UNA NUEVA ARQUITECTURA DE SIMULACIÓN DISTRIBUIDA DIRIGIDA POR EVENTOS

A NEW EVENT-DRIVEN DISTRIBUTED SIMULATION ARCHITECTURE

TRABAJO FIN DE MÁSTER
CURSO 2019-2020

AUTOR
LUIS FERNANDO ALMENDRAS ARUZAMEN

DIRECTORES
JOSÉ LUIS RISCO MARTÍN
KATZALIN OLCOZ HERRERO

MÁSTER EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

UNA NUEVA ARQUITECTURA DE SIMULACIÓN DISTRIBUIDA DIRIGIDA POR EVENTOS

A NEW EVENT-DRIVEN DISTRIBUTED SIMULATION ARCHITECTURE

TRABAJO DE FIN DE MÁSTER EN INGENIERÍA INFORMÁTICA
DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA

AUTOR
LUIS FERNANDO ALMENDRAS ARUZAMEN

DIRECTORES
JOSÉ LUIS RISCO MARTÍN
KATZALIN OLCOZ HERRERO

CONVOCATORIA: JUNIO/JULIO 2020
CALIFICACIÓN: 9.5

MÁSTER EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

02 DE JULIO DE 2020

DEDICATORIA

A Hilda, Fernando y Noelia a quienes

les hice la promesa de regalarles la

dedicatoria de este trabajo.

AGRADECIMIENTOS

A mi esposa y mis hijos, por haberme apoyado en la culminación de este trabajo a pesar de haberles quitado el tiempo valioso de esposo y padre.

A los directores y personas que participaron en la revisión y consolidación de este trabajo, por su tiempo y sugerencias valiosas.

RESUMEN

Los computadores actuales son sistemas Multi-Core, en los cuales cada procesador contiene varios núcleos de ejecución. Estos sistemas de memoria compartida pueden utilizarse de manera individual o combinarse para formar un supercomputador de memoria distribuida. Aprovechar estos dos niveles de paralelismo en estas máquinas es importante para conseguir explotar al máximo su capacidad de cálculo, lo que constituye un área de investigación de gran interés en la actualidad.

Por otro lado, el paradigma de la computación en la nube ofrece a los usuarios un conjunto casi ilimitado de recursos en un modelo de pago por uso. Para que las aplicaciones que usan este paradigma escalen adecuadamente es necesario proporcionar a los usuarios de estos sistemas herramientas capaces de explotar los recursos contratados, ya sea un procesador de memoria compartida o un clúster de memoria distribuida. Para ello son muy útiles los contenedores, como Docker, que permiten a los desarrolladores de aplicaciones incluir en el contenedor todo el entorno de ejecución. Así, los usuarios pueden disponer de la aplicación correctamente configurada sin más que elegir la imagen adecuada para su sistema.

Un grupo de aplicaciones de especial relevancia son los simuladores, que se usan ampliamente en el campo científico para analizar la viabilidad de ciertos sistemas. La simulación, y en particular la simulación de eventos discretos, se ha exportado a la nube durante la última década bajo el paradigma "**simulación como servicio**". No obstante, se ha demostrado que este formato de simulación es ciertamente limitante, y exclusivo de sistemas grandes, dejando de lado su ejecución en sistemas de memoria compartida. En este trabajo de fin de máster se analizan nuevos protocolos de simulación distribuida desde puntos de vista más pragmáticos, de acuerdo con los desarrollos tecnológicos tanto de los actuales procesadores como de la nube en sí.

Palabras clave

Simulación distribuida, eventos discretos, servicios en la nube, contenedores Docker, Máquinas virtuales, DEVS, DEVStone.

ABSTRACT

Today's computers are Multi-Core systems, in which each processor contains several execution cores. These shared memory systems can be used individually or combined to form a distributed memory supercomputer. Taking advantage of these two levels of parallelism in these machines is important in order to get the most out of their computing capacity, which is an area of research of great interest at present.

On the other hand, the cloud computing paradigm offers users an almost unlimited set of resources in a pay-per-use model. For applications using this paradigm to scale properly, users of these systems need to be provided with tools capable of exploiting the resources contracted, whether a shared memory processor or a distributed memory cluster. For this purpose, containers, such as Docker, are very useful, allowing application developers to include the entire execution environment in the container. This way, users can have the application correctly configured only having to choose the right image for their system.

A group of applications of special relevance are simulators, which are widely used in the scientific field to analyze the viability of certain systems. Simulation, and in particular the simulation of discrete events, has been exported to the cloud over the last decade under the "simulation as a service" paradigm. However, it has been shown that this simulation format is certainly limiting, and exclusive of large systems, leaving aside its execution in shared memory systems. In this Master Thesis work, new distributed simulation protocols are analyzed from a more pragmatic point of view, in accordance with technological developments both in current processors and in the cloud itself.

Keywords

Distributed simulation, discrete events, cloud services, Docker containers, Virtual Machines, DEVS, DEVStone.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1 - Introducción

El presente capítulo pretende exponer los objetivos que persigue este Trabajo de Fin de Máster, el plan de trabajo que se deberá seguir para alcanzar los objetivos planteados, además de un resumen de cada uno de los capítulos que se desarrollan en el presente trabajo.

1.1 Objetivos

Uno de los principales objetivos es el de construir una extensión del motor de simulación xDEVS [1] que permita definir modelos en plataformas distribuidas con la capacidad de ejecutarse en infraestructuras de la nube.

Asimismo, con la anterior extensión y utilizando servicios de la nube, probar diferentes arquitecturas de simulación que contemplen comunicación entre máquinas virtuales, entre contenedores y clúster.

Por otro lado, utilizar un software que permita realizar pruebas de rendimiento para comparar simulaciones realizadas con xDEVS en su formato secuencial, paralelo y la nueva versión distribuida, para de esa manera obtener conclusiones cuantitativas.

Finalmente, plasmar el proceso y los resultados de los anteriores objetivos en un documento de trabajo de finalización de máster.

1.2 Plan de trabajo

Emplear 4 meses para realizar la investigación del problema y el aprendizaje de las herramientas que permitirán realizar la codificación y pruebas de la versión xDEVS distribuida de manera local, además de usar el modelo de simulación GPT (generador-procesador-transductor) como caso base para las pruebas en adelante.

Asimismo, en 2 meses realizar las pruebas de la versión xDEVS distribuida en infraestructuras de un proveedor de servicios en la nube usando las capacidades de comunicación entre máquinas virtuales, contenedores y clúster que permitan desplegar el sistema de simulación distribuido en estas posibles arquitecturas.

Posteriormente, en 2 meses realizar las pruebas de rendimiento de la versión secuencial, paralela y la nueva versión distribuida de xDEVS para obtener conclusiones cuantitativas y recomendaciones del mejor escenario para cada versión.

Finalmente, en 2 meses realizar el documento de trabajo final de máster con su respectiva revisión y aprobación por parte de los directores.

1.3 Estructura de la memoria

La memoria consta de cinco capítulos explicados de manera resumida a continuación.

Capítulo 1: Introducción. El mismo que expone los principales objetivos y pasos a seguir en el transcurso del proyecto.

Capítulo 2: Motivación y estado del arte. En este capítulo se cubren y explican los aspectos principales que motivan la ejecución de este trabajo, además de los fundamentos y estado en el que se encuentra actualmente la computación de altas prestaciones, junto con el nuevo paradigma que ofrecen los servicios tecnológicos de infraestructura, plataformas y software en la nube. El capítulo termina con un repaso de la situación en la que se encuentra la simulación, especialmente distribuida y basada en eventos discretos, y como la clásica simulación secuencial o paralela pueden trasladarse a la nube para aprovechase de las características subyacentes en esta.

Capítulo 3: Arquitectura del sistema. Aquí se detallan los cambios y ampliaciones realizados a la herramienta de simulación xDEVS para soportar una simulación distribuida capaz de ejecutarse en la nube. Además, se examinan tres arquitecturas de simulación distribuida, todas compatibles con la extensión implementada en este trabajo.

Capítulo 4: Evaluación experimental. En este capítulo se verifica la plataforma distribuida, usando un modelo considerado estándar en la literatura, el Generador-Procesador-Transductor (o GPT, también de sus siglas en inglés Generator, Processor, Transducer). También se detalla la evaluación comparativa de las tres soluciones de simulación que ofrece xDEVS: secuencial, paralela y la nueva opción distribuida. Para cuantificar esta comparativa se utilizan los benchmarks incluidos en DEVStone, y cuyo

resultado será la obtención de aspectos y escenarios en los que cada solución se comporta de una mejor manera.

Capítulo 5. Conclusiones y trabajo futuro. En este capítulo se exponen finalmente las conclusiones del trabajo además de las recomendaciones o sugerencias futuras derivadas de los experimentos realizados y de la experiencia obtenida.

Capítulo 2 - Motivación y estado del arte

En este capítulo se exponen la motivación que da origen a este proyecto, la evolución de la simulación distribuida, el estado actual de la computación de altas prestaciones, los servicios y características ofrecidas en la nube y cómo estas se pueden aprovechar en el ámbito de la simulación distribuida.

2.1 Motivación

En la actualidad las plataformas de simulación se utilizan para resolver problemas importantes en el área de la arquitectura de computadores, sistemas de control, defensa, energía, medio ambiente, evacuación, salud, humanidades, fabricación [2], cadena de suministro [3], marina, redes, espacio, transporte y tráfico [4]. Estas plataformas de simulación suelen utilizarse para la ejecución de tareas complejas que requieren de una gran capacidad de cómputo. Es por ello que frecuentemente dichas plataformas se apoyan en características de la computación de altas prestaciones (High Performance Computing - HPC) [5]. Un estudio de las fuentes de ruido sísmico ambiental y su distribución en función del tiempo y el espacio, es una de las tantas aplicaciones que requieren combinar las ventajas específicas de las plataformas HPC y de la nube, proporcionando así una plataforma distribuida viable para resolver los problemas sismológicos que requieren tanto cálculos complejos como datos sofisticados y gestión del flujo de trabajo [6].

La simulación distribuida promueve la capacidad de tener unidades autónomas, pero capaces de comunicarse entre ellas en un escenario virtual con una noción común de espacio y tiempo. Esto conlleva enormes ventajas desde un punto de vista de escalado, mantenimiento y coste. Pero también se debe tener en cuenta sus desventajas en la falta de rapidez y la falta de sistemas de simulación distribuidos de fácil uso [4].

Durante los últimos años, la computación de altas prestaciones ha tenido una evolución en metodologías, procesos, hardware y software subyacente permitiendo a los proyectos trabajar con infraestructuras multi-core, clusters afrontando retos de alta

complejidad de procesamiento como los que requieren los sistemas de simulación. Sin embargo, también ha presentado limitaciones en cuanto a un fácil escalado, dificultades a la hora de tratar sistemas complejos, no se tiene flexibilidad en la asignación y administración de recursos, coste alto, no siempre se responde con agilidad ante un proyecto [7] [8].

Con la aparición de la computación en la nube [9], se abre una enorme posibilidad de poder unir lo mejor de ambos mundos (HPC y Cloud) y plantear nuevos escenarios de ejecución de simulaciones distribuidas [7][10]. Recientemente se ha propuesto el uso de la computación de altas prestaciones en entornos distribuidos. No obstante, en muchos de estos casos se han olvidado las enormes ventajas que ofrece mantener la memoria compartida como mejor solución en cuanto a la prestación del servicio [5] [4]. Asimismo, al proponerse el uso de servicios de simulación distribuido en la nube con SOA, servicios Rest o servicios de la nube tipo FAAS (Función como servicio – Function As A Service), se han identificado limitaciones no solo de procesamiento y comunicación, sino de almacenamiento local de datos que limitan recordar de manera autónoma el estado en el que se encuentra cada componente del modelo durante el proceso de ejecución [4] [9]. Si bien el modelo de servicio de la nube tiene aún retos que la computación de altas prestaciones y la simulación tradicional debaten y observan, como el uso de la velocidad de comunicación de la red y de la seguridad y/o privacidad de los datos, se espera que estas características vayan mejorando o compensándose como ha sucedido con otras propuestas nuevas y de uso aceptado por el potencial con el que cuentan [8] [9].

El presente trabajo pretende aportar nuevas ideas a la "simulación distribuida" y en concreto a usar los servicios de computación en la nube y las buenas prácticas de la computación de altas prestaciones para proponer una "**nueva arquitectura de simulación distribuida dirigida por eventos**". Este trabajo se desarrollará bajo el uso de los actuales servicios de computación en la nube, las herramientas o software asociados a este, y cómo estos se aplicarían a la simulación distribuida.

En los siguientes apartados, se dará un repaso a la simulación distribuida, computación de altas prestaciones y computación en la nube, todo ello para tener una mejor idea de las áreas con las que se trabajará en el presente proyecto.

2.2 Simulación distribuida

Si tomamos ejemplos de sistemas reales, sucede que las entidades están distribuidas, son independientes y usan sus propios recursos, buscando un medio común y lográndose comunicar con las demás entidades de manera coordinada y bajo un protocolo estándar de interacción. Este último escenario es análogo a cómo se plantea una simulación distribuida en donde la carga de la simulación puede ser compartida por múltiples procesadores distribuidos a través de la red [11].

En un escenario en donde el volumen de mensajes no es grande, ya que se trata de modelos débilmente acoplados para ejecutarse juntos o ejecuciones replicadas del mismo modelo, la distribución de la carga puede verse beneficiada. Asimismo, si los modelos por naturaleza son distribuidos, más compatibilidad tendrán para ser resueltas mediante una simulación distribuida.

