Programación políglota con la máquina virtual Graal

José Antonio Romero Ventura, Ulises Juárez Martínez,
Lisbeth Rodríguez Mazahua, María Antonieta Abud Figueroa,
S. Gustavo Peláez Camarena

Tecnológico Nacional de México,
Instituto Tecnológico de Orizaba,
México

antonioromerov@gmail.com,
{ujuarezm, mabudf}@orizaba.tecnm.mx,
lrodriguez@itorizaba.edu.mx,
gpelaez@ito-depi.edu.mx

Resumen. Hoy en día, el desarrollo de aplicaciones hace uso de entornos separados con el propósito de que los lenguajes de programación usados sean capaces de ejecutarse en ambientes con características específicas, pero esto implica que sea difícil, tardado y costoso el desarrollo y mantenimiento de dichas aplicaciones. Como solución, la programación políglota con GraalVM permite el desarrollo de aplicaciones usando más de un lenguaje de programación a la vez, en un mismo entorno de desarrollo y permitiendo la comunicación entre diferentes lenguajes de programación. En este artículo se emplea la programación políglota en dos escenarios, uno para explorar las capacidades nativas de GraalVM y otro para explorar sus capacidades de interoperabilidad usando Node.js, para observar el potencial de la programación políglota y cómo facilita el desarrollo de aplicaciones al momento de combinar lenguajes de programación que de manera cotidiana se encuentran en entornos separados y se comunican por medio de API's.

Palabras clave: GraalVM, políglota, interoperabilidad, imagen nativa, Node.js.

Polyglot Programming with the Virtual Machine Graal

Abstract. Nowadays, application development makes use of separate environments so that the programming languages used are capable of running in environments with specific characteristics, but this implies that the development and maintenance of said applications is difficult, time-consuming and expensive. As a solution, polyglot programming with GraalVM allows the development of applications using more than one programming language at the same time, in the same development environment and allowing communication between different programming languages. This article uses polyglot programming in two scenarios, one to explore the native capabilities of GraalVM and the other to explore its interoperability capabilities using Node.js. To see the potential of polyglot programming and how it makes it easier to develop applications on the

fly. to combine programming languages that are found in separate environments on a daily basis and communicate through APIs.

Keywords: GraalVM, polyglot, interoperability, native image, Node.js.

1. Introducción

Hoy en día, el desarrollo de aplicaciones se lleva a cabo por medio de diferentes lenguajes de programación [1], ya sea en el desarrollo web [2], aplicaciones de escritorio, aplicaciones de dispositivos móviles o aplicaciones en la nube. Cada uno de los lenguajes requeridos necesita un entorno específico para ejecutarse, ya sea un navegador web, un servidor de aplicaciones, una máquina virtual o un conjunto de intérpretes, así como el uso de entornos separados por sus características de ejecución o compilación. En consecuencia, se tiene el uso de diferentes servidores, aplicaciones en equipos separados debido a ciertas características del programa y el uso de *API's (Application Programming Interface*, Interfaz de programación de aplicaciones*)* que permitan la comunicación entre programas para el envío de mensajes o ejecución de tareas, éstas *API's* en ocasiones son desarrolladas por terceros.

GraalVM [3] *(Graal Virtual Machine*, Máquina Virtual de Graal*)* cuenta con un amplio soporte de lenguajes de programación e interoperabilidad entre estos, así como la ejecución de lenguajes de programación a nivel nativo. *GraalVM* es compatible con sistemas operativos de *Linux, MacOS* y recientemente con *Windows*, aunque la versión de este último aún es versión beta. Estas características permiten ubicar a *GraalVM* como una plataforma universal multilenguaje de alto desempeño para el desarrollo aplicaciones compiladas.

En este artículo se presentan dos escenarios de programación políglota usando *GraalVM*. El primer escenario explora las capacidades nativas y demuestra el potencial de *GraalVM* con diferentes lenguajes de programación; el segundo escenario, con base en un caso de estudio, demuestra las capacidades de interoperabilidad de *GraalVM* usando *Node.js* [4] como *framework*, explorando las capacidades políglotas con diferentes lenguajes de programación, como lo es *Java, Ruby, Python* [5] *y C*. Se analiza y se observa cómo es el desempeño de *GraalVM* en cada uno de los escenarios y cómo es que se comunican los diferentes lenguajes de programación entre sí, lenguajes que comúnmente se encuentran en entornos separados.

El presente artículo está estructurado de la siguiente forma: la sección 2 comprende estado del arte; la sección 3 menciona qué es la programación políglota y presenta algunos ejemplos; la sección 4 habla del desarrollo y cómo funciona la máquina virtual Graal; la sección 5 habla acerca de un caso de estudio y cómo la programación políglota con *GraalVM* ayudó en su mejora; y por último en la sección 6 se mencionan las respectivas conclusiones.

2. Estado del arte

Esta sección presenta los trabajos relacionados a la programación políglota considerando su evolución y aplicación.

