Interoperabilidad entre .Net y J2EE

Por Diego Quiroga

Contenido

Introducción
La Necesidad es del Negocio
¿Cómo lograrlo?
Mecanismos Orientados a Servicios
CORBA y Remoting, la Misión
Construyendo Puentes
Máxima Velocidad
Conversión de Plataforma y Cross-Compiling
Conclusión: Abriendo la *Caja de Pandora*
Enlaces

Introducción

Más allá de nuestras preferencias personales todos sabemos que existen hoy dos plataformas dominantes para el desarrollo de software: **.Net** y **J2EE**. La tendencia aparente, y mi sensación personal, es que ninguna de ellas se impondrá de forma absoluta sino que, de alguna u otra manera, se repartirán el mercado. En la mayoría de los casos veremos cómo se establecen entornos mixtos donde ambas tecnologías trabajarán en conjunto.

La Necesidad es del Negocio

Más allá de lo estrictamente técnico, existen variadas razones que provocan la existencia de ambientes heterogéneos en las empresas. Casi una mezcla de circo y museo, se conjuga lo último y lo antiguo con distintos sistemas operativos, bases de datos y plataformas de aplicaciones.

Es que al entrar en juego factores tan diversos tales como los cambios de estrategia desde la alta dirección, la selección de nuevos proveedores de software, el **outsourcing** del desarrollo, los recortes presupuestarios, las fusiones entre empresas, crisis y otras cuestiones político/económicas, hacen que inevitablemente las medianas y grandes empresas tengan "un poco de cada cosa".

Veamos algunos escenarios posibles que pueden hacer necesaria la integración entre aplicaciones J2EE y .Net:

- Surge un proyecto nuevo basado en .Net que tiene que reutilizar un set de librerías Java cuya migración es inviable, ya sea por su complejidad, por la inexistencia de documentación o bien porque se dispone únicamente de binarios desarrollados por terceros (que no participan del proyecto, claro).
- El mecanismo estándar definido por las políticas internas de la empresa para comunicarse con otros sistemas propietarios, **SAP**, **AS400** o el **Datawarehouse** está implementado únicamente en una sola de las dos plataformas.
- El directorio de una empresa toma una decisión estratégica que dicta la migración de todos los sistemas a una plataforma única. La casa matriz decide que a partir de ahora "todas las aplicaciones serán construidas en la plataforma X".
- Se decide instalar una misma aplicación para todas las empresas de un mismo grupo y surgen casos donde la plataforma dominante es distinta a la prevista. Se prueban prototipos que una vez aprobados tienen que pasar a producción interactuando con plataformas distintas a la original. Dado que .Net y J2EE tienen fortalezas y debilidades que se complementan, se decide que una solución mixta podría hacer valer las fortalezas de cada una.

Encontraremos que casi la totalidad de las soluciones que existen apuntan a resolver el problema haciendo que la lógica desarrollada en Java/J2EE sea accedida desde C#/.Net y, en muy pocos casos, en sentido contrario. Esto tiene que ver con el hecho de que Java, y en particular J2EE, dominan fuertemente el ambiente corporativo donde se establecieron bastante antes

del surgimiento de Microsoft .Net.

Tengamos en cuenta que la publicación del modelo J2EE impactó fuertemente en este sector al establecer una infraestructura basada en estándares abiertos para aplicaciones distribuidas incluyendo transacciones, seguridad, componentes server-side, acceso a datos, persistencia, clustering y balanceo de carga. El **bonus** adicional del modelo tiene que ver con asegurarse la independencia de un proveedor específico ya que teóricamente un desarrollo particular puede instalarse en cualquier plataforma que respete el estándar (esta “compatibilidad asegurada” se vio afectada negativamente cuando los proveedores empezaron a incorporar mejoras adicionales cubriendo las falencias del estándar). En este sentido, a pesar de que .Net es una excelente opción para el desarrollo de software, como participante tardío de esta carrera todavía tiene camino por recorrer para consolidarse.