Algunas ventajas y desventajas que arroja la simulación distribuida pueden verse en la siguiente tabla:

Ventajas	**Desventajas**
• Tener la capacidad de la reutilización que le permite construir modelos por composición. • Promueve la heterogeneidad y la dispersión, vinculando la simulación de componentes de varios tipos (discreto y continuo). • Se beneficia con un mejor manejo de tolerancia a fallos.	• Existen puntos a trabajar como la interoperabilidad de componentes de sistemas heterogéneos. • Dificultad de crear nuevos modelos. • Uniformidad del comportamiento de los modelos a lo largo de la ejecución de una simulación. • Ejecución de modelos dentro de sistemas distribuidos complejos.

• Puede aumentarse el poder de computo al utilizar de mejor manera cada recurso asociado al elemento de simulación.	

***Tabla 2.1**: Ventajas y desventajas de la simulación distribuida.*

Las tecnologías como la programación funcional, el desarrollo de sistemas reactivos, los micro servicios, Arquitecturas Orientadas a Servicios (SOA), virtualización mediante contenedores y servicios de computación en la nube, son herramientas que pueden ayudar a cubrir las desventajas mencionadas en la anterior tabla y potenciar aún más las ventajas asociadas a la simulación distribuida [12] [9].

En ese sentido, muchas soluciones distribuidas del mercado comenzaron utilizando HLA (High-Level Architecture, obsoletas a la fecha), para luego usar las Arquitecturas Orientadas a Servicios (SOA) que permitieron integrar y consumir servicios independientemente de la naturaleza de su implementación a través del paso de mensajes y en la actualidad pueden fácilmente desplegarse en la nube. Sin embargo, SOA es conceptualmente monolítica, por lo que en los despliegues de actualizaciones de software el sistema distribuido entero debe sufrir tiempos de inactividad para los usuarios finales mientras no concluyan las actualizaciones. Esto limita la capacidad de escalar fácilmente la aplicación en caso de aumento o disminución de las cargas de trabajo, lo que contradice con la propuesta de la nube de hacer la disponibilidad de servicios 24x7 y con un fácil escalado de recursos [12] [9]. Así, SOA evolucionó al estilo arquitectónico de microservicios, diseñados para ser independientemente desplegables, actualizables y escalables horizontalmente. Esta arquitectura, ante varias peticiones de un servicio, permite que se lancen más instancias de servicio y en cambio si las peticiones disminuyen, las instancias de servicio se cierran para liberar recursos [9]. Por otro lado, los microservicios han hecho populares el uso de los contenedores al tratarlos como unidades de despliegue autónomas, lo que permite realizar un escalado sencillo y rápido, además de consumir menos recursos que el despliegue de máquinas virtuales. Cabe destacar que los proveedores de los servicios de computación en la nube adoptaron a los contenedores como un estándar para desplegar y hacer que sus

clientes desplieguen o administren sus servicios especialmente IaaS (Infraestructura como servicio). Así mismo adoptaron el servicio de Kubernetes para realizar el escalado u orquestación de contenedores según la carga de peticiones en un clúster [12] [9].

Aunque las arquitecturas de microservicio resuelven los problemas de SOA, el concepto de servicio es un concepto siempre activo, por lo que al menos una instancia de servicio (contenedor) debe estar viva todo el tiempo para atender el microservicio. Ante esta observación del microservicio, se dio paso a los conceptos: función como servicio (FaaS) y arquitecturas sin servidores, de tal manera que estas permiten ejecutar instancias de servicio sólo en el caso de tener peticiones reales. Las FaaS exponen funciones sin estado que se facturan por el tiempo de ejecución consumido, generalmente en milisegundos. Estas pueden desplegarse rápidamente y escalarse automáticamente. Sin embargo, para la simulación se deberá considerar que estas no almacenan los estados o pasos que sucedieron durante parte o la totalidad de la simulación realizada [9].

Una simulación distribuida apoyada con las posibilidades y herramientas mencionadas anteriormente hace que sea posible hablar del concepto de Modelado y Simulación como Servicio (MSaaS), que es algo que la comunidad ha estado comenzado a buscar para no dejar pasar la oportunidad de aprovechar especialmente las capacidades de la Nube [5] [12] [9]. Aunque aún está en una fase muy temprana, el tan solo proponer una arquitectura alternativa de simulación distribuida en la nube ya es un gran avance hacia un MSaaS eficiente, para lo que aún queda un largo camino por recorrer, y en el que este trabajo pone especial énfasis.

2.3 Computación de altas prestaciones

La computación de altas prestaciones (High Performance Computing - HPC) puede resolver problemas y encarar desafíos que exigen alto poder de cómputo para resolver tareas complejas. Se apoya en la creación de un sistema productivo que incluye lo mejor de los procesadores (velocidad y paralelismo), aceleradores (más velocidad), memoria, almacenamiento (mejor gestión de la escalabilidad en datos y mejor velocidad de acceso) y redes (mayor ancho de banda y baja latencia), pero también

de un software moderno, extensible, de alto rendimiento y flexible [13] [10]. Son muchas las áreas que dependen en gran medida de la computación de altas prestaciones, entre las que se encuentran la simulación, genómica, fabricación, química computacional, modelización financiera, exploración de la energía, etc. [13]. Sin embargo, hay consideraciones a tomar en cuenta como la ejecución de las aplicaciones en equipos especiales y específicos de hardware y software, que conllevan un mantenimiento económico alto, pero también implican el coste de conseguir y mantener a un personal especializado. Estos son solo algunos de los principales retos que HPC enfrenta hoy en día, y que la computación en la nube ya ha comenzado a resolver [7] [10].

Antes de entrar a describir las bondades de HPC o de la computación en la nube, conviene mostrar datos acerca de sus arquitecturas y cómo estas se diferencian [14]:

Característica	**HPC**	**Cloud**
Gestión de recursos	Utilizado para encontrar recursos, software de aplicaciones, además de la gestión de procesadores.	Utilizado para la creación y recuperación de recursos dinámicos, y la gestión de procesadores, memoria, almacenamiento, redes y aplicaciones.
Virtualización	Poco soportado.	Virtualización de servidores, virtualización de almacenamiento, virtualización de redes.
Gestión de usuarios	Sistema de gestión independiente del usuario: el usuario no puede acceder a los recursos de manera directa e independiente.	Gestión unificada de usuarios: los usuarios pueden acceder a recursos exclusivos.

Soporte de plataforma	Incapaz de modificar la plataforma instalada e incapaz de modificarla de manera dinámica.	Soportan múltiples plataformas al mismo tiempo y pueden ser modificadas de manera dinámica.
Almacenamiento de datos	No hay mecanismo de respaldo ni soporte para el almacenamiento de datos heterogéneos.	Tiene un perfecto mecanismo de respaldo y recuperación, y una plataforma de soporte de almacenamiento de datos heterogéneos.
Uso	No hay un proceso de aprobación de recursos y no se puede personalizar la asignación de recursos.	mecanismo de aprobación, rechazo y reserva, además de poder personalizar la asignación de recursos.

***Tabla 2.2**: Diferencias entre las arquitecturas HPC y Cloud*

En las plataformas de la nube se usan soluciones OpenStack [15] que permiten proporcionar una manera de juntar múltiples silos en una sola nube privada, además de hacer que los recursos sean aún más accesibles a través de portales de autoservicio y API's [7] [16]. Utilizando OpenStack, la computación en la nube puede distribuir múltiples cargas de trabajo entre los recursos de forma más granular, lo que aumenta la capacidad de utilización general y reduce el coste [7] [16]. En tanto que los sistemas tradicionales de HPC son mejores para una cierta y especifica carga de trabajo, las infraestructuras de la nube pueden acomodar muchas otras [7]. Asimismo, las infraestructuras tradicionales de HPC son excelentes para resolver un problema particular, pero no son muy buenas para el tipo de colaboración que requiere la investigación y la empresa moderna [7] [10]. Mediante los servicios de la nube es posible proporcionar el resultado final del proceso no solo al personal exclusivo de HPC sino a miles o millones de usuarios finales de manera directa [7] [13] [10].

Algunos aspectos pueden interesar actualmente a HPC, aunque finalmente llegaran a ser parte de la computación en la nube. Ejemplos son la virtualización y los

contenedores que han ido evolucionando para reducir aún más la brecha de rendimiento entre los clústeres in situ y las nubes públicas[7] [10] [13]. De esta manera, las redes rápidas y de baja latencia, y los nuevos aceleradores: GPU, FPGAs, TPU, son ya o podrán ser parte de la nube. Es cuestión de que dejen de ser componentes exclusivos de HPC [7] [10] [13].

Con el aumento de los micro servicios y las tecnologías para DevOps[1] [17] sobre una plataforma OpenStack, la transformación de las aplicaciones de HPC existentes en SaaS, capaces de abstraer las capas de infraestructura, puede convertirse en una tendencia para hacer más popular HPC [7]. Por otro lado, los esfuerzos comerciales y de investigación pueden impulsar una mejor comprensión de los modelos de precios de asignación de recursos sostenibles tanto para los proveedores de la nube como para los usuarios de HPC (conciencia del uso de recursos indiscriminado porque en la nube estos tienen coste) [7] [10] [13].

Finalmente, procesar grandes volúmenes de datos y resolver cálculos complejos se vuelven cada vez más relevantes debido a la creciente salida de aplicaciones HPC en un contexto distribuido [7]. Esto es producto de la capacidad de capturar y generar datos mediante sistemas IoT (Internet de las cosas), inteligencia artificial, Deep learning, machine learning, procesamiento del lenguaje natural, la digitalización y automatización, entre otros [13] [7]. Está claro que ante tanta y variada complejidad HPC tendrá que trabajar de la mano con lo mejor de la computación en la nube.

2.4 Computación en la nube

Si bien desde la década de los sesenta existen varios intentos, conceptos e ideas de computación en la nube, no es hasta 1999 en donde la empresa salesforce.com de manera pionera, promueve y logra ofrecer aplicaciones empresariales mediante internet [18]. Posteriormente, Amazon EC2/S3 en 2008 se postula como la primera

[1] El término DevOps, es una combinación de los términos ingleses development (desarrollo) y operations (operaciones), designa la unión de personas, procesos y tecnología para ofrecer valor a los clientes de forma constante.

empresa en ofrecer servicios de infraestructura en la nube [18]. Esta corriente de servicios y aplicaciones en la nube motiva a dos grandes del área de la tecnología, Google y Microsoft, a proponer aplicaciones basadas en un navegador web dando una clara señal de que el ordenador tal como se lo conoce ya no era tan necesario para el procesamiento y almacenamiento de la información diaria y cotidiana y que cualquier otro servicio por más complejo que sea, era posible en la nube [18]. Asimismo, los adelantos en la computación distribuida, tanto en hardware como en software, además de la aparición y uso concreto de la virtualización de software dan paso a que la computación en la nube sea una realidad y una propuesta a la que más usuarios, áreas y proveedores se van sumando día a día [19].

En la actualidad y de manera práctica, la computación en la nube contempla la entrega de varios servicios a través de Internet, permitiendo principalmente la implantación de infraestructura de hardware y software rápida y ágil, evitando realizar grandes inversiones iniciales y posibilitando el consumo de recursos a demanda, es decir, la posibilidad de pagar solo por los servicios que se consumen, lo que beneficia de sobremanera a las pequeñas y medianas empresas a poner en marcha sus proyectos o emprendimientos lo antes posible [8] [9]. Por tanto, el poder de computación que ofrecen los servicios de la nube está al alcance de cualquier persona u organización y puede ser aplicado a cualquier área, especialmente a las que requieren altos picos de carga como los que se hacen en la simulación distribuida, sin la necesidad de tener siempre activa la infraestructura computacional [9].

De la misma manera, cabe destacar que los servicios de computación en la nube proporcionan características como [8]: alta disponibilidad, escalabilidad, elasticidad, agilidad, tolerancia a fallos, recuperación, latencia del cliente, coste predictivo, requisitos o consideraciones de habilidades técnicas, aumento de la productividad y seguridad. Por otro lado, una las razones que influyen de manera importante en la viabilidad de los proyectos es la inversión y gestión económica. La computación en la nube plantea un cambio de visión en la gestión de costes al momento de pagar por el uso de los servicios consumidos y recomienda moverse bajo dos factores [8]: ingresar al uso de las economías de escala y transformar gastos de capital en gastos operativos.

En general la computación en la nube es estudiada desde el punto de vista del **servicio**: Software como servicio (SaaS – Software as a Service), Plataforma como servicio (PaaS – Platform as a Service) e Infraestructura como servicio (IaaS - Infraestructure as a Service). O desde el punto de vista de **despliegue:** nube privada, nube pública, nube híbrida y nube de comunidad. Otros lo ven desde el punto de vista del **agente/usuario**: proveedor, consumidor, auditor, ejecutor, portador. O simplemente desde el punto de vista **funcional:** despliegue de servicios, orquestación de servicios, gestión de servicios, la seguridad, la privacidad. Sin embargo, los puntos de vista mencionados son útiles desde el lado del proveedor de servicio, pero no desde un punto de vista de la ingeniería de aplicaciones sobre la nube[9], por lo que cada área que desee aplicar los servicios de la nube deberá plantearse y analizar el punto de vista sobre el cual conviene desplegar sus aplicaciones, pero tomando en cuenta la tecnología implicada y aplicada al área que corresponda. Por ejemplo, en el área de simulación existen arquitecturas de hardware y software, tipos de cargas de trabajo, procedimientos, metodologías y tipos de comunicación. Tal vez una nube hibrida, con una mínima participación de recursos IaaS acompañado de una orquestación y despliegue de servicios sea una alternativa estándar para la simulación. Sin embargo, como la anterior, seguramente habrá más alternativas y propuestas que deberán realizarse antes de llegar a algo concluyente, por lo pronto es algo que no se ha resuelto aún en muchas áreas y la simulación no es la excepción [9].