En [6], se menciona acerca de los adaptadores que se usan en cuanto al envío de datos entre diferentes lenguajes usando *GraalVM* como herramienta de ejecución de código. Sin embargo, también existen problemas en dichas ejecuciones, un problema común es el paso de datos entre lenguajes. Se implementó un prototipo de adaptadores políglotas usados en *Python 3* y se demostró cómo trabajan en combinación con el *shell* de *GraalVM*.

En [7] *Niephaus et al.* se han enfocado en la interoperabilidad de los lenguajes y el diseño y la implementación de ejecuciones políglotas rápidas. Para ello se utilizó *GraalSqueak*, una implementación de una máquina virtual de *Squeak/SmallTalk* para *GraalVM*. Como conclusión, se tiene que *GraalSqueak* tiene limitaciones en cuanto a la integración de lenguajes que serán investigadas más a futuro. El objetivo principal fue encontrar una manera apropiada de manejar *Squeak/SmallTalk* en el uso de interrupciones y en las grandes cantidades de objetos políglotas.

En [8], *Niephaus* et al. continuaron con la programación políglota, haciendo uso de *bytecodes* con *Truffle*. *Truffle* es un *framework* de implementación de lenguajes, que está diseñado para crear *AST* [9] *(Abstrac Syntax Tree,* Árbol de sintaxis abstracta*)* como intérpretes, el proceso para llevar a cabo la implementación está muy bien documentado. Los *AST* serán los encargados de generar los intérpretes de *bytecodes*, sin embargo, el implementar *bytecodes* en *Truffle* no es algo intuitivo, por lo cual, se requiere la creación de nodos *AST*. Se implementaron todos los *bytecodes* y primitivas necesarias para ejecutar *tinyBenchmarks* de *Squeak* y los resultados de *OpenSmallTalkVM* (*OpenSmallTalk Virtual Machine,* Máquina Virtual de *OpenSmallTalk*) se tratan como base de una máquina virtual para *Squeak/SmallTalk*.

Würthinger et al. [10] describe la construcción de una nueva máquina virtual que aminora el esfuerzo inicial al momento de usar nuevos lenguajes de programación. Comúnmente, las implementaciones de estos nuevos lenguajes se crean en lenguaje *C* o *C++*, que las hace poco seguras, con un grado de complejidad alto, y trabajan comúnmente con interfaces de tipo *bytecode*. El compilador explora la estructura del intérprete y realiza una evaluación parcial del mismo cuando el código se genera. *Würthinger et al.* [10] obtuvieron un alto rendimiento de la combinación de las siguientes técnicas: reescritura de nodos usando *AST*, y des-optimización desde el código máquina de vuelta a los intérpretes *AST*.

Šipek et al. [11] mencionan que la interoperabilidad entre lenguajes de programación puede provocar una baja considerable en el rendimiento del software. Uno de los proyectos que solucionan el problema mencionado y soporta una gran cantidad de lenguajes de programación, así como la interoperabilidad entre estos en la *JVM* (*Java Virtual Machine*, máquina virtual de Java), es el proyecto de Graal *OpenJDK (Open Java Development Kit,* Kit de herramientas de Java *Open)*, que evolucionó del proyecto *Maxine VM* [12] (*Virtual Machine,* máquina virtual).

Salim et al. [13] trabajaron con *WebAssembly* [14], que es un compilador de formato binario para lenguajes como *C/C++*, *Rust* y *Go*. Además, habilita la ejecución en navegadores web y programas de tipo *Standalone* (programa de carácter único, sin dependencias). Los módulos compilados interactúan con otros lenguajes de programación, como *JavaScript*. La compilación *WebAssembly* usa la infraestructura *LLVM* [15] *(Low Level Virtual Machine*, Máquina Virtual de Bajo Nivel*)*, para producir binarios de *WebAssembly* sin hacer uso de una *API* en específico.

Tabla 1. Tabla comparativa de aspectos principales con *GraalVM.*

Artículo	Aspectos			
	Multiplataforma	**Web**	**Múltiple soporte de lenguajes**	**Ambiente GUI**
Niephaus et al. [6]	X		X	
Niephaus et al. [7]	X		X	
Niephaus et al. [8]	X		X	
Würthinger et al. [10]	X		X	
Šipek et al. [11]	X		X	
Salim et al. [13]	X	X	X	
Niephaus et al. [16]	X	X		X
Programación políglota con la máquina virtual Graal	**X**	**X**	**X**	**X**

Niephaus et al. [16] evaluaron *PolyJus*, por medio de una demostración de un ambiente políglota y discutiendo las ventajas y desventajas que se encontraron. Desde que la comunidad científica hace uso de una gran cantidad de lenguajes de programación, especialmente en el análisis de datos y *machine learning* (aprendizaje automático), pueden seleccionar cualquier lenguaje de programación para su uso en los ambientes. Pero esta libertad es un poco limitada, desde que solo un lenguaje se usa por ambiente. Para resolver este problema, hicieron uso del proyecto *Jupyter*, los ambientes *Jupyter* evolucionaron de *IPython*. En un futuro planean crear más ambientes para analizar conjuntos de información más grandes.