¿Cómo lograrlo?

La aproximación más sencilla que podemos implementar es una base de datos de uso compartido, donde a través de **JDBC** y **ADO .Net** nuestras aplicaciones J2EE y .Net, respectivamente, puedan acceder para escribir y leer información. Esta opción tiene mucho sentido si consideramos componentes que en ambos casos se encuentren en la capa de negocios. Para hacer interoperar componentes que se encuentren en diferentes capas, el mecanismo a emplear dependerá del punto de interconexión que estemos considerando (Ver Figura 1).

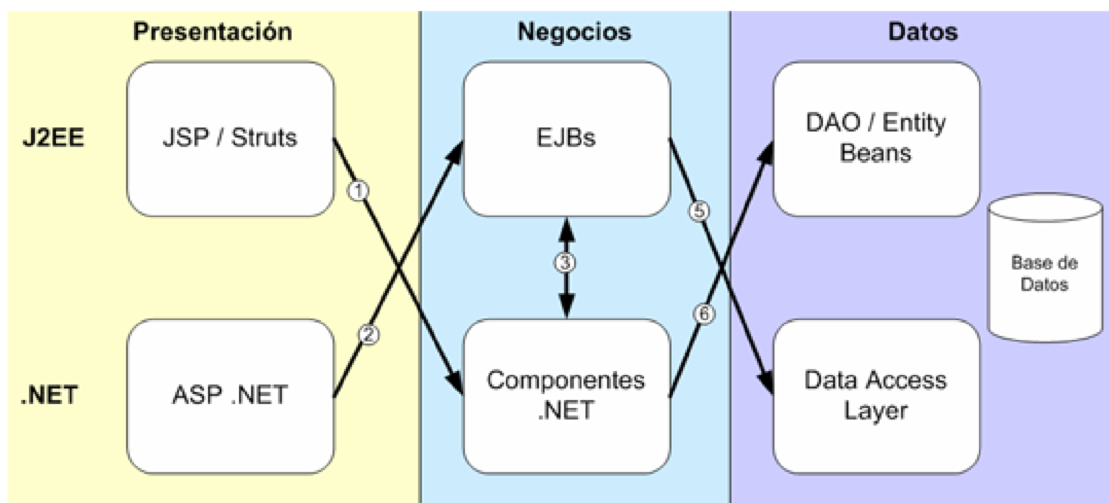

Figura 1: Puntos de interconexión para aplicaciones multi-capa. Volver al texto.

Si el punto de interconexión se extiende a través de Internet o una intranet, la alternativa natural es la utilización de **Web Services**. Tenemos herramientas para generarlos y consumirlos en ambas plataformas y cuentan con la ventaja de ser estándares abiertos que nos permitirán conservar la compatibilidad con futuras aplicaciones. Estaremos limitados, eso sí, por la velocidad del enlace y el costo de emplear XML para transmitir los mensajes.

Si no disponemos de un enlace permanente o el que tenemos es poco confiable, tendremos que considerar servicios de mensajería asíncrona que usualmente emplearemos para la interacción entre las capas de negocios y de datos. Lo que se hace es implementar colas de mensajes en cada una de las plataformas e interconectarlas a través de un **bridge** (cada aplicación produce y consume mensajes a través de la API habitual). Por otra parte, si podemos sacrificar el soporte de transacciones y **callbacks**, es posible generar un **wrapper** que permita el acceso a través de un Web Service.

Cuando lo que estamos buscando es acceder a nivel de clases y métodos, tendremos que optar por un **runtime bridge**. Estos

componentes se agregan en cada plataforma para administrar la interacción y comunicación entre objetos de diferente origen.

Si nos animamos a una solución de más bajo nivel podemos llegar a eliminar el **overhead** de la comunicación remota aprovechando los métodos de interacción con código nativo que tiene cada plataforma.

Por último, si estamos encarando una migración masiva a gran escala, lo que seguramente resulte más conveniente sea emplear herramientas de conversión de plataforma, las cuales se hacen cargo del mapeo de tipos y de proporcionar las APIs correspondientes.