Por otro lado, a nivel base, los estándares de la nube se centran en un conjunto mínimo pero necesario de recursos: los nodos de cómputo (máquinas virtuales), almacenamiento (archivo, bloque, objeto), y de red (privada virtual). Sin embargo, también es necesario pensar en otro tipo de servicios complementarios, como la capacidad de tener sistemas de balanceo de carga, auto escala o de gestión de colas de mensajes para diseñar un sistema elástico y escalable, capaz de ser soportado de manera estándar por todos los proveedores, lo que no sucede hasta ahora, teniéndose como resultado la ausencia de una pila estándar de recursos y servicios subyacentes a la administración y despliegue de cualquier tipo de aplicación en la nube pero independiente del proveedor [9].

En la actualidad, la optimización de recursos ha evolucionado en unidades de despliegue autónomas y estandarizadas (contenedores), junto con las recientes soluciones de Función como Servicio (FaaS - Function as a Service), esto ha permitido aumentar la capacidad de despliegue del hardware mediante la aplicación de tiempo compartido por cada servicio usado [9]. Esta optimización de utilización de recursos, está acompañada por los respectivos estilos arquitectónicos que se esfuerzan por superar a las propuestas de despliegue monolítico y prefieren más la descomposición de servicios de grano fino [9]. Estas tendencias podrían hacer que los servicios de simulación en la nube tiendan a ser similares a un micro servicio [9]. En ese sentido, para aprovechar las oportunidades de la computación en la nube, estos deben ser mucho más estructurados y parecerse a componentes de servicios de simulación de grano fino. Asimismo, los servicios de simulación deben esforzarse por convertirse en "**sin estado**" o "**estado aislado**", o tener un mínimo de componentes con estado [9]. Sin embargo, la naturaleza inherente de las simulaciones se basa profundamente en los estados (de datos), con lo que se abre una puerta más a resolver [9].

Este trabajo intentará contribuir en este sentido, como se verá en los capítulos siguientes.

Capítulo 3 - Arquitectura del sistema

Este capítulo comienza explicando de manera general el formalismo DEVS y la estructura de simulación utilizada en el modelo acoplado EF-P, que luego servirá de base para verificar el uso del modelo de simulación distribuida DEVS sobre la nube propuesto en este trabajo. Posterior a ello, describe el diseño de clases, así como las condiciones que guiarán los cambios a realizar en el código de la librería xDEVS y que permitirán soportar la simulación distribuida. En tercer lugar, se explica el uso de las nuevas clases de simulación distribuida, diseñando un ejemplo sobre el modelo acoplado EF-P. Finalmente se describen las condiciones a cumplir para el despliegue del software de simulación distribuida en la nube, así como también tres posibles arquitecturas de simulación distribuida que pueden desplegarse usando los servicios de infraestructura de la nube: con **máquinas virtuales**, con **contenedores Docker** y con un **clúster usando Kubernetes**.

3.1 Formalismo DEVS en el presente trabajo.

DEVS [20] es un formalismo genérico que permite describir modelos, realizar simulaciones o análisis de sistemas conducidos por eventos discretos. En él se propone que un sistema esté formado por modelos atómicos en su unidad más simple e indivisible, y por modelos acoplados como la composición de modelos atómicos y a su vez acoplados, lo que significa que es posible la combinación de ambos para formar un todo interconectado mediante sus puertos de entrada y salida, tal como se observa en la siguiente figura [21]:

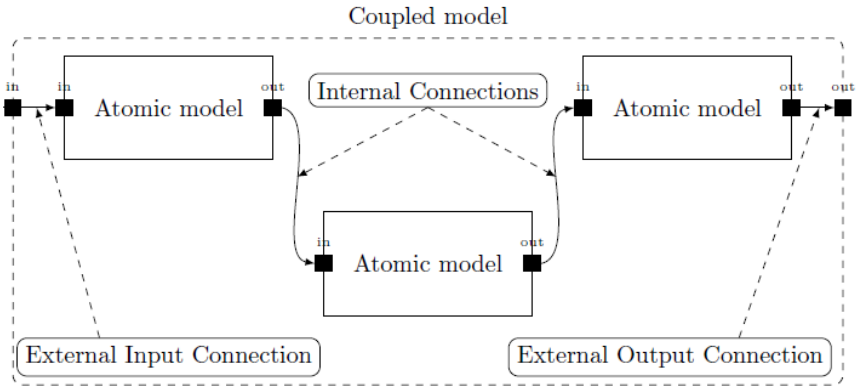

***Figura 3.1**: Modelo general, compuesto por unidades atómicas y/o acopladas junto con sus respectivos puertos de entrada y salida internos y externos [21].*

En el presente trabajo se tomará el modelo acoplado "Experimental Frame – Processor" o EF-P como ejemplo para realizar las pruebas. La estructura de este modelo está compuesta por tres modelos atómicos y un modelo acoplado, tal como se muestra a continuación [21]:

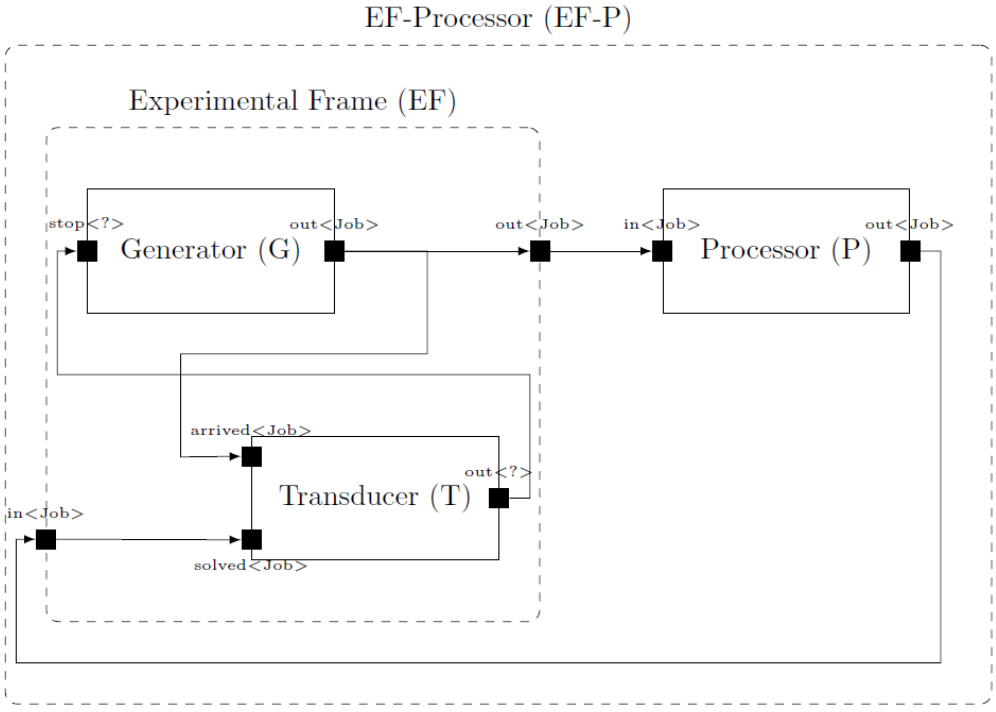

***Figura 3.2**: Estructura del modelo acoplado DEVS EF-P [21].*

En DEVS, para especificar la estructura de una simulación se usan coordinadores y simuladores. Cada modelo o componente atómico se asocia a un simulador. En caso

de ser un modelo acoplado este se asocia con un coordinador, tal como se muestra en la figura [22]:

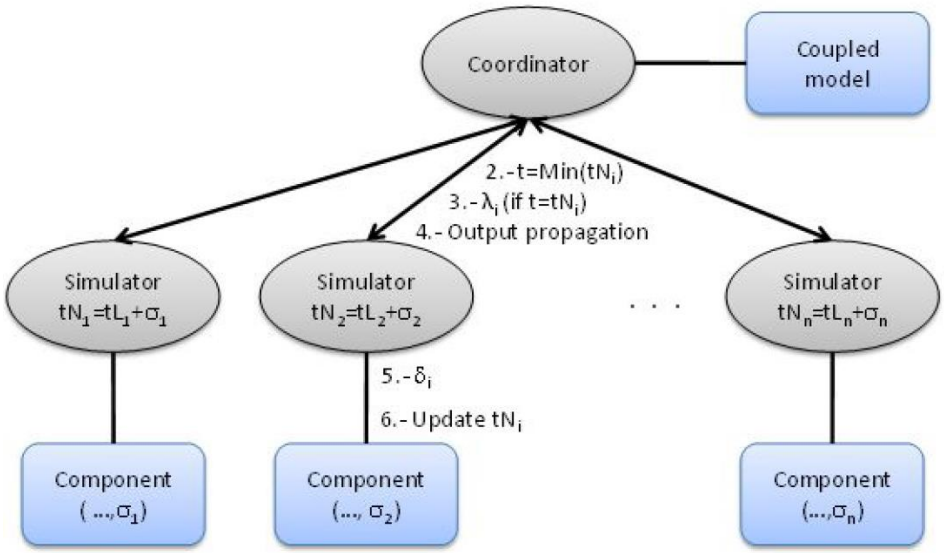

***Figura 3.3**: Ejemplo de estructura simulación DEVS [22].*

Una vez establecido la estructura de simulación, el coordinador solicita a cada simulador que proporcione el instante de su próximo evento (tN) y determina el mínimo de los valores devueltos para obtener la hora del próximo evento. El tiempo actual se establece en: $t = \min(tN_i)$ con $i = 2$ en el caso de la figura anterior. Los simuladores aplican en el modelo correspondiente $t = tN_i$ el método λ_i, para producir una salida y propagarla en todos los puertos en los que el modelo correspondiente ha dejado uno o más valores. A continuación, todos los simuladores ejecutan una función de transición interna que trata de determinar el efecto combinado de la salida propagada y el nuevo estado. Otro cálculo que se produce es la actualización del instante del próximo evento: tN usando el tiempo de avance del modelo. Finalmente, el coordinador obtiene el siguiente tiempo del próximo evento y los pasos explicados hasta ahora se repiten hasta que se alcance el número de iteraciones o un tiempo de simulación establecido previamente.

Tanto el modelo EF-P como el motor de simulación DEVS ya se encuentran implementadas en la librería xDEVS [1], y se usarán como base para incorporar

funcionalidad y poner a prueba el modelo de simulación distribuida DEVS sobre la nube diseñado en este trabajo.

3.2 Diseño de xDEVS distribuido para la nube

Como se había indicado en el anterior apartado, la librería xDEVS ya cuenta con un desarrollo para la especificación de modelos DEVS, así como la posibilidad de realizar simulaciones secuenciales y paralelas. Sin embargo, no cuenta con un desarrollo específico para una simulación distribuida que pueda ejecutarse sobre las diferentes arquitecturas que los proveedores de servicios de la nube ofrecen como servicio. En ese sentido, como paso inicial se plantea la implementación del siguiente modelo de clases que proporciona a xDEVS la capacidad de simulación distribuida:

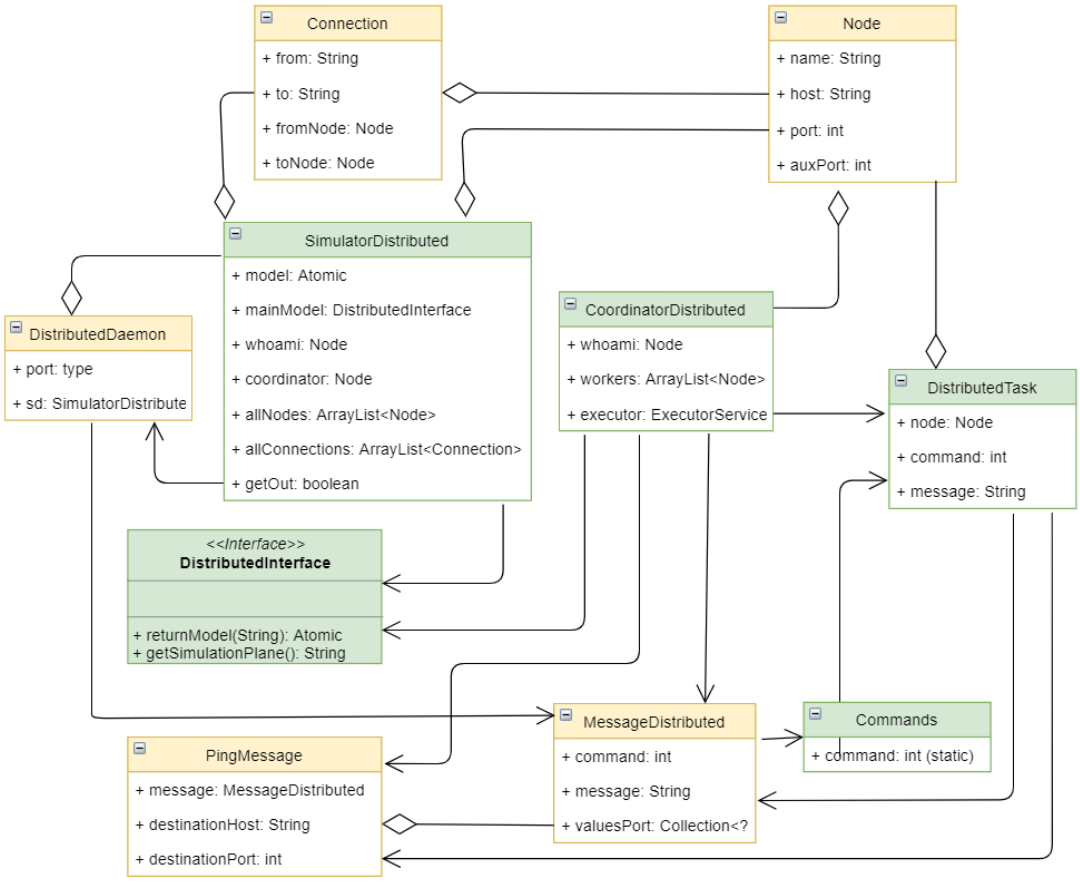

***Figura 3.4**: Diagrama de clases para soportar el modelo de simulación distribuida sobre xDEVS.*

Este diseño está basado en una arquitectura distribuida tradicional en la que cada cliente/servidor es capaz de escuchar, contestar y procesar mensajes de manera

independiente y concurrente. Este diseño se puede analizar identificando dos conjuntos de clases: las de red/hardware (fondo amarillo en la Fig. 3.4) y las de software de simulación (fondo verde en la Fig. 3.4).