En la Tabla 1, se presenta una comparación de características principales entre los artículos del estado del arte y el trabajo del presente artículo.

Como se observa en la Tabla 1, y considerando el estado actual que presenta *GraalVM*, es posible considerar todos los aspectos políglotas para el desarrollo de aplicaciones. Cabe mencionar que, aunque el entorno web ha sido naturalmente políglota, *GraalVM* lo supera al generar código compilado y nativo para cualquier plataforma y combinación de lenguajes utilizados.

3. Programación políglota

La programación políglota es el desarrollo de una solución de software usando más de un lenguaje de programación a la vez en un mismo ambiente de desarrollo. Por ejemplo, en el desarrollo de aplicaciones web, es necesario: el manejo de *HTML* [17] *(HyperText Markup Language*, Lenguaje de Marcas de HiperTexto*)* para la presentación de datos al usuario desde un navegador web; *SQL* [18] (*Structured Query Language*, Lenguaje Estructurado de Consulta) para la obtención y manejo de datos desde un sistema gestor de bases de datos; y por último, por lo menos uno o dos

```
1    const express = require('express');
2    const app = express();
3    app.listen(3000);
4    app.get('/', function(req,res){
5        var text = 'Hello World!';
6        const BigInteger = Java.type('java.math.BigInteger');
7        text += BigInteger.valueOf(2).pow(100).toString(16);
8        text += Polyglot.eval('R','runif(100)')[0];
9        res.send(text);
10   });
```

Fig. 1. Código políglota *Node.js* con *Java y R.*

```
13  try(Context context = Context.newBuilder()
14    .allowAllAccess(true).allowIO(true).build()){
```

Fig. 2. Creación de objeto context.

```
20  for(String arg : args){ texto += arg + " ";}
21      texto += "- JAVA";
```

Fig. 3. Almacenamiento de la cadena de texto en una variable de *Java*.

```
24  codigoRuby += "out_file = File.new('" + filename + "', 'w+') \n";
25  codigoRuby += "out_file.puts('"+ texto + " - RUBY') \n";
26  codigoRuby += "out_file.close \n";
27  context.eval("ruby", codigoRuby);
```

Fig. 4. Creación del archivo en *Ruby*.

lenguajes de *scripting* para el procesamiento de datos desde el servidor. Todos estos componentes combinados en un solo ambiente, sin el uso de software de terceros o alguna *API* intermedia que logre las comunicaciones entre estos, hace que la programación políglota se aplique en una misma solución de software. Desde el punto de vista metodológico, la programación políglota hace uso de las metodologías convencionales (pesadas y ágiles) de desarrollo de software debido a que los lenguajes que se utilizan son orientados a objetos, independientemente de la parte híbrida que actualmente se tiene con enfoques funcionales.

En los siguientes ejemplos se presenta cómo se realiza la programación políglota por medio de la plataforma *GraalVM*, usando diferentes lenguajes de programación y combinándolos en un solo código. En la Fig. 1 se observa un ejemplo de código políglota combinando *JavaScript* [19] potenciado por *Node.js* como lenguaje base, con *Java* [20] y *R* como lenguaje de uso interno o complementario. Como se observa en la Fig. 1, en la línea 5 se tiene el concatenado de un texto para mostrar directamente en *JavaScript*, en la línea 6 se observa el manejo de clases de Java, en este caso el manejo de "*java.math.BigInteger*" que crea un numero a partir de dicha clase, en la línea 8 se genera un número aleatoriamente usando lenguaje *R.* Todos estos resultados se concatenan en una cadena de texto que se mostrará en pantalla desde un navegador *web* por medio de un *request GET* de *Node.js* desde el servidor local en el puerto 3000.

A continuación, se presenta un ejemplo políglota de un programa con *Java* como lenguaje base (cabe mencionar que se pueden tener los demás lenguajes soportados por *GraalVM* como lenguajes base por igual) que obtiene una cadena de texto desde una terminal. Posteriormente hace la creación de un archivo escribiendo la cadena de texto introducida desde la terminal en lenguaje *Ruby*, después hace la lectura de dicho archivo

en lenguaje *Python* y por último hace la impresión en pantalla del contenido del archivo en lenguaje *C*. El código es más complejo ya que se tiene la combinación de cuatro lenguajes de programación en un mismo programa, debido a esto es un programa más extenso, por lo tanto, solo se mostrarán los bloques principales donde se presenta la combinación de lenguajes y la sección de comunicación entre estos.

En la Fig. 2 se observa la creación del objeto "*context*", este objeto permite la creación del ambiente políglota de *GraalVM* donde se ejecutan los diferentes lenguajes de programación. En la línea 14 se especifican los permisos generales a *GraalVM*, y el permiso a la creación, lectura y escritura de archivos dentro del equipo de cómputo. En la fig. 3 se realiza la obtención de la cadena de texto que se introdujo desde una terminal y se almacena en una variable de *Java*, en el concatenado de la línea 21 es solamente para mostrar el nombre del lenguaje que lo está ejecutando, solo para cuestiones de flujo de código y ver cómo la cadena de texto va pasando por los diferentes lenguajes de programación.