Mecanismos Orientados a Servicios

¿Qué es lo primero que nos viene a la mente cuando nos hablan de integración de aplicaciones? Algo que la maquinaria publicitaria viene machacando desde hace algunos años: ¡Los Web Services, por supuesto! Tenemos la ventaja de que se trata de un mecanismo estándar y que existen en ambas plataformas (aunque lo de “estándar” es algo bastante relativo pues todavía falta un poco para que los proveedores se pongan de acuerdo al respecto).

Repasemos brevemente el funcionamiento de un Web Service como mecanismo de integración en la Figura 2, que se muestra a continuación:

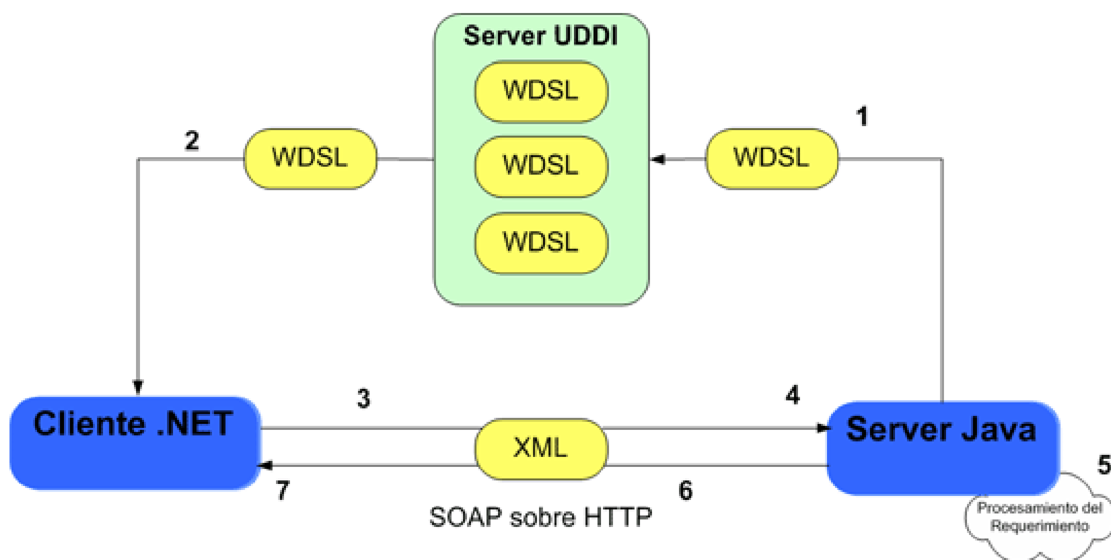

Figura 2: Web Services. Volver al texto.

La aplicación Java, que en este caso expone su funcionalidad a través de un Web Service, registra su descripción (1) a través de un archivo WDSL (**Web Server Description Language**) en un servidor UDDI (**Universal Description, Discovery and Intregration**), que funciona como un DNS para los servicios. La aplicación cliente (en este caso implementada en .Net) consulta (2) al servidor UDDI para poder generar un requerimiento con el formato adecuado. El requerimiento se envía (3) usando el protocolo **SOAP** (que encapsula la información con **XML**) sobre **HTTP**. El servicio recibe el requerimiento (4), lo procesa (5) y devuelve la respuesta (6) nuevamente utilizando SOAP. La aplicación cliente abre el “sobre” SOAP y interpreta el XML obteniendo la información (7) que necesitaba.

Los Web Services son útiles a la hora de integrar componentes a gran escala, dispersos geográficamente, y en donde se realizan unas pocas conexiones por unidad de tiempo. En otros casos, debido a sus limitaciones, pueden llegar a resultar poco recomendables. El overhead que introduce la serialización/deserialización de SOAP (XML) hace que tengamos que pagar un

alto costo en la comunicación y en la conversión.