Dentro de las **clases de red/hardware**, se encuentra la clase "Node" que abstrae el comportamiento de un nodo o elemento de la red: hostname (name), IP (host), Puerto (port) y Puerto auxiliar (auxPost)[2]. También está la clase "Connection", que representa el enlace de red que se establece entre dos objetos de la clase "Node". Además, está la clase "MessageDistributed", que abstrae el mensaje que se envía entre los nodos. La clase "PingMessage" implementa el mecanismo para enviar mensajes a los nodos (análogo al comando de red ping usado para enviar paquetes ICMP). Por último, la clase "DistributedDaemon", que permite a los nodos la capacidad de escuchar peticiones en sus dos puertos: principal (usado para la comunicación Coordinador-Simulador) y auxiliar (usado para la comunicación entre simuladores).

Por otro lado, dentro del grupo de **clases de software de simulación** se encuentra la clase utilitaria "Commands", que contiene el lenguaje que permite entender a todos los nodos (simuladores o coordinadores) cuál es el proceso que deben ejecutar. Estos solo reconocerán los comandos que se encuentran registrados en esta clase, en la que cada comando es una constante con un valor entero único. La clase "SimulatorDistributed" es una clase que hereda comportamiento inicial de la clase base "AbstractSimulator", sin embargo, añade la funcionalidad necesaria para que un modelo atómico se ejecute en cualquier nodo de la red y pueda dialogar con los demás nodos que contienen un simulador o coordinador como software. La clase "CoordinatorDistributed" se encarga de coordinar las tareas que tienen que ejecutar los simuladores de la clase "SimulatorDistributed". Esta comunicación coordinador-simulador no se realiza de manera secuencial sino de manera concurrente gracias a la funcionalidad añadida por la clase "DistributedTask". Finalmente, la interfaz

[2] Este último se usará para comunicarse con otro elemento de la red de manera concurrente.

"DistributedInterface" deberá ser utilizada por aquellos modelos DEVS que quieran correr sobre este diseño distribuido.

La condición principal para que un modelo DEVS se ejecute de manera distribuida sobre este diseño, es que se encuentre modelado a nivel de componentes atómicos y no acoplados, es decir cualquier componente acoplado del modelo total deberá reducirse a componentes atómicos (esta característica es posible en el modelado DEVS, que permite "aplanar" modelos compuestos eliminando los componentes acoplados, y manteniendo únicamente los atómicos hijos, todo ello de forma recursiva [20]). Para reflejar mejor la idea anterior, en la figura 3.5 se muestra un ejemplo de conversión del modelo EF-P a la nueva estructura de simulación distribuida.

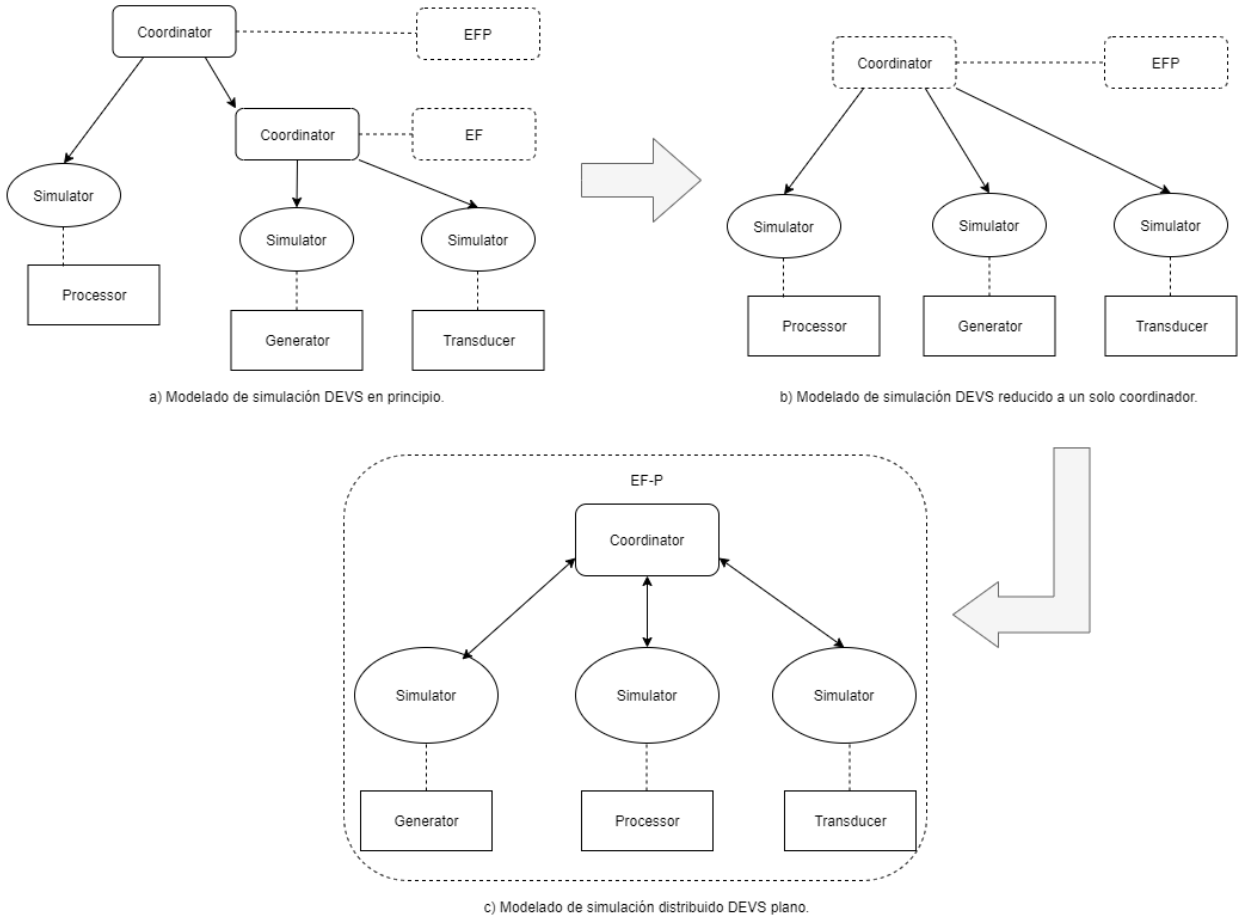

***Figura 3.5**: Forma de presentar un modelo de simulación DEVS al nuevo modelo de simulación distribuida.*

El modelo de simulación **a)** se presenta de manera inicial con un coordinador por cada modelo acoplado, sin embargo, es necesario reducir el modelo de simulación a **b)** en donde se tiene un solo coordinador asociado a su modelo acoplado principal EF-P, para finalmente, quedarnos con la idea de **c),** entendiendo que el modelo de simulación total tiene un solo coordinador y tres simuladores (cada uno atendiendo un modelo atómico específico). Juntos representan el modelo de simulación para EF-P. Las trayectorias de estados de a), b) y c) son exactamente las mismas, son modelos equivalentes.

Basados en la nueva propuesta de simulación distribuida, está claro que las posibilidades de ejecutar cada componente de un modelo DEVS en un nodo diferente dentro de la red es posible, como también es posible ejecutar todo el modelo de simulación en una sola máquina o nodo. Las posibilidades de combinación son muchas, en tanto exista conectividad de red y que los puertos sean exclusivos para cada nodo.

3.3 Verificación del nuevo motor de simulación DEVS distribuido a través del modelo estándar EF-P

Para probar el modelo DEVS EF-P en el nuevo motor de simulación xDEVS distribuido, es necesario crear una clase principal que herede de la clase base "Coupled" e implemente la interfaz "DistributedInterface", tal como se muestra a continuación:

```java
public class GptDistributed extends Coupled implements DistributedInterface{

    // Atributtes GptDistributed
    private String simulationPlane;
    private double period;
    private double processingTime;
    private double observationTime;

    // Constructors GptDistributed
    public GptDistributed(String name, double period, double observationTime, String simulationPlane){
        this(period, observationTime, simulationPlane);
        super.name= name;
    }
    public GptDistributed(double period, double observationTime, String simulationPlane){
        super(GptDistributed.class.getSimpleName());
        this.period = period;
        this.processingTime = period*3;
        this.observationTime = observationTime;
        this.simulationPlane = simulationPlane;
    }

    // Methods GptDistributed
    public Atomic returnModel(String ClassName) {
        if (ClassName.compareTo(Generator.class.getName()) == 0){
            return new Generator(ClassName, this.period);
        } else if (ClassName.compareTo(Processor.class.getName()) == 0){
            return new Processor(ClassName, this.processingTime);
        } else {
            return new Transducer(ClassName, this.observationTime);
        }
    }

    public String getSimulationPlane() {
        return simulationPlane;
    }

}
```

***Figura 3.6**: GptDistributed.java*

En el ejemplo anterior, la clase "GptDistributed" abstrae todas las características de configuración necesarias para llevar a cabo la simulación distribuida del modelo EF-P, entre estas está el periodo, el tiempo de observación y un plano de simulación. Este último es el nombre de un archivo XML que contiene la información necesaria para identificar a cada nodo en la red y su asociación con su respectivo simulador y su componente atómico además del coordinador. Este archivo XML tiene que cumplir con la siguiente estructura predeterminada:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<simulation>
    <coordinator name="CoordinatorDistributed" class="xdevs.core.simulation.distributed.CoordinatorDistributed"
                 host="127.0.0.1" mainPort="5000" auxPort="6000"/>
    <atomic name="Generator" class="xdevs.core.test.efp.Generator" host="127.0.0.1" mainPort="5001" auxPort="6001">
        <inport name="stop" class="xdevs.core.test.efp.Job"/>
        <outport name="out" class="xdevs.core.test.efp.Job"/>
    </atomic>
    <atomic name="Processor" class="xdevs.core.test.efp.Processor" host="127.0.0.1" mainPort="5002" auxPort="6002">
        <inport name="in" class="xdevs.core.test.efp.Job"/>
        <outport name="out" class="xdevs.core.test.efp.Job"/>
    </atomic>
    <atomic name="Transducer" class="xdevs.core.test.efp.Transducer" host="127.0.0.1" mainPort="5003" auxPort="6003">
        <inport name="solved" class="xdevs.core.test.efp.Job"/>
        <inport name="arrived" class="xdevs.core.test.efp.Job"/>
        <outport name="out" class="xdevs.core.test.efp.Job"/>
    </atomic>
    <connection atomicFrom="Processor" classFrom="xdevs.core.test.efp.Processor" portFrom="oOut" atomicTo="Transducer"
                classTo="xdevs.core.test.efp.Transducer" portTo="iSolved"/>
    <connection atomicFrom="Generator" classFrom="xdevs.core.test.efp.Generator" portFrom="oOut" atomicTo="Processor"
                classTo="xdevs.core.test.efp.Processor" portTo="iIn"/>
    <connection atomicFrom="Generator" classFrom="xdevs.core.test.efp.Generator" portFrom="oOut" atomicTo="Transducer"
                classTo="xdevs.core.test.efp.Transducer" portTo="iArrived"/>
    <connection atomicFrom="Transducer" classFrom="xdevs.core.test.efp.Transducer" portFrom="oOut" atomicTo="Generator"
    classTo="xdevs.core.test.efp.Generator" portTo="iStop"/>
</simulation>
```

Figura 3.7: *Ejemplo del plano/arquitectura de simulación distribuida del modelo acoplado EF-P descrito en el archivo gpt.xml*

EL archivo XML pide de cada elemento de la simulación (coordinador, simuladores/atómicos) el nombre (name), el IP (host), el puerto principal (mainPort) y el puerto auxiliar (auxPort) como datos obligatorios. El segmento de conexiones entre simuladores/Atómicos exige que cada conexión tenga como dato obligatorio el nombre del modelo atómico origen (atomicFrom), el nombre del puerto de salida del modelo atómico origen (portFrom) y sus respectivos nombres de modelo atómico y su puerto de entrada de la conexión destino (atomicTo y portTo).

Una vez construido el anterior archivo XML, se deben realizar dos pasos: **crear el coordinador y ejecutarlo** y **crear los simuladores y ejecutarlos**.

Para **crear el coordinador y ejecutarlo**, es necesario implementar el método principal en una nueva clase (para el caso de la figura 3.8 el método se encuentra en la clase TestCoordinator):