En la Fig. 4 se observa la creación del archivo en lenguaje *Ruby*, se genera una cadena de texto con el script a ejecutar de *Ruby* y se almacena en una variable en *Java*, después en la línea 27 se ejecuta con el objeto "*context*" usando la función "*eval()*", se especifica primero el lenguaje de programación y después el *script* o líneas de código a ejecutar.

En la Fig. 5 se realiza la ejecución de la lectura del archivo. Este bloque es muy parecido al código de la Fig. 4 ya que se concatena una cadena de texto con el código a ejecutar, en este caso es en lenguaje *Python.* Por último, en la línea 35 se realiza la ejecución de dicho código con el objeto "*context*".

Por último, en la Fig. 6 se realiza la impresión en pantalla de la cadena de texto introducida desde la terminal junto con los nombres de los lenguajes en los cuales se ejecutó. En *GraalVM*, el lenguaje *C* [21] también es compilado, por lo tanto se tienen acceso a este como un programa objeto y se ejecuta desde un objeto "*context*", posteriormente se almacena en un objeto de tipo "*Value*", como se observa en la línea 39. Con esto se tiene acceso a las funciones internas del programa generadas por el desarrollador, incluso se tiene acceso a funciones propias del lenguaje *C*. En la línea 40 se ejecuta "*printMensajeArray*", que es una función desarrollada en *C* que imprime en pantalla una cadena de texto que se obtiene como parámetro de función. La línea 42 hace la ejecución final de la impresión en pantalla desde *C*.

En el siguiente ejemplo se presenta lo que es la implementación del programa anterior, pero desde una imagen nativa, el código es el mismo, solo varia una sección que se explicará a continuación. En la fig. 7 se observa lo que es la declaración de las propiedades "*option*" en el objeto "*context*", estas opciones se declaran para permitir a la imagen nativa localizar las ubicaciones de los lenguajes a usar en ella, por ello se observan palabras como "*ruby*" o "*python*" en las líneas 15 a 20, que son los lenguajes de programación que se usaron en dicha prueba.

Al ejecutar la imagen nativa, esta se ejecuta como un objeto *bash* de *Linux*, en este ejemplo se pasa como parámetro de ejecución el valor "*-Dllvm.home*", este parámetro es necesario ya que en el programa de la imagen nativa se ejecuta código en lenguaje *C* usando *LLVM*, esta "*option*" se envía de esta manera debido a la versión de *GraalVM* que se está manejando, la versión 20.1.

Para efectos de rendimiento se hizo un conjunto de pruebas de desempeño de programas, se realizó la ejecución del programa de creación y lectura de archivos

```
30  codigoPython += "cadena = '' \n";
31  codigoPython += "for line in open('" + filename + "'): \n";
32  codigoPython += "       cadena = cadena + line \n\n";
33  codigoPython += "cadena.strip()+' - PYTHON'";
34  Value filecontent = context.eval("python", codigoPython);
35  String textodesdePython = filecontent.asString();
```

Fig. 5. Lectura del archivo en *Python*.

```
38  Source s = Source.newBuilder("llvm", new File("imprimeTexto.o")).build();
39  Value lib = context.eval(s);
40  Value printMensajeArray = lib.getMember("printMensajeArray");
41  Value msgarray = getvaluyeArrayCharASCII(textodesdePython,context);
42  printMensajeArray.executeVoid(msgarray); //Imprime en pantalla desde C
```

Fig. 6. Impresión en pantalla desde *C*.

```
13  try(Context context = Context.newBuilder().allowNativeAccess(true)
14  .allowHostAccess(HostAccess.ALL).allowAllAccess(true).allowIO(true)
15  .option("ruby.home","GRAALVM/jre/languages/ruby")
16  .option("python.SysPrefix","GRAALVM/jre/languages/python")
17  .option("python.CoreHome","GRAALVM/jre/languages/python/lib-graalpython")
18  .option("python.StdLibHome","GRAALVM/jre/languages/python/lib-python/3")
19  .option("python.Executable","GRAALVM/jre/languages/python/bin/graalpython")
20  .option("python.CAPI","GRAALVM/jre/languages/python/lib-graalpython")
```

Fig. 7. Declaración de valores "*option*" en context.

explicados anteriormente, con y sin imagen nativa; ambas versiones se ejecutaron un total de 100 veces en un equipo *Lenovo Legion Y530*, con procesador *Core i5 8300H* a 60 *Hz* y 16 *GB* de *RAM* en Ubuntu 18.04 64 *bits*. Los resultados fueron los siguientes:

Como se observa en la Tabla 2, los resultados oscilan de 5 a 7 incluso 9 segundos, dando un promedio de 5.840 segundos de tiempo de ejecución. En la Tabla 3 se observa lo que son los resultados en segundos del tiempo de ejecución del programa, pero desde una imagen nativa, por lo que se observa un tiempo que oscila de los 0.424 segundos a los 1.529 segundos, dando un promedio de todos los datos de 0.43997 segundos de tiempo de ejecución, por lo que se aprecia el gran potencial que tiene la imagen nativa sobre el programa de Java.