Tampoco será posible utilizarlos cuando necesitemos soporte nativo de transacciones, hacer callbacks, pasar objetos por referencia o estemos pensando en utilizar de forma extensiva la funcionalidad de una API (sería realmente poco práctico o más bien imposible tener que publicar cada uno de sus métodos). Si nuestra situación se identifica con esto último, sigamos avanzando para descubrir otras soluciones que nos permitan operar a nivel de clase.

CORBA y Remoting, la Misión

Si provienes del mundo Java, seguramente has escuchado acerca de CORBA (**Common Object Request Broker Architecture**), quizás no tanto si provienes de .Net. Dentro de este estándar para la comunicación de componentes distribuidos se emplea el protocolo IIOP (**Internet Inter-ORB Protocol**) para comunicar ciertas entidades definidas como ORB (**Object Request Broker**). Este estándar lleva bastante tiempo en el mercado y podríamos decir que tiene una difusión algo limitada debido a su alta complejidad (este punto genera ásperas discusiones entre sus defensores y detractores en los foros sobre aplicaciones distribuidas).

Dado que en Java se incluye una implementación parcial de CORBA con RMI/IIOP, varias soluciones de interoperabilidad aprovechan la flexibilidad de **Remoting** para armar **channels** y **formatters** específicos que den soporte al protocolo IIOP y a la conversión de tipos entre .NET e IDL (el lenguaje para definir interfases en CORBA). Esta sería la solución ideal para equipos de desarrollo que tengan experiencia usando CORBA entre aplicaciones Java o bien quieran establecer comunicación con otros ORB tales como **mainframes** o aplicaciones **legacy**.

Para empezar a jugar tenemos algunas soluciones **freeware** como **Remoting .CORBA** o **IIOP .Net**, que son muy interesantes. La última parece un poco más práctica porque incluye un generador para crear el archivo IDL, pero es todo cuestión de gustos.

Cualquiera sea el caso, es inevitable adquirir algo de experiencia en la plataforma destino ya que tendremos que meter bastante mano en las clases. La integración necesitará de un poco de **refactoring** tanto en la aplicación cliente como en la aplicación servidor. Podemos pensar en todo caso en soluciones comerciales. La apuesta es reducir el tiempo de desarrollo automatizando este proceso y ocultando la complejidad de la integración al programador. Por ejemplo, **Janeva** (un producto de Borland) se integra en el Visual Studio .Net. Cuando incluimos dentro de nuestro proyecto una referencia a un objeto Java remoto genera un **proxy** al cual podemos acceder en forma directa a través de Remoting, desentendiéndonos del mapeo entre tipos .Net e IDL en que se hace forma automática. (Ver Figura 3):

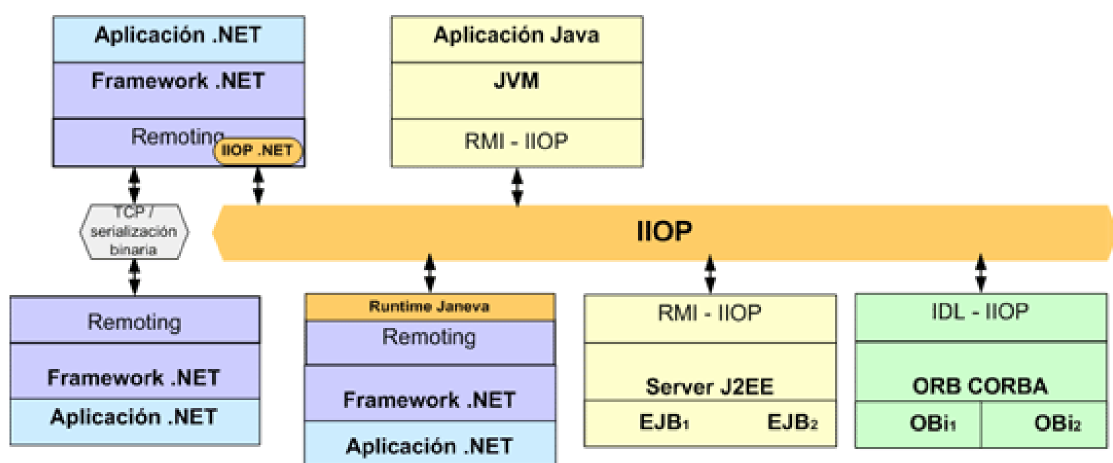

Figura 3: Acceso por CORBA con IIOP .Net y Janeva. Volver al texto.