```java
public static void main(String[] args) {
    DevsLogger.setup(Level.INFO);
    GptDistributed gpt = new GptDistributed("gpt", 1, 30, "gpt.xml");
    CoordinatorDistributed coordinator = new CoordinatorDistributed(gpt);
    coordinator.initialize();
    coordinator.simulate(Long.MAX_VALUE);
    coordinator.exit();
}
```

***Figura 3.8**: Método principal que pondrá a funcionar al coordinador.*

Este método crea inicialmente un objeto de la clase "GptDistributed" que básicamente tiene la finalidad de contener el plano de la red y el software de la simulación, para luego pasar ese objeto como argumento a un objeto de la clase "CoordinatorDistributed" el mismo que al momento de inicializar su reloj y atributos avisará a todos los simuladores que hagan la misma tarea de manera local "coordinator.initialize". Se debe tomar en cuenta que hay un solo reloj global en posesión del coordinador, llevando el compás de pasos para toda la simulación. Una vez concluida la simulación con "coordinator.simulate(Long.MAX_VALUE)", el coordinador enviará un mensaje de finalización a todos los simuladores para indicarles que deben de terminar su ciclo de vida "coordinator.exit()".

Hasta este punto se proporciona en este trabajo una arquitectura de código como la que se muestra en la siguiente figura 3.9, en donde claramente se ve que las clases que se encuentran dentro del marco de línea punteada (inciso a) se utilizan y colaboran para expresar la ubicación, relación y características de red de cada nodo junto con los modelos atómicos que cada uno de ellos deberá ejecutar. También se debe recalcar que el usuario de la versión distribuida de xDEVS, deberá implementar una clase similar a la de GptDistributed, para cada modelo acoplado que tenga en mente para ejecutar de forma distribuida en la nube.

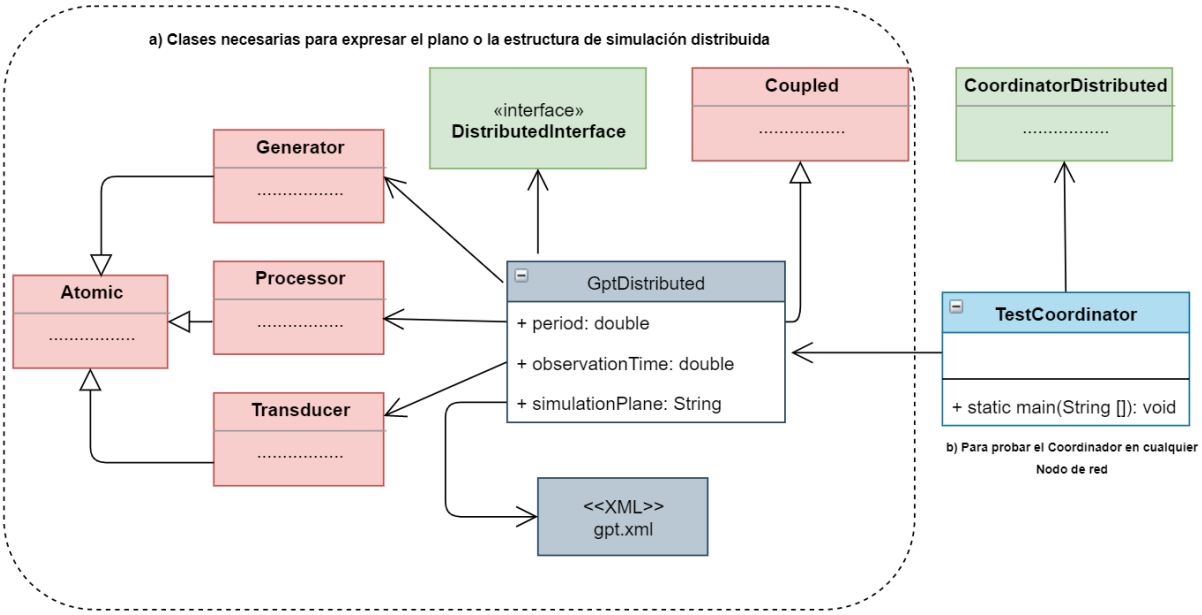

***Figura 3.9**: Uso de las clases necesarias para poder lanzar un Coordinador Distribuido.*

Para **crear y ejecutar los simuladores**, el paso de construcción es aún más simple ya que solo se necesita una clase con un método principal que contenga un objeto de la clase "GptDistributed" con el mismo archivo XML que se le paso al coordinador. Como siguiente paso es necesario crear un objeto de la clase "SimulatorDistributed" cuyos parámetros serán el objeto creado de la clase "GptDistributed", el IP y los puertos principal y auxiliar sobre el cual correrá el simulador (estos tres últimos parámetros deberán coincidir con el dato que figura en el archivo XML para emparejar con el componente atómico asociado a este simulador). Se deberá tener en cuenta que al momento de crear el objeto de la clase "SimulatorDistributed" este se pone a escuchar automáticamente peticiones de los otros simuladores o del coordinador. A continuación, se muestran las tres clases que simularán los modelos atómicos: Generator, Processor y Transducer:

```java
public class TestSimulatorG {
    public static void main(String[] args) {
        DevsLogger.setup(Level.INFO);
        GptDistributed gpt = new GptDistributed(1, 30, "gpt.xml");
        //SimulatorDistributed p = new SimulatorDistributed(gpt, "172.17.0.3",5001,6001);
        System.out.println("Run Generator Atomic....");
        SimulatorDistributed p = new SimulatorDistributed(gpt, "127.0.0.1",5001,6001);
    }
}

public class TestSimulatorP {
    public static void main(String[] args) {
        DevsLogger.setup(Level.INFO);
        GptDistributed gpt = new GptDistributed(1, 30, "gpt.xml");
        //SimulatorDistributed p = new SimulatorDistributed(gpt, "172.17.0.4",5002,6002);
        System.out.println("Run Processor Atomic....");
        SimulatorDistributed p = new SimulatorDistributed(gpt, "127.0.0.1",5002,6002);
    }
}

public class TestSimulatorT {
    public static void main(String[] args) {
        DevsLogger.setup(Level.INFO);
        GptDistributed gpt = new GptDistributed(1, 30, "gpt.xml");
        //SimulatorDistributed p = new SimulatorDistributed(gpt, "172.17.0.5",5003,6003);
        System.out.println("Run Tranducer Atomic....");
        SimulatorDistributed p = new SimulatorDistributed(gpt, "127.0.0.1",5003,6003);
    }
}
```

***Figura 3.10**: Clases con sus respectivos métodos principales que lanzarán los simuladores asociados al componente Generator, Processor y Transducer.*

Una vista rápida del código anterior en un diagrama de clases permite evidenciar claramente que es necesario pasar como parámetro a cada simulador del modelo, un objeto de la clase GptDistributed, de manera que cada simulador sepa qué modelo atómico tiene que ejecutar, qué puertos usar y qué datos de red y modelos están ejecutando los demás simuladores (incisos c, d y e):

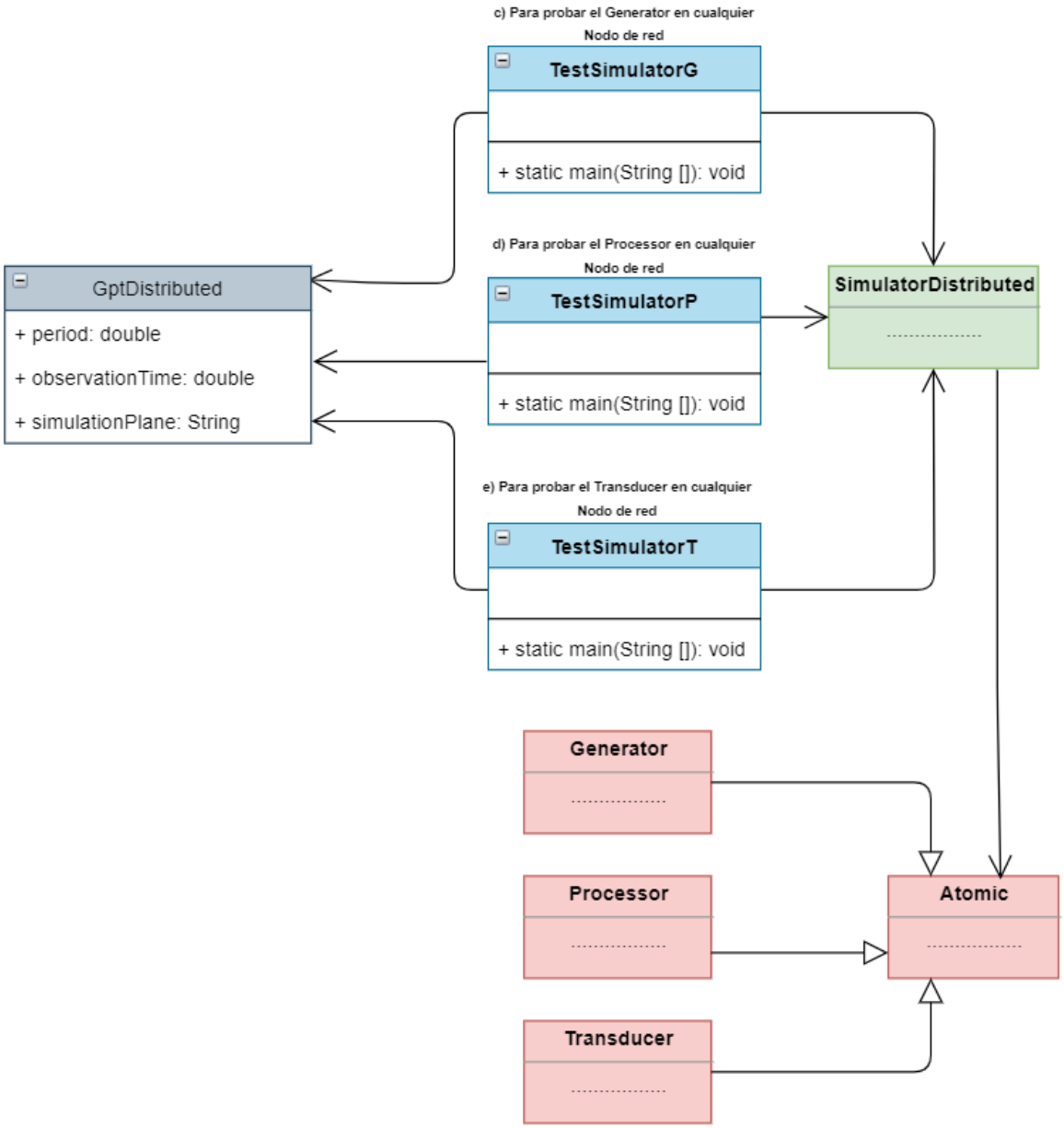

***Figura 3.11**: Clases independientes para ejecutar el simulador de Generator, Processor y Transducer.*

Antes de la ejecución de la simulación distribuida, los elementos de código descritos hasta este punto deberían estar dispuestos en una estructura de clases como la que se muestra en la figura 3.12, en donde se observa que cada elemento de la simulación es independiente en su ejecución y por tanto puede ser desplegada y ejecutada en cualquier nodo dentro de una red interconectada:

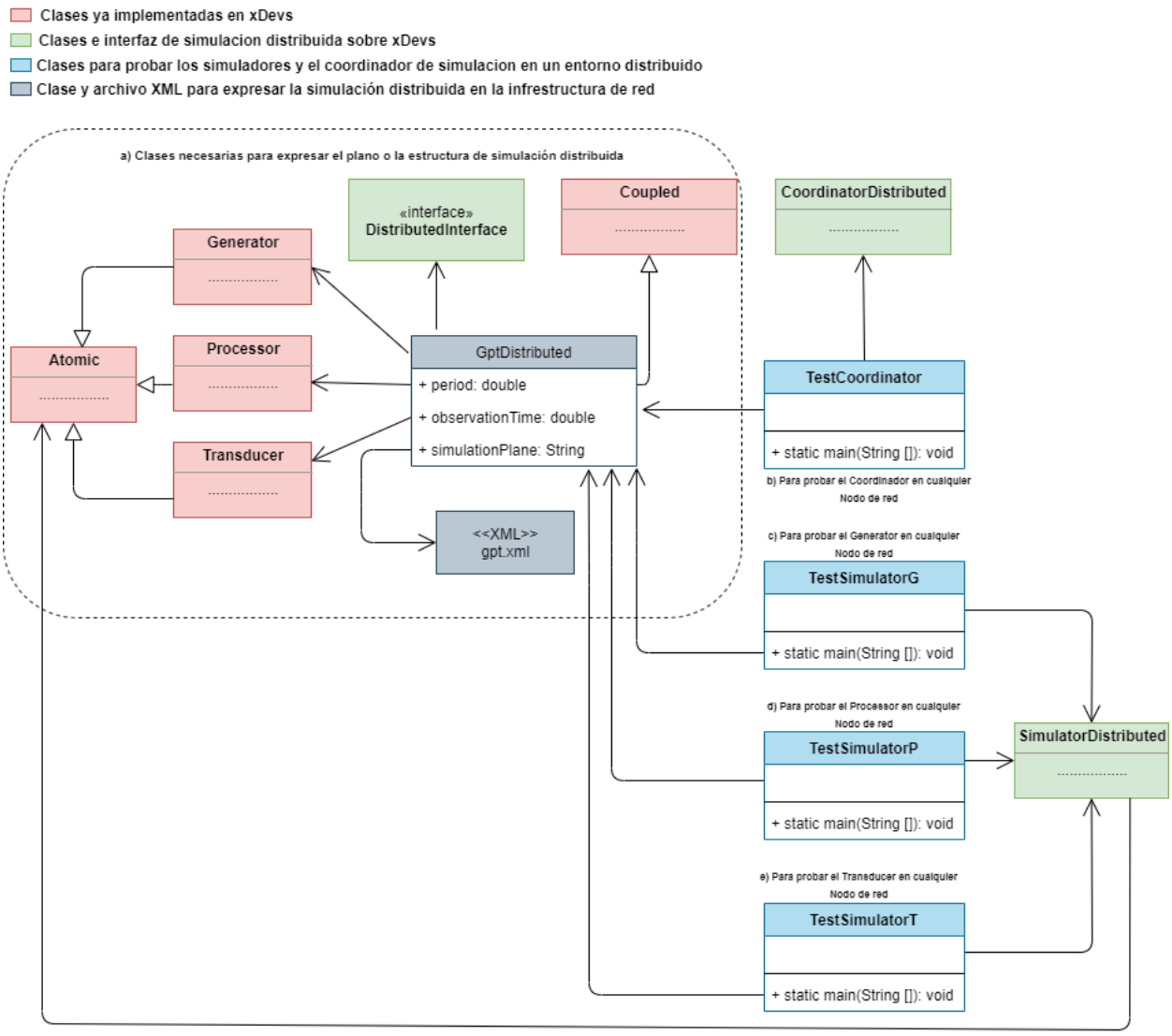

***Figura 3.12**: Disposición de las clases antes de ejecutar la simulación distribuida del modelo EF-P.*

Por último y para ejecutar de manera preliminar el funcionamiento de la solución de la simulación distribuida bajo xDEVS, se muestran los resultados de la simulación del modelo EF-P de manera local, aunque distribuida, es decir en un ordenador usando la ip 127.0.0.0 y puertos diferentes para cada simulador y coordinador:

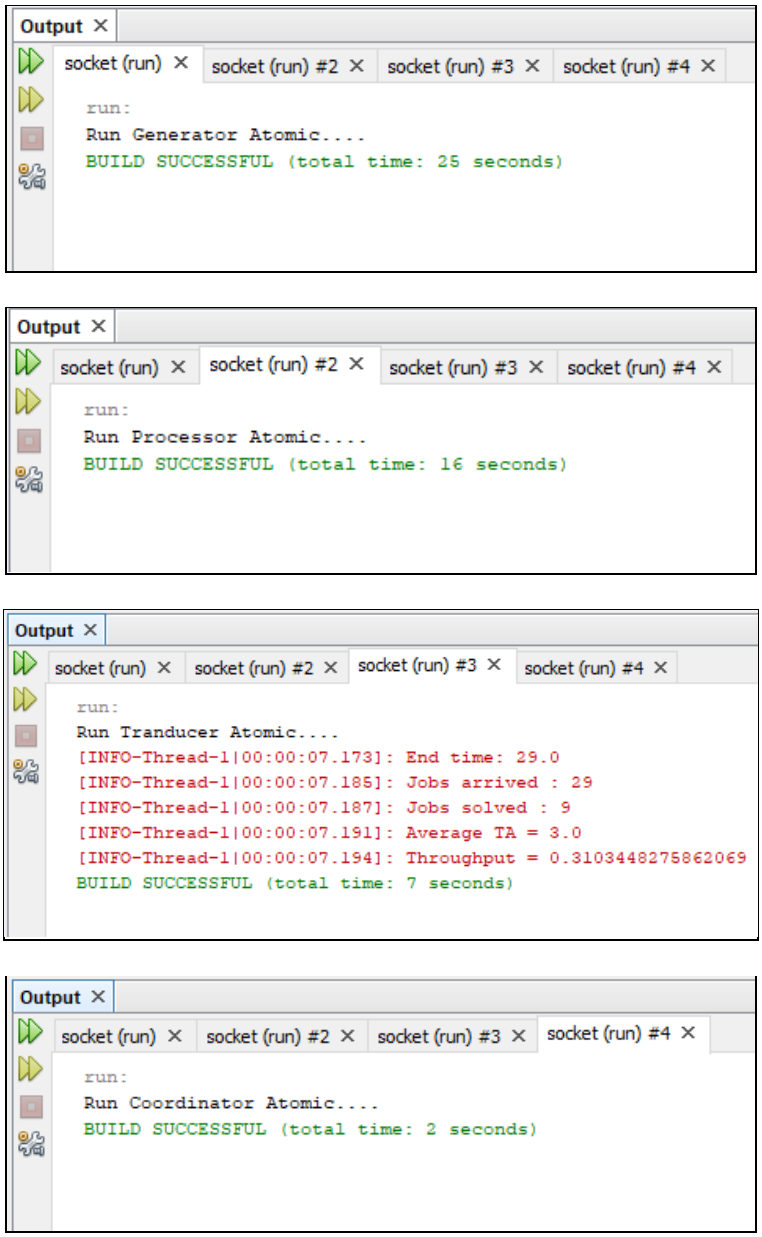

***Figura 3.13**: Ejecución del modelo distribuido en los tres simuladores correspondientes a los componentes atómicos: Generator, Processor y Transducer, además del Coordinator.*

3.4 Tipos de despliegue sobre la nueva arquitectura de simulación distribuida

Una vez comprobado el funcionamiento de la versión distribuida de manera local, se pasa a demostrar la versatilidad del diseño propuesto sobre tres tipos de

arquitectura distribuida en las que es compatible: máquinas virtuales, contenedores y clústeres.

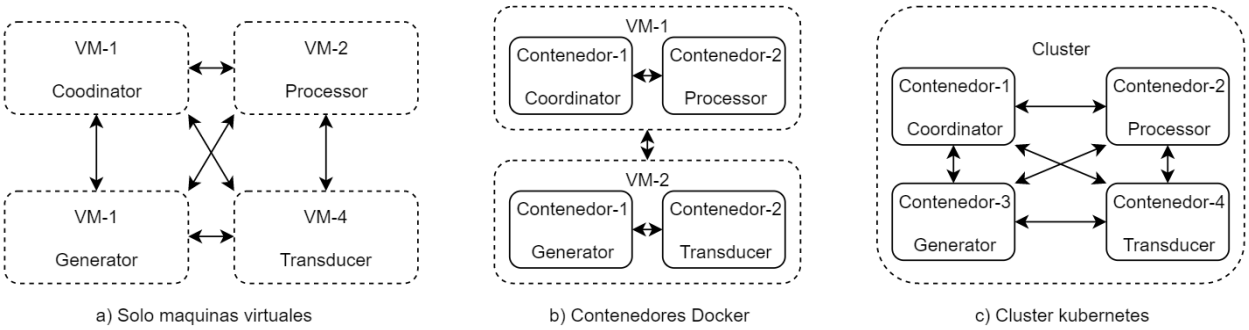

***Figura 3.14**: Arquitecturas propuestas para el despliegue de la simulación en la nube.*

Estas arquitecturas son posibles y factibles de desplegarlas en la nube usando los servicios proporcionados por los proveedores de infraestructura y que ocupan notoriedad a la fecha: Google, Amazon y Microsoft Azure entre otros, y cuyos servicios se parecen o al menos usan herramientas estándar de virtualización como Docker y Kubernetes. En el presente trabajo se decidió usar los servicios de Google Cloud Platform [23] debido a la factibilidad de usar diferentes recursos sin coste, específicos para probar servicios y conceptos en la nube (Nivel gratuito con durante 12 meses y con saldo inicial de 300 USD para utilizar en cualquier servicio de Google Cloud).

Antes de proceder con el despliegue de modelo EF-P en la nube, es necesario empaquetar el código en un solo ejecutable para que este sea fácilmente copiado en cada uno de los contenedores, máquinas virtuales o nodos que se crearán en la nube. Para ello es necesario crear una clase única que permita desde línea de comando pasarle al método principal los datos necesarios para entender la disposición de los nodos en la red y los modelos. Por ejemplo, a continuación, se muestra el código de una clase que cubre los requisitos de ejecutar cualquier componente del modelo EF-P en cualquier máquina sin necesidad de compilar el software para cada escenario:

```java
/**
 *
 * @author Almendras
 *
 * Arguments:
 * args[0] = Type of node: "Atomic" or "Simulator" (*)
 * args[1] = Name (*)
 * args[2] = Period (*)
 * args[3] = Observation Time (*)
 * args[4] = XML name (*)
 * args[5] = Simulation Plane. Example: C;IP;Port1;Port2#G;IP;Port1;Port2#P;IP;Port1;Port2#T;IP;Port1;Port2
 * args[6] = IP (-)
 * args[7] = Primary Port (-)
 * args[8] = Secondary Port (-)
 *
 * (*) = Required for all
 * (-) = Optional for Simulator
 */
public class Node {
    public static void createXmlFile(String importantData, String XMLName){
        // ......................
    }

    public static void main(String[] args) {
        DevsLogger.setup(Level.INFO);
        createXmlFile(args[5], args[4]);
        GptDistributed gpt = new GptDistributed(args[1], Integer.parseInt(args[2]), Integer.parseInt(args[3]), args[4]);
        SimulatorDistributed p;
        CoordinatorDistributed c;
        if( args[0].compareTo("Atomic")==0 ){
            p = new SimulatorDistributed(gpt, args[6],Integer.parseInt(args[7]),Integer.parseInt(args[8]));
        } else if( args[0].compareTo("Simulator")==0){
            c = new CoordinatorDistributed(gpt);
            c.initialize();
            c.simulate(Long.MAX_VALUE);
            c.exit();
        }
    }
}
```

***Figura 3.15**: Node.java*

Esta clase, aparte de recibir ocho argumentos, genera de manera automática un archivo XML que le permite acoplarse de manera fácil a la estructura de simulación distribuida ya creada (Es necesario tener permisos de escritura en el ordenador, contenedor o nodo en donde se ejecute esta clase).

3.5 Despliegue en una arquitectura con solo máquinas virtuales

A continuación, se realiza la explicación del despliegue del modelo EF-P, de cuatro máquinas virtuales utilizando la consola en línea sobre Google Cloud Platform. La idea principal es la de hacer que cada componente del modelo de simulación se encuentre en una sola máquina virtual como se muestra en la siguiente figura:

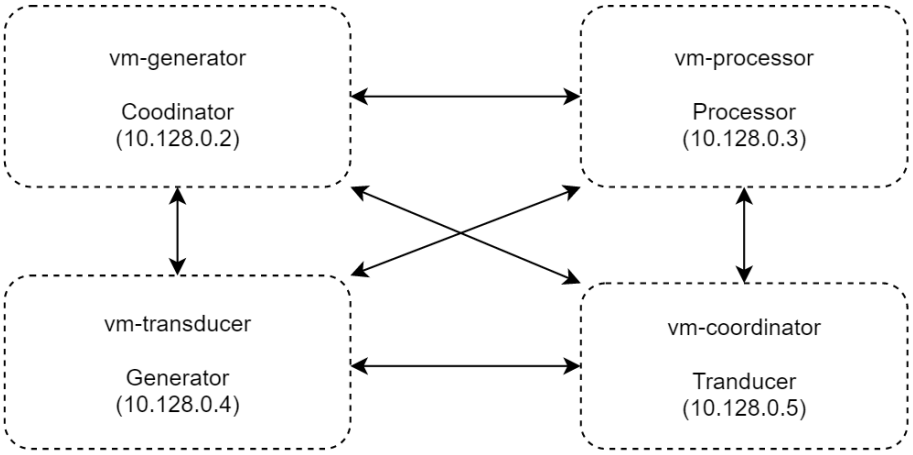

***Figura 3.16**: Arquitectura de máquinas virtuales desplegadas en la nube con el software de simulación distribuida xDEVS.*

Para lograr una simulación distribuida del modelo EF-P en la arquitectura presentada, es necesario seguir mediante comandos de Google Cloud Shell los pasos siguientes:

- Inicializar el proyecto con el cual se trabajará en la nube. Para ello es necesario contar con una cuenta activa en Google Cloud, además de un proyecto creado con la "Compute Engine API" activada.

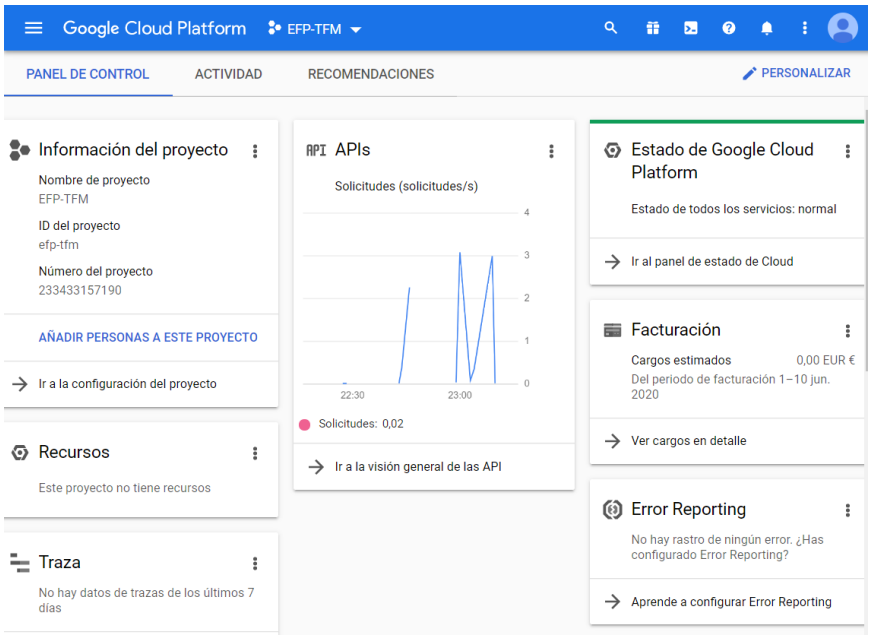

***Figura 3.17**: Consola Google Cloud habilitada para realizar la prueba de la arquitectura DEVS distribuida en la nube.*

- Crear las máquinas virtuales/Instancias con un sistema operativo determinado (En el caso del proyecto ordenadores virtuales con sistema operativo Linux, software Java Openjdk y una dirección de red IP). Por ejemplo, la instancia o máquina virtual que ejecutará el simulador conteniendo el modelo atómico Generator:

```
gcloud compute instances create vm-generator \
    --project=efp-tfm \
    --image-family ubuntu-1804-lts \
    --image-project ubuntu-os-cloud \
    --subnet=default \
    --private-network-ip=10.128.0.2 \
    --metadata-from-file startup-script=install.sh
```

- Copiar el software de simulación distribuida (archivo socket.jar) en cada una de las instancias además del archivo *.sh que contiene los comandos para ejecutar los simuladores usando un scrip bash para ejecutar los simuladores y coordinador. Por ejemplo, para el simulador que ejecutará el modelo atómico Generator:

```
gcloud compute scp socket.jar vm-generator:/tmp/socket.jar
gcloud compute scp generator.sh vm-generator:/tmp/generator.sh
```

- Ejecutar de manera remota en cada una de las instancias, el software jar de cada simulador y coordinador mediante una conexión ssh. Por ejemplo, para lanzar la instancia del simulador que simula el modelo atómico Generator se ejecuta:

```
gcloud compute ssh \
    --project efp-tfm root@vm-generator \
    --command="bash /tmp/generator.sh"
```

- Para verificar los resultados de la ejecución de cada simulador/coordinador (viendo el contenido de cada archivo logger.log), además de comprobar que los archivos jar y sh se copiaron correctamente en cada máquina virtual, se puede realizar una conexión ssh mediante un cliente como putty. Por ejemplo, para realizar la conexión al simulador que contiene el modelo atómico Generator se puede ejecutar desde gcloud el comando:

gcloud compute ssh --project efp-tfm root@vm-generator

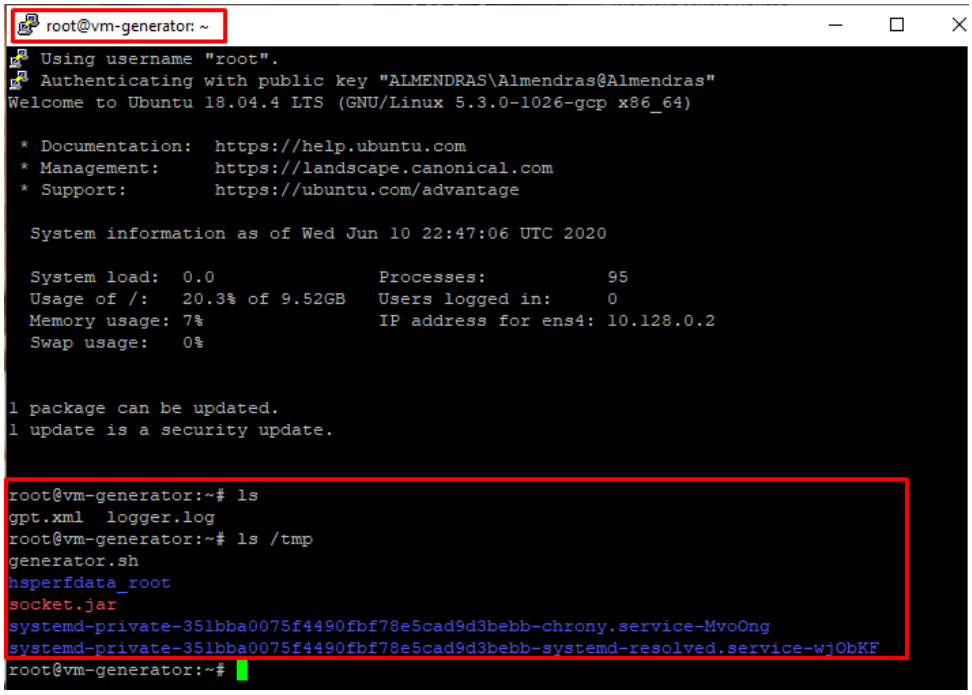

***Figura 3.18**: Cliente ssh que muestra el contenido de la máquina virtual que contiene el simulador del modelo atómico Generator.*

- Borrar las máquinas virtuales creadas para evitar facturar más tiempo. por ejemplo, para eliminar las cuatro instancias creadas ejecutar:

gcloud compute instances delete \

vm-generator vm-processor vm-transductor vm-coordinator

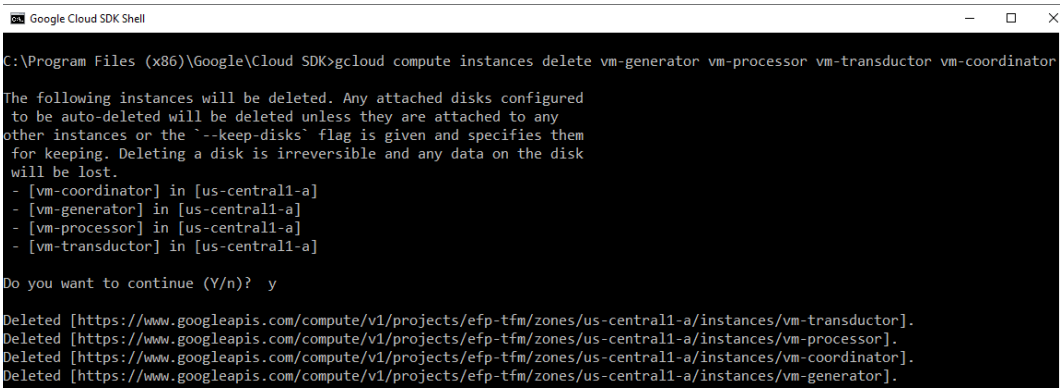

***Figura 3.19**: Eliminación de las máquinas virtuales desde Google Cloud Shell.*

El script completo de comandos en Google Cloud Shell para tener un despliegue concreto del modelo acoplado EF-P distribuido en la nube, es el que se muestra en la figura 3.20 y su resultado puede observarse en la figura 3.21:

```
-- Initialization
gcloud config configurations activate tfm2020
gcloud config set project efp-tfm
gcloud auth configure-docker
gcloud config set compute/zone us-central1-a
gcloud config set compute/region us-central1

-- Create virtual machines
gcloud compute instances create vm-generator --project=efp-tfm --image-family ubuntu-1804-lts \
    --image-project ubuntu-os-cloud --subnet=default --private-network-ip=10.128.0.2 --metadata-from-file startup-script=install.sh
gcloud compute instances create vm-processor --project=efp-tfm --image-family ubuntu-1804-lts \
    --image-project ubuntu-os-cloud --subnet=default --private-network-ip=10.128.0.3 --metadata-from-file startup-script=install.sh
gcloud compute instances create vm-transductor --project=efp-tfm --image-family ubuntu-1804-lts \
    --image-project ubuntu-os-cloud --subnet=default --private-network-ip=10.128.0.4 --metadata-from-file startup-script=install.sh
gcloud compute instances create vm-coordinator --project=efp-tfm --image-family ubuntu-1804-lts \
    --image-project ubuntu-os-cloud --subnet=default --private-network-ip=10.128.0.5 --metadata-from-file startup-script=install.sh

-- Copy main files to instance: jar and run.sh
gcloud compute scp socket.jar vm-generator:/tmp/socket.jar
gcloud compute scp generator.sh vm-generator:/tmp/generator.sh
gcloud compute scp socket.jar vm-processor:/tmp/socket.jar
gcloud compute scp processor.sh vm-processor:/tmp/processor.sh
gcloud compute scp socket.jar vm-transductor:/tmp/socket.jar
gcloud compute scp transductor.sh vm-transductor:/tmp/transductor.sh
gcloud compute scp socket.jar vm-coordinator:/tmp/socket.jar
gcloud compute scp coordinator.sh vm-coordinator:/tmp/coordinator.sh

-- Connect to VM with root user and run simulators and coordinator
gcloud compute ssh --project efp-tfm root@vm-generator --command="bash /tmp/generator.sh"
gcloud compute ssh --project efp-tfm root@vm-processor --command="bash /tmp/processor.sh"
gcloud compute ssh --project efp-tfm root@vm-transductor --command="bash /tmp/transductor.sh"
gcloud compute ssh --project efp-tfm root@vm-coordinator --command="bash /tmp/coordinator.sh"

-- Connect to VM and to see the results
gcloud compute ssh --project efp-tfm root@vm-generator
gcloud compute ssh --project efp-tfm root@vm-processor
gcloud compute ssh --project efp-tfm root@vm-transductor
gcloud compute ssh --project efp-tfm root@vm-coordinator