4. Componentes de la programación políglota

En la Fig. 8, se presenta un diagrama que muestra los componentes principales que interactúan con *GraalVM*, algunos componentes internos y los productos que se obtienen de procesar códigos políglotas. Como se observa en la Fig. 8, GraalVM funciona con los siguientes lenguajes de programación: *Java*, *JavaScript*, *Python*, *Ruby*, *R*, *C/C++*, *WebAssembly (Wasm)*. Dichos lenguajes son interpretados o compilados según sea su naturaleza por *GraalVM (*en el caso de *C* y *C++* que son lenguajes compilados). Sus componentes principales que llevan a cabo dicha tarea son: (1) los *AST*, estos permiten la interpretación de los diferentes lenguajes de programación por medio de árboles de sintaxis abstracta; (2) *Truffle* [22] es un *framework* de *GraalVM* que permite implementar estos lenguajes de programación con ayuda de los *AST*; (3) el compilador *JIT (Just In Time*, Justo en tiempo*)*, este componente permite la ejecución o interpretación de los programas políglotas en

Tabla 2. Resultados del tiempo de ejecución del programa sin imagen nativa.

5.699	5.563	5.791	5.565	5.899	5.681	5.666	5.766	5.616	5.573
5.897	5.700	5.593	5.770	5.717	5.843	5.770	5.762	5.767	5.802
5.766	5.876	5.761	5.554	5.971	5.670	5.943	5.834	5.671	5.752
5.749	5.962	5.605	5.651	5.607	7.302	5.734	5.774	5.870	5.733
9.375	5.589	5.855	5.790	5.641	5.624	6.064	5.845	5.967	5.709
5.762	5.623	5.830	5.811	5.697	5.902	5.735	5.745	6.376	5.880
7.522	5.743	5.665	5.722	5.879	5.616	5.872	5.607	6.665	5.631
7.029	5.634	5.675	5.753	5.771	5.940	5.745	5.966	5.952	5.715
5.803	5.649	5.748	5.676	5.773	5.745	5.683	5.684	5.696	5.574
5.674	5.671	5.977	5.618	5.581	5.770	5.560	5.704	5.726	5.585

Tabla 3. Resultados del tiempo de ejecución del programa con imagen nativa.

1.529	0.428	0.427	0.425	0.427	0.427	0.435	0.426	0.433	0.433
0.433	0.426	0.437	0.427	0.429	0.428	0.425	0.431	0.427	0.426
0.427	0.426	0.431	0.426	0.426	0.432	0.428	0.433	0.429	0.438
0.432	0.432	0.433	0.425	0.427	0.425	0.427	0.430	0.426	0.432
0.430	0.429	0.429	0.429	0.426	0.438	0.427	0.427	0.430	0.434
0.426	0.437	0.427	0.426	0.433	0.429	0.438	0.427	0.430	0.428
0.426	0.424	0.432	0.430	0.432	0.425	0.429	0.432	0.427	0.433
0.435	0.426	0.432	0.426	0.426	0.435	0.427	0.425	0.426	0.425
0.426	0.426	0.425	0.426	0.426	0.431	0.430	0.432	0.428	0.430
0.428	0.425	0.433	0.425	0.431	0.433	0.427	0.426	0.426	0.426

tiempo de ejecución permitiendo detectar los puntos de fuga en las excepciones o manejo de errores de sintaxis que le permiten al usuario corregir sus códigos o encontrar las excepciones de manera más rápida entre los enlaces de los lenguajes políglotas; por último, (4) el compilador *AOT (Ahead Of Time,* Antes de tiempo*)*, esta propiedad es principalmente aplicable a las imágenes nativas generadas por *GraalVM*, ya que gracias a ello la ejecución de estas son más rápidas, eficientes, y mantienen su interoperabilidad de lenguajes.

5. Caso de estudio

El caso de estudio consiste en una aplicación web de una empresa de desarrollo de software en la ciudad de Monterrey, Nuevo León, México, y debido a derechos de autor y temas de confidencialidad, solo se mostrarán diagramas representativos del sistema estudiado de la compañía.