Al compilar el proyecto, se agrega al **assembly** un **runtime** que se hace cargo de la comunicación entre IIOP y Remoting, manteniendo actualizados los proxies para que "espejen" el comportamiento del objeto destino y su estado.

Construyendo Puentes

Si jamás utilizamos CORBA, IDL nos suena a *aprender chino*; si la cuestión de los ORB nos resulta algo confusa, tenemos otras opciones que utilizan el camino inverso: en lugar de implementar un protocolo Java-compatible, trasladan Remoting a Java. Los productos más representativos son: **Ja.Net** de Intrinsync y **JNBridgePro** de JNBridge. Para hacer el "**bridging**" instalan un runtime de cada lado el cual se encarga de:

- Establecer la comunicación entre las plataformas usando Remoting, principalmente con TCP y serialización binaria, aunque HTTP/SOAP también esté soportado.
- Crear un proxy C# para cada clase Java que se quiere acceder.
- Crear un proxy Java para cada clase .Net que se quiere acceder.
- El mapeo de clases y tipos y la conversión de los valores de retorno en cada caso.
- La invocación de métodos en objetos .Net desde Java.
- La invocación de métodos en objetos Java desde .Net.
- Propagar excepciones entre Java y .Net, y viceversa.
- Brindar soporte de callbacks.

Tal como aparece en el esquema de funcionamiento que se muestra en la Figura 4, soportado por ambas herramientas:

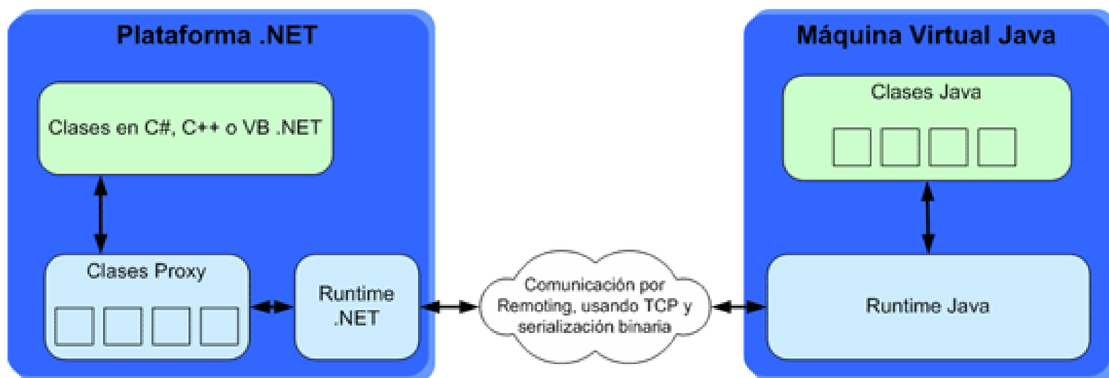

Figura 4: Esquema de un **runtime bridge**. Volver al texto.

Máxima Velocidad

Las alternativas que hemos visto hasta ahora comparten la restricción de tener que publicar obligatoriamente una referencia remota para hacer que el objeto esté disponible en la otra plataforma. Esto necesita de una infraestructura de soporte para la comunicación, control y conversión de tipos.

En la búsqueda de una mayor performance podemos avanzar un paso más y operar directamente a bajo nivel para resolver este problema aprovechando los mecanismos para acceder a código nativo (binario propio del sistema) que proporcionan tanto Java como .Net.

