ME y IPhone.
- Extensibilidad: no se han hecho pruebas que permitan determinar que otros protocolos basados en XML, además de XMPP, puedan ser soportados. La arquitectura sobre la cual se construyó el servidor stub debería permitirlo, pero no ha sido validado con ninguna prueba de concepto.

Hay un aspecto importante a considerar que es inherente a la definición de cualquier caso de prueba unitario. Hay que ser altamente cuidadoso al momento de escribir los archivos skeletons, ya que los mismos deben representar apropiadamente la forma en que el protocolo XMPP y sus XEPs se encuentran implementados en el servidor (o la forma en que serán implementados si aún no lo están). Si la definición de estos archivos no es correcta no se puede confiar en los resultados obtenidos al ejecutar los casos de prueba.

5 Conclusiones

La incorporación del servidor stub permitió aprovechar las ventajas que las pruebas unitarias proporcionan en el desarrollo de aplicaciones cliente basadas en XMPP, además de satisfacer los requerimientos adicionales especificados en torno a flexibilidad, modularidad, funcionamiento en múltiples plataformas, generación de acciones por parte del servidor, soporte a transacciones y ejecución de pruebas funcionales. De esta forma fue posible efectuar pruebas basadas en el intercambio de stanzas entre el cliente móvil y un servidor, siendo capaz de simular condiciones y escenarios de ejecución complejos y difíciles de reproducir, favoreciendo la realización de actividades de desarrollo y prueba en paralelo. Queda pendiente el ampliar su utilización a otros equipos de trabajo que realizaron pruebas preliminares satisfactorias, pero sin profundizar su utilización. Igualmente es interesante el hecho de poder validar en un futuro que esta herramienta de pruebas puede ser aplicada para otros protocolos basados en XML.

6 Referencias

1. Saint-Andre, Peter. Smith, Kevin. Tronçon, Remko: XMPP: The Definitive Guide. O'Reilly. Sebastopol (2009)
2. Fowler, Martin. Article: "Mocks aren't Stubs", http://martinfowler.com/articles/mocksArentStubs.html.
3. Rajendran, R.Venkat. Article: "White paper on Unit Testing",http://www.mobilein.com/WhitePaperonUnitTesting.pdf
4. Riley, Tim. Goucher, Adam: Beautiful Testing: Leading Professionals Reveal How They Improve Software. O'Reilly, Sebastopol (2009)
5. Hefczyc, Artur. Article; "Jabber/XMPP servidor testing tool", http://www.tigase.org/node/1169