En dicho sistema se tiene la separación de lenguajes de programación debido a que un lenguaje es para desarrollo propio del entorno web, como lo es *HTML, CSS* [23]

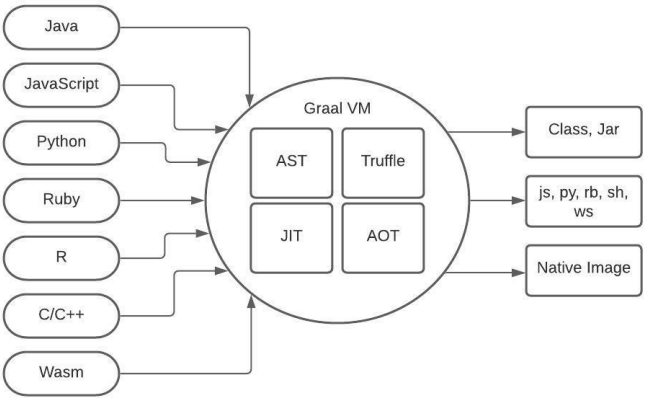

Fig. 8. Diagrama que muestra los componentes dentro de *GraalVM.*

Fig. 9. Diagrama de Arquitectura SOA en sistema del caso de estudio.

(*Cascading Style Sheets*, Hojas de Estilo en Cascada), *JavaScript*; los lenguajes de *scripting*, que en este caso es *ASP.NET (Active Server Pages .NET,* páginas activas del servidor .NET*)* y *C#* como lenguaje de programación; y el lenguaje *SQL*. Cada uno de estos ambientes se encuentran en entornos separados, aplicando procesos robustos de la ingeniería de software como lo es la separación de intereses para atender los requerimientos no funcionales.

Como se observa en la Fig. 9, se aplica la arquitectura *SOA (Service Oriented Architecture,* Arquitectura Orientada a Servicios*)*, esto permite que la aplicación se consulte desde canales *web* y dispositivos móviles. En dicho diagrama se observa de forma general la separación de intereses por medio de una capa de "Lógica de negocio" que realiza todas las operaciones y algoritmos referentes a las funciones de la compañía; en la capa de "Acceso a datos" es propiamente para el manejo de consultas a bases de datos para su posterior procesamiento en la capa de "Lógica de negocio"; la capa de "Cliente sistemas externos" se usa para la obtención de datos desde un sistema externo; se tiene como agentes externos lo que es la "base de datos" y el "sistema externo" que interactúan con los servicios por medio de *API* o software de terceros; por último la implementación es una solución de servicios SOAP/REST (*Representational State Transfer*, transferencia de estado representacional).

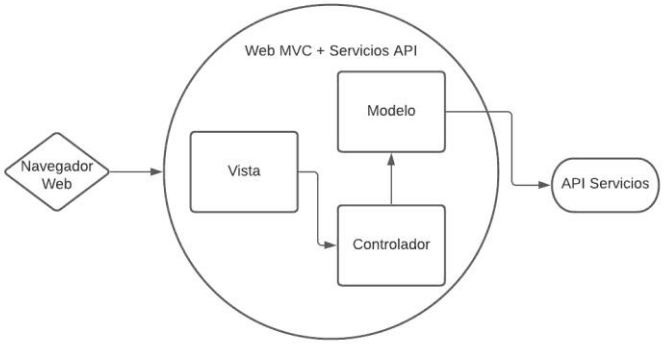

Fig. 10. Diagrama de aplicación del modelo MVC con consumo de una API de servicios.

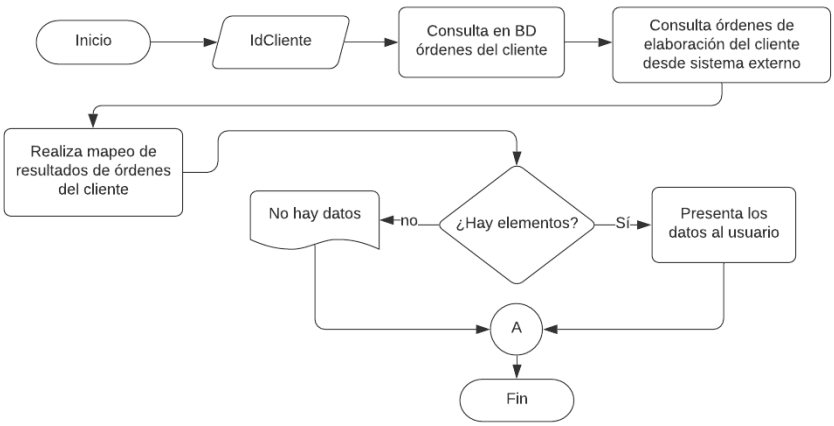

Fig. 11. Diagrama de flujo de programa políglota con base en el caso de estudio.

Tal y como se observa en la Fig.10, por medio del canal web, se maneja el modelo *MVC (Model View Controller,* Modelo Vista Controlador*)* junto con el consumo de una *API* pare el consumo de servicios explicado en la Fig. 9, también se tiene la vista que se presenta desde un navegador web al usuario, el controlador que realiza la administración de llamadas y obtención de datos desde diferentes acciones del usuario y el modelo que lleva a cabo el procesamiento de datos y obtención de los mismos desde el uso de una *API* de servicios. Como se aprecia, la programación políglota fortalece la separación en capas requeridas por el patrón arquitectónico MVC, al homogenizar en un solo estilo de programación el trabajo con varios lenguajes.