JuggerNet es un producto que aprovecha esta vuelta de tuerca brindando herramientas para facilitar el proceso. Para usarlo, lo primero que tenemos que hacer es seleccionar las clases Java y generar (cuando no) los correspondientes proxies en .Net, para luego poder programar directamente contra estas clases sin inconvenientes o servicios adicionales. Las clases proxy

tienen que comportarse y tener el mismo estado que sus homólogas Java. Para conseguirlo, el runtime C# de JuggerNET emplea JNI (**Java Native Interfase**) para instanciar una JVM (**Java Virtual Machine**) donde se crean los objetos de las clases Java que queremos acceder y manipular. Ahora, como JNI es una API en C no es posible usarla directamente desde C#, sino es a través de una DLL que se accede con Pinvoke (**Platform Invocation Services**), el mecanismo que .Net ofrece para acceder a código nativo escrito como funciones sueltas no orientadas a objetos tales como las DLLs de la API Win32.

Veamos cómo es la secuencia de funcionamiento. El runtime en C# que instala JuggerNet recibe los requerimientos que nuestro código le hace a las clases proxy que se generaron en el primer paso y se los reenvía a la DLL en C que se comunica a través de JNI con la máquina virtual Java que hemos creado para ejecutar los métodos correspondientes o bien leer el estado de algún objeto en particular (Ver Figura 5):

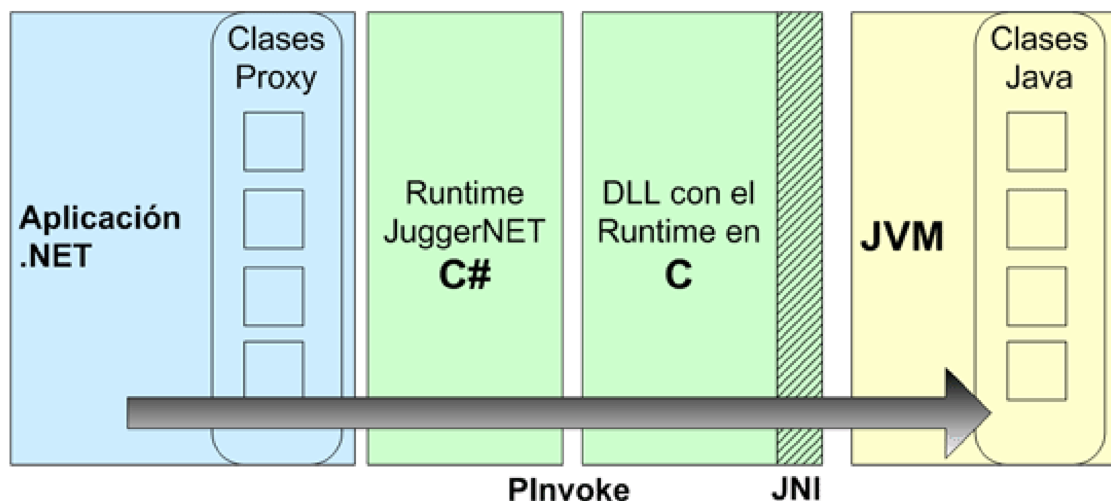

Figura 5: Interconexión C#/pInvoke/JNI/JVM. Volver al texto.

El runtime se hace cargo además de la parte más compleja del asunto, la que tiene que ver con el mapeo de tipos, la propagación de excepciones y los callbacks. El siguiente fragmento de código genera una caja de dialogo Java hecha con Swing para ingresar información en una aplicación .Net:

```
Using System;
Using Java;

Clase Ejemplo
{
public static void Main()
{
string color = JoptionPane.showInputDialog ("Cual es tu color preferido?");
Console.WriteLine("Elegiste: " + color.ToLower());
}
}
```

Como vemos, la integración a nivel código es la más transparente de todas pero exige un nivel de conocimiento mucho más profundo.

Conversión de Plataforma y Cross-Compiling

Finalmente, para el escenario de migración a plataforma única hablaremos del enfoque más drástico de todos, que consiste en realizar una conversión total de nuestras aplicaciones. Nos llevaría muchísimo tiempo, demasiado, revisar y convertir todo nuestro código fuente en forma manual. Peor aún resulta la búsqueda de equivalencias entre tipos y servicios que nos brinda cada una de las APIs de las plataformas.

Conozcamos entonces cómo trabajan dos soluciones comerciales que se hacen cargo de esta tarea: **iNET** de Stryon y **Visual MainWin** de MainSoft.

Ambos productos se integran en Visual Studio haciendo que podamos usar referencias a clases Java, librerías de J2SE o a componentes tales como EJBs. Cuando compilamos, se generan **assemblies** "tradicionales" en MSIL (el "**assembler**" de .NET).

A partir del MSIL, MainWin produce el **bytecode** Java (archivos .class) directamente; en cambio, iNET genera primero el código fuente (archivos .java) y luego lo compila. El objetivo en ambos casos es el mismo: componer algo que pueda correr en una máquina virtual Java (Ver Figura 6):

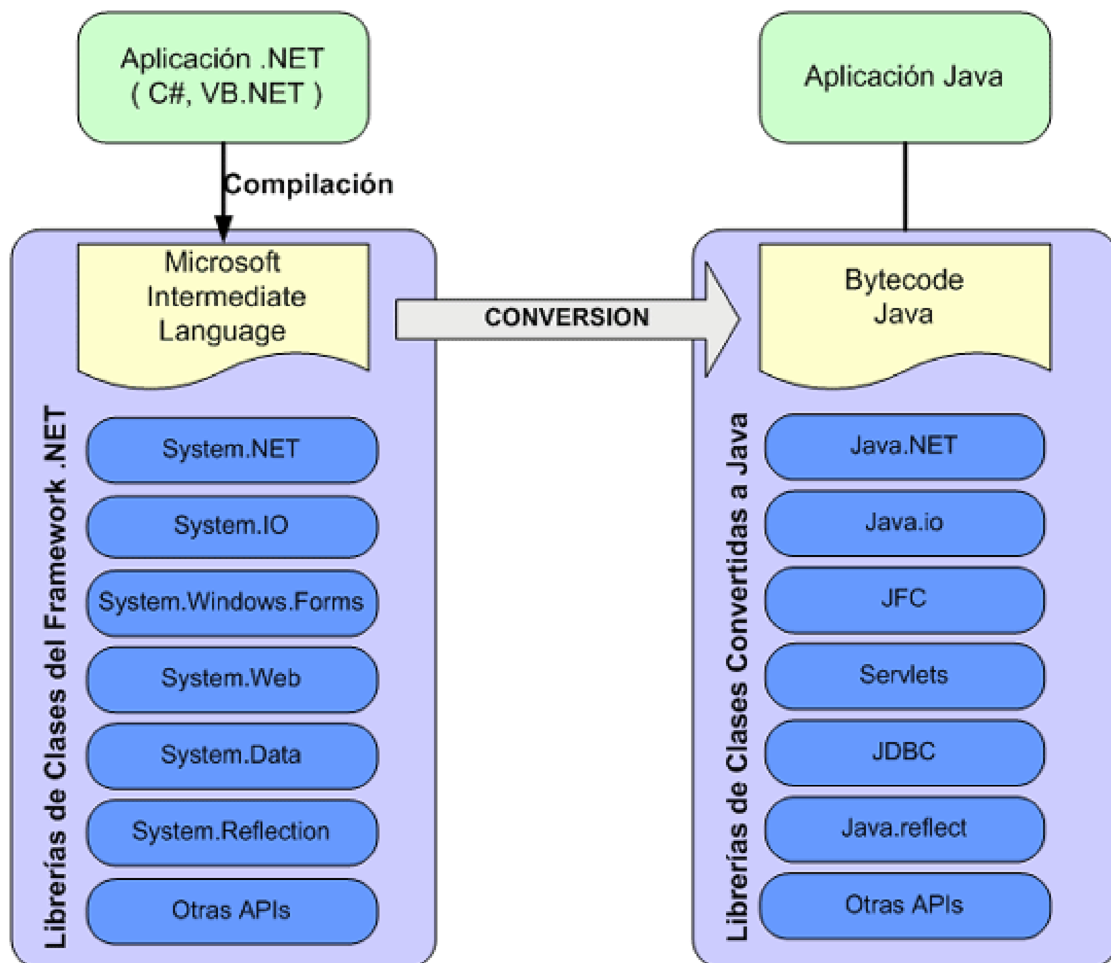

Figura 6: Conversión de Plataforma / APIs. Volver al texto.