Cada capa representa a un conjunto de asuntos en específico, incluyendo los intereses como capacidad de mantenimiento, de documentación y en este caso de implementación homogénea. La programación políglota facilita el desarrollo de sistemas complejos al permitir un lenguaje base para interactuar con los otros lenguajes requeridos.

También se hizo un rediseño e implementación de cómo sería su funcionamiento si este se aplicara por medio de un entorno políglota con *GraalVM*, por ello, se realizó el

```
70  function getWorkOrdersByClientFromExternalSystem(idclient) {
71      return new Promise(resolve => {
72          var WorkOrderJava = Java.type('WorkOrder');
73          var list = WorkOrderJava.GetWorkOrdersByClientArrayObj(idclient);
74          var jsonordersfromextsys = JSON.stringify(list);
75          resolve(jsonordersfromextsys);
76  }); }
```

Fig. 12. Consulta ordenes de elaboración del cliente desde sistema externo.

```
80  function getOrdersOnProcessOrFinished(orders, workorders) {
81  return new Promise(resolve => {
82    resolve(Polyglot.eval('python', "set("+orders+") & set("+workorders+")"));
83  });  }
```

Fig. 13. Comparación de conjuntos de ordenes en *Python*.

```
20  app.get('/', function(req, res) {
21      var text = 'Welcome!<br> '
22      text += "Client = " + clientId + '<br>'
23      if (ordersonprocessorfinished.length <= 0) {
24          text += span("No orders in process".red) + " <br>"
25      } else {
26          text += span("Orders in process".green) + " <br><ul>"
27          for (var i = 0; i < ordersonprocessorfinished.length; i++) {
28              text += "<li> " + ordersonprocessorfinished[i] + " </li>"
29          }
30          text += "</ul>"
31      }
32      res.send(text)
33  })
```

Fig. 14. Presenta en pantalla resultados por medio de un *request GET* de *Node.js.*

diagrama de la Fig. 11. En la Fig. 11 se tiene el diagrama de flujo del funcionamiento del programa políglota: se introduce el identificador del cliente "IdCliente", con este valor se busca en la base de datos las órdenes o pedidos que ha realizado el cliente, después se consulta desde un sistema externo por medio de una clase en Java, se obtienen los datos de órdenes de elaboración del cliente, sin hacer uso de alguna API o software de terceros. Después se realiza un mapeo de dichas órdenes para ver si hay una intersección de ambos conjuntos de elementos donde el conjunto A serían las órdenes en la base de datos y el conjunto B serían las órdenes de elaboración desde el sistema externo obteniendo $C = A \cap B$ donde C es el conjunto de órdenes en proceso de elaboración pertenecientes al id del cliente. Después solo se analiza si el conjunto C tiene elementos, si no, presenta en pantalla un mensaje de "No hay datos", y si los hay, presenta la lista de las órdenes de elaboración.

El código se realizó en *JavaScript* con *Node.js* [24], y debido a que es un código largo se explicarán los bloques principales siguiendo la lógica del diagrama de la Fig. 11. En la Fig. 12 se observa el bloque de código que realiza la ejecución de un programa en *Java*, obteniendo las órdenes de elaboración desde un sistema externo. Anteriormente se consultaban las ordenes en proceso desde un sistema externo consumiendo una API/REST, sin embargo, ahora se usa la clase de Java "WorkOrder" desde Node.js, por medio de propiedad políglota "Java.type()" como se ve en la línea 72. Después se obtiene el listado de órdenes por el identificador del cliente ejecutando la función "GetWorkOrdersByClientArrayObj" en la línea 73, por último, se hace un

formateo a *JSON (JavaScript Object Notation*, Notación de Objetos de JavaScript*)* en la línea 74.

En la Fig. 13, el código realiza la comparativa de conjuntos de órdenes y órdenes de elaboración para obtener la intersección de ambos conjuntos y presentarlos al usuario, esta comparativa se realiza en lenguaje Python. Se ejecuta el elemento políglota "*Polyglot.eval()*" en la línea 82, donde se especifica como primer parámetro el lenguaje de programación a ejecutar seguido de la línea de código, en este caso se especifica el lenguaje *Python*.

En la fig. 14, se encuentra el código que representa la lógica del diagrama de la Fig. 11 donde hace la condición si el conjunto de órdenes de elaboración en proceso tiene o no elementos, si no los tiene presenta un mensaje en color rojo como se ve en la línea 26, y si los tiene presenta un mensaje en color verde como se ve en la línea 29, seguido de las órdenes en proceso.

El programa con Node.js funcionó de acuerdo con el funcionamiento esperado como se manejaba en el sistema de .NET. La gran diferencia es que se realizó una disminución de capas de desarrollo debido al diferente manejo de lenguajes y sistemas externos. Dicho cambio a código políglota permitió una ejecución más transparente y rápida. La parte políglota del programa permite de manera homogénea trabajar con menos capas, ya sea de manera compilada o como una imagen nativa. En general el caso de estudio presenta un considerable cambio de estructura del programa original debido a la disminución de capas.

6. Conclusiones

Con base en los resultados y comparaciones realizadas, la programación políglota con *GraalVM* muestra un amplio soporte de lenguajes e interoperabilidad entre los mismos, las capacidades nativas favorecen el desempeño de las aplicaciones al tener archivos ejecutables compilados y no se requiere de metodologías nuevas para soportar el desarrollo de aplicaciones políglotas. Desde el punto de vista arquitectónico, el modelo MVC, así como los estilos de programación, no se ven afectados por el contexto políglota, solo se favorece la implementación al usar cada parte de la solución en el lenguaje que mejores propiedades ofrece. También es remarcable mencionar la disminución de código observado en el caso de estudio, así como la simplificación de las capas requeridas en su arquitectura. En el mismo sentido, la comunicación entre componentes se mejora al tener de forma transparente un solo medio de programación.

Adicionalmemte, a través del framework Truffle, es posible incorporar nuevos lenguajes de programación que actualmente no se soportan en *GraalVM*, lo cual trae como consecuencia mejorar las capacidades de implementar sistemas que requieran necesidades muy específicas de diversas áreas, por ejemplo, la programación lógica para favorecer implementaciones de inteligencia artificial.

Como trabajo futuro se tiene considerado hacer pruebas con casos de mayor complejidad, evaluar de forma exhaustiva el desempeño nativo y no nativo de *GraalVM*, incorporar al menos un lenguaje no soportado actualmente en *GraalVM* para evaluar la facilidad de extensibilidad nata de la plataforma, y finalmente, se considera utilizar la naturaleza políglota de *GraalVM* para reimplementar aplicaciones que trabajan en área transversales como es el caso de la bioinformática.

Referencias

1. Juganaru-Mathieu, M.: Introducción a la programación. Grupo Editorial Patria (2014)
2. Mateu, C.: Desarrollo de aplicaciones web. Eureca Media, SL, pp. 39−43 (2004)
3. Oracle and/or its affiliates: GraalVM. https://graalvm.org/ (2020)
4. Nodejs: Acerca de Node.js. https://nodejs.org/es/about/ (2020)
5. Wiki.python: Beginner's Guide to Python. https://wiki.python.org/moin/ BeginnersGuide (2020)
6. Niephaus, F., Felgentreff, T., Hirschfeld, R.: Towards polyglot adapters for the GraalVM. In: Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming, pp. 1–3 (2019)
7. Niephaus, F., Felgentreff, T., Hirschfeld, R.: GraalSqueak: Toward a smalltalk-based tooling platform for polyglot programming. In: Proceedings of the 16th ACM (SIGPLAN) International Conference on Managed Programming Languages and Runtimes, pp. 14–26 (2019)
8. Niephaus, F., Felgentreff, T., Hirschfeld, R.: GraalSqueak: A fast smalltalk bytecode interpreter written in an AST interpreter framework. In: Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, pp. 30–35 (2018)
9. Eclipse: AST. https://eclipse.org/jdt/ui/astview/index.php (2020)
10. Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: Wolczko One VM to rule them all. In: Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, pp. 187–204 (2013)
11. Šipek, M., Mihaljević, B., Radovan, A.: Exploring aspects of polyglot high-performance virtual machine graalVM. In: International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 1671-1676 (2019)
12. Maxine-VM: Welcome to the Maxine VM project. https://maxine-vm.readthedocs.io/en/stable/ (2020)
13. Salim, S.S., Nisbet, A., Luján, M.: Towards a WebAssembly standalone runtime on GraalVM. In: Proceedings Companion of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, Athens, Greece, pp. 15–16 (2019)
14. Webassembly.org: https://webassembly.org/ (2020)
15. Llvm.org: The LLVM Compiler Infrastructure. https://llvm.org/ (2020)
16. Niephaus, F., Krebs, E., Flach, C., Lincke, J., Hirschfeld, R.: PolyJuS: A Squeak/Smalltalk-based polyglot notebook system for the GraalVM. In: Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming, pp. 1–6 (2019)
17. Mozilla.org: HTML. https://developer.mozilla.org/es/docs/Web/HTML (2020)
18. Oppel, A., Sheldon, R.: Fundamentos de SQL. McGraw-Hill Interamericana Editores (2009)
19. Mozilla.org: JavaScript. https://developer.mozilla.org/es/docs/Web/JavaScript (2020)
20. Java: Java. https://java.com/en/download/faq/whatis_java.xml (2020)
21. Maas, A.J., Nazaré, H., Liblit, B.: Array length inference for C library bindings. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 461–471 (2016)
22. Github: Truffle. https://github.com/oracle/graal/tree/master/truffle (2020)
23. Mozilla.org: CSS. https://developer.mozilla.org/es/docs/Web/CSS (2020)
24. Sun, H., Bonetta, D., Humer, C., Binder, W.: Efficient dynamic analysis for Node.js. In: Proceedings of the 27th International Conference on Compiler Construction, pp. 196–206 (2018)