En la conversión, estas herramientas aplican un conjunto de reglas para mapear llamadas a API, primitivas y tipos a sus

equivalentes en Java. Para aquellas cosas que no tienen un equivalente, se genera código que lo resuelve aprovechando el **runtime** propietario que es necesario instalar en el server J2EE.

Este runtime además contiene la versión Java de distintas API del .Net Framework, tales como ASP .Net, ADO .Net, WinForms, Remoting, y otras. Cuando finaliza la conversión, se empaqueta la aplicación en un JAR, WAR o en un EAR para su **deployment** en el server J2EE.

Como todo el código se termina ejecutando en la plataforma Java, si la implementación de las API es eficiente, no notaremos diferencia alguna en su funcionamiento respecto de otras aplicaciones que tengamos corriendo.

Sin embargo, el framework .Net es grande, tiene miles de clases y resulta muy complejo (¡Pregunta a los muchachos del proyecto MONO si no lo crees así!). A menos que tengamos totalmente controlado el código que generemos, es muy probable que algunas funciones que usemos no estén totalmente soportadas. El desarrollo en Java por su parte, pierde en portabilidad ya que depende del runtime que contiene la implementación de las API de .Net.

Conclusión: Abriendo la *Caja de Pandora*

La integración de aplicaciones sigue siendo una tarea artesanal; no existe nada que pueda reemplazar la experiencia real programando en cada una de las plataformas. Mi consejo es que llegado el caso, lo más conveniente es en primer lugar evaluar detalladamente la necesidad, luego identificar un par de estrategias factibles y testear cómo se comportan en una prueba de concepto. La aplicación de prueba de concepto debe incluir código que atraviese todas las capas que la conforman (lo que comúnmente se conoce como **slice**) y debe ser testeada con un volumen de carga acorde a su destino real.

Si no es hoy, será mañana, pero seguramente pronto vas a tener la oportunidad de cruzar la frontera que separa ambos mundos, ¡Buena suerte, pionero!

Enlaces

Remoting
JNBridge Pro, www.jnbridge.com/jnbpropg.htm Ja.NET, http://ja.net.intrinsyc.com/
Remoting y IIOP
Janeva, www.borland.com/janeva/ Remoting .CORBA, remoting-corba.sourceforge.net/ IIOP .NET, iiop-net.sourceforge.net/
Conversión de Plataforma
MainWin, www.mainsoft.com/ iNET, www.stryon.com/
Conversión Directa por JNI
JuggerNET, www.codemesh.com/
Interop con .COM
J-Integra Pure Java Runtime, j-integra.intrinsyc.com/
RMI / IIOP
IBM ActiveX Bridge, www.ibm.com/devlopment/
Experimental
OVERONE DotNetJ, www.overone.com/

Diego Quiroga se desempeña como Consultor para empresas tales como Soluziona, Accenture e IBM y ha participado en la construcción de soluciones IT de gran escala en varios países de Latinoamérica y Estados Unidos. Actualmente está a cargo de la Arquitectura Técnica en Movistar, donde trabaja diseñando soluciones tecnológicas y dirigiendo equipos de programadores. Su principal interés reside en el campo de la Ingeniería de Software y en la búsqueda de nuevas y mejores metodologías para el desarrollo de aplicaciones. Es Ingeniero en Sistemas de la Universidad Tecnológica Nacional, donde actuó como docente durante 3 años en la cátedra de Internetworking.