-- Delete VM's
gcloud compute instances delete vm-generator vm-processor vm-transductor vm-coordinator
```

***Figura 3.20**: Comandos en Google Cloud Shell para el despliegue del EF-P en máquinas virtuales en la nube.*

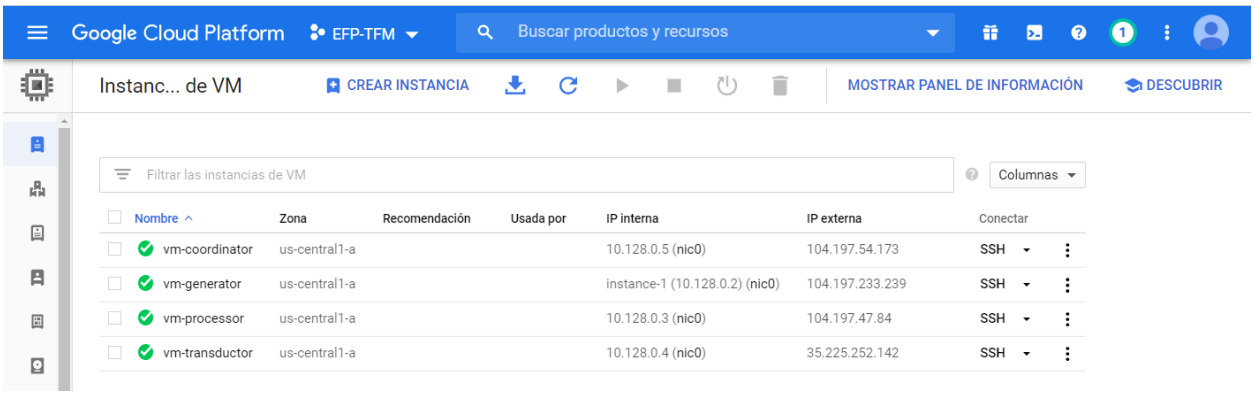

***Figura 3.21**: Máquinas virtuales vistas desde la consola de Google Cloud.*

Asimismo, esta es la salida del despliegue en la nube de las cuatro máquinas virtuales después de lanzar el software de simulación distribuida en cada una de ellas:

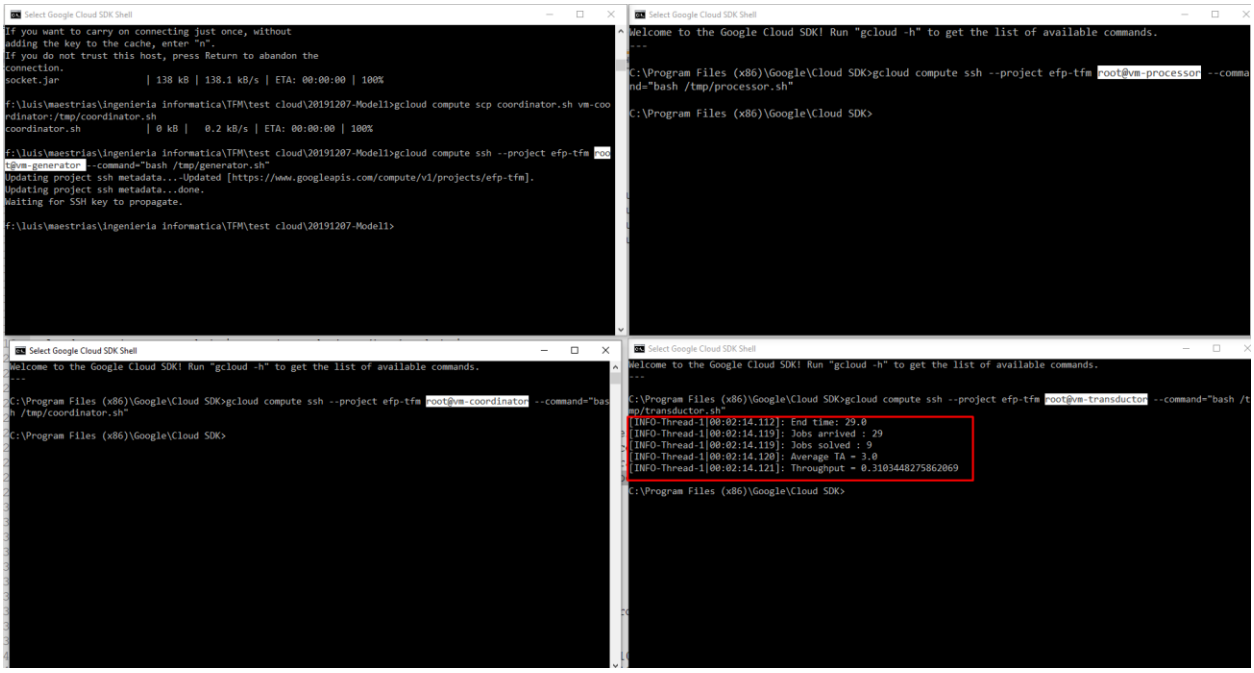

***Figura 3.22**: Despliegue del EF-P distribuido en cuatro máquinas virtuales en la nube.*

3.6 Despliegue en una arquitectura de contenedores Docker

A continuación, se realizará la explicación del despliegue del modelo EF-P utilizando cuatro contenedores Docker [24]. La idea es que cada componente del modelo de simulación se encuentre en un contenedor y este se encuentre en una máquina virtual. Para el caso concreto se usarán dos máquinas virtuales creadas en la

nube y cada una de ellas contendrá a dos contenedores como se muestra en la figura siguiente:

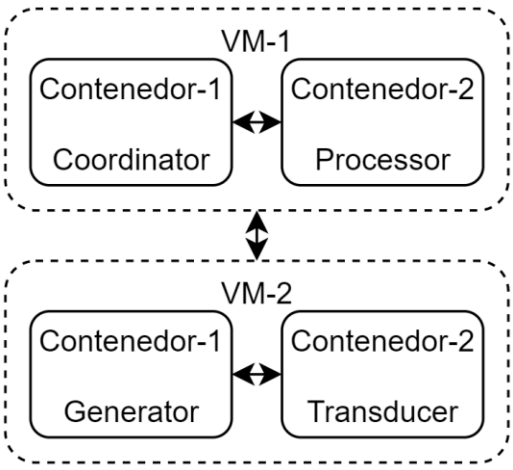

***Figura 3.23**: Arquitectura de contenedores Docker desplegadas en la nube con el software de simulación distribuida xDEVS.*

Para lograr tener corriendo el modelo EF-P en la arquitectura presentada, es necesario seguir mediante comandos de Google Cloud Shell los siguientes pasos:

- Inicializar el proyecto con el cual se trabajará en la nube. Se debe activar la "Google Container Registry API" para el proyecto.

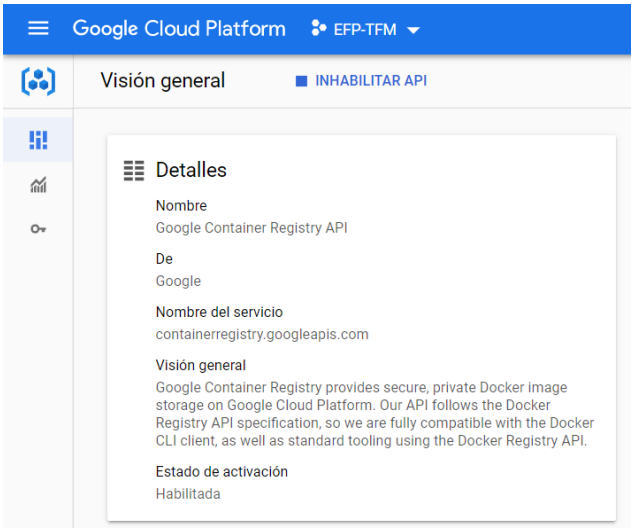

***Figura 3.24**: Google Container Registry API, habilitada para el proyecto.*

- Crear una imagen en la nube asociada al proyecto conteniendo el software de simulación distribuido a partir de un Docker file. Se puede

utilizar una línea de comando de Google Cloud Shell (GCS) como la siguiente:

gcloud builds submit --tag gcr.io/efp-tfm/img-xdevs:v1 .

El comando anterior necesitará que en el directorio en donde se ejecute el comando exista un archivo Docker de instalación de sistema operativo Linux Ubuntu, Open Java, Utilitarios de red, editor de texto y el archivo JAR que contiene el software del sistema de simulación distribuido.

- Crear las dos máquinas virtuales en la nube usando un comando de GCS como el siguiente:

  **gcloud compute instances create-with-container vm-server1 **

  **--container-image gcr.io/efp-tfm/img-xdevs:v1 **

  --subnet=default --private-network-ip=10.128.0.2

  Se hace notar que estas máquinas virtuales se crean utilizando la imagen Docker que se subió a la nube, esto con la finalidad de usar un sistema operativo preparado con solo las herramientas necesarias y otro para probar que efectivamente las imágenes están correctas y con el software necesario para la simulación.

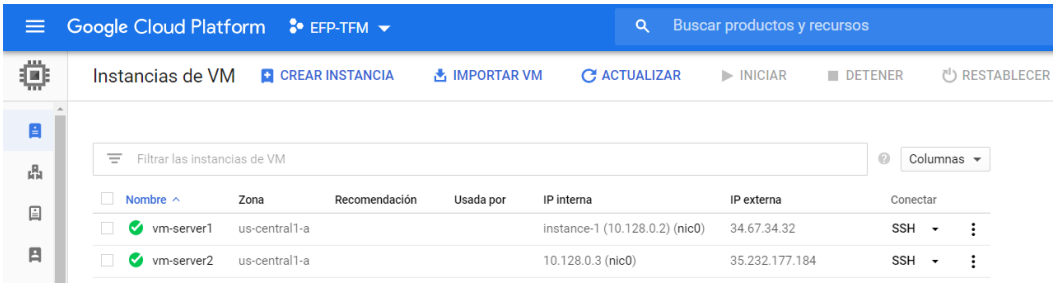

***Figura 3.25**: Dos máquinas virtuales creadas en la nube y asociadas al proyecto.*

- Conectarse mediante SSH a la máquina virtual vm-server1, habilitar el servidor Docker y crear los dos contenedores que ejecutarán la simulación del modelo atómico Generator y Processor. El contenedor puede ser creado con un comando como el siguiente:

  **docker run -itd --net net-xdevs --ip 192.168.2.2 **

**--hostname=Generator **

--name=xdevs-Generator gcr.io/efp-tfm/img-xdevs:v1

Se hace notar que el contenedor se crea a partir de la imagen que se subió a la nube, además de usar una red IP local mediante el cual los contenedores se comunicaran.

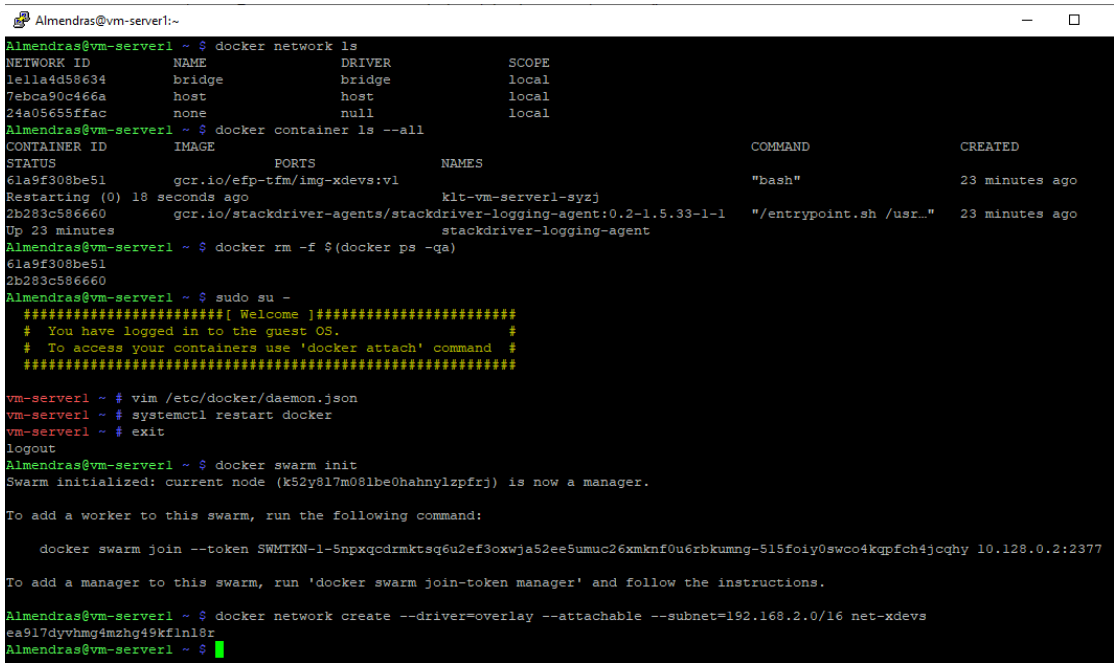

***Figura 3.26**: Configuración de la máquina virtual vm-server1.*

En las líneas finales de la imagen anterior, se observa la creación explícita una red interna (**192.168.2.0**) que permitirá a vm-server1 ser el manejador de peticiones de red entre las dos máquinas virtuales (10.128.0.0) que a su vez van a permitir la comunicación de los contenedores en una red local.

- Conectarse mediante SSH a la máquina virtual vm-server2, habilitar el servidor Docker y crear los dos contenedores que ejecutarán la simulación del modelo atómico Transducer y del Coordinador de la simulación Coordinator.

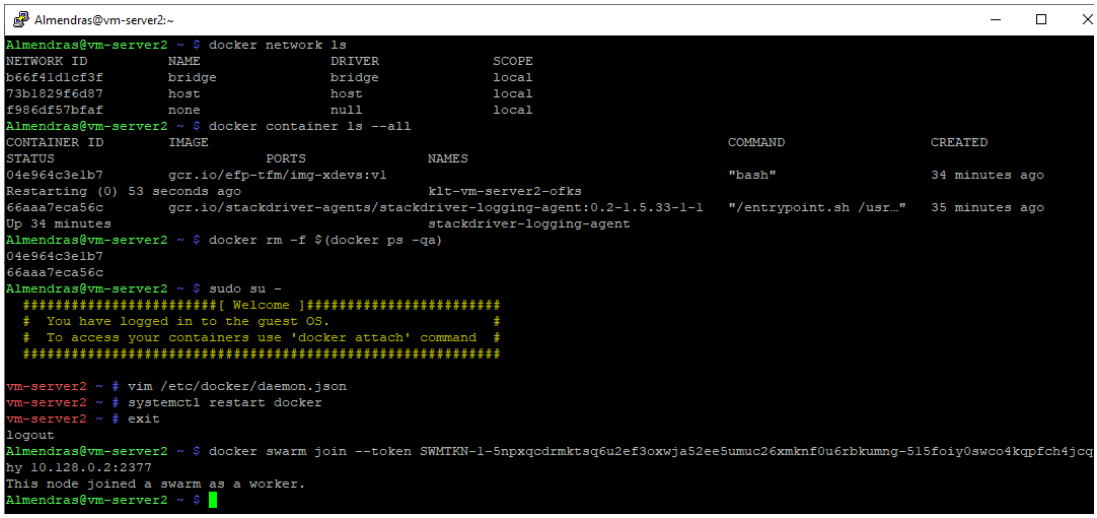

***Figura 3.27**: Configuración de la máquina virtual vm-server2.*

En la línea final de la imagen anterior, se observa que la máquina virtual vm-server2 se añade junto con sus contenedores a la red local 192.168.2.0 para permitir la comunicación con los contenedores de VM-server1 (por seguridad se usa un token generado para que todas las máquinas virtuales que se agreguen a la red local creada).

- Ejecutar los contenedores Docker contenidas en las máquinas virtuales VM-server1 y VM-server2, con el comando Docker similar al siguiente:

  **docker run -itd --net net-xdevs --ip 192.168.2.2 **

  **--hostname=Generator --name=xdevs-Generator **

  gcr.io/efp-tfm/img-xdevs:v1

- Ejecutar el software de simulación distribuida contenida en cada uno de los contenedores Docker, usando estos dos comandos desde una BASH SHELL dentro de cada máquina virtual (usar un cliente SSH):

  docker attach xdevs-Generator

java -jar /socket.jar Atomic Node1 1 30 gpt.xml "C;192.168.2.5;5000;6000#G;192.168.2.2;5001;6001#P;192.168.2.3;5002;6002 #T;192.168.2.4;5003;6003" 192.168.2.2 5001 6001

- Verificar los resultados de la ejecución.
- Borrar las máquinas virtuales creadas con un comando de GCS similar al siguiente:

gcloud compute instances delete vm-server1 vm-server2

El conjunto de comandos Google Cloud Shell y Docker que permiten realizar el despliegue de la simulación EF-P usando contenedores Docker se muestra a continuación:

```
-- Initialization
gcloud config configurations activate tfm2020
gcloud config set project efp-tfm
gcloud auth configure-docker
gcloud config set compute/zone us-central1-a
gcloud config set compute/region us-central1

-- Create XDEVS image starting from a docker file
gcloud builds submit --tag gcr.io/efp-tfm/img-xdevs:v1 .

-- Create virtual machines with XDEVS image created
gcloud compute instances create-with-container vm-server1 --container-image gcr.io/efp-tfm/img-xdevs:v1 \
    --subnet=default --private-network-ip=10.128.0.2
gcloud compute instances create-with-container vm-server2 --container-image gcr.io/efp-tfm/img-xdevs:v1 \
    --subnet=default --private-network-ip=10.128.0.3

-- Connect with SSH to vm-server's and Create container in vm-server1:
# gcloud compute --project efp-tfm ssh vm-server1
# docker network ls
# docker container ls --all
# docker rm -f $(docker ps -qa)
# -- Set up the swarm for MANAGER
# -- you should change to: "live-restore": false
# sudo su -
# vim /etc/docker/daemon.json
# systemctl restart docker
# exit
docker swarm init
docker network create --driver=overlay --attachable --subnet=192.168.2.0/16 net-xdevs

-- Connect with SSH to vm-server's and Create container in vm-server2:
# gcloud compute --project efp-tfm ssh vm-server2
# docker network ls
# docker container ls --all
# docker rm -f $(docker ps -qa)
# -- Set up the swarm for WORKER
# -- you should change to: "live-restore": false
# sudo su -
# vim /etc/docker/daemon.json
# systemctl restart docker
# exit
docker swarm join --token SWMTKN-1-5npxqcdrmktsq6u2ef3oxwja52ee5umuc26xmknf0u6rbkumng-515foiy0swco4kqpfch4jcqhy 10.128.0.2:2377


-- Launch docker containers from vm-server1
docker run -itd --net net-xdevs --ip 192.168.2.2 --hostname=Generator --name=xdevs-Generator gcr.io/efp-tfm/img-xdevs:v1
docker run -itd --net net-xdevs --ip 192.168.2.3 --hostname=Processor --name=xdevs-Processor gcr.io/efp-tfm/img-xdevs:v1
#docker network inspect net-xdevs

-- Launch docker containers from vm-server2
docker run -itd --net net-xdevs --ip 192.168.2.4 --hostname=Transductor --name=xdevs-Transductor gcr.io/efp-tfm/img-xdevs:v1
docker run -itd --net net-xdevs --ip 192.168.2.5 --hostname=Coordinator --name=xdevs-Coordinator gcr.io/efp-tfm/img-xdevs:v1
#docker network inspect net-xdevs

-- Launch simulators and coordinator
# gcloud compute --project efp-tfm ssh vm-server1
docker attach xdevs-Generator
java -jar /socket.jar Atomic Node1 1 30 gpt.xml \
    "C;192.168.2.5;5000;6000#G;192.168.2.2;5001;6001#P;192.168.2.3;5002;6002#T;192.168.2.4;5003;6003" 192.168.2.2 5001 6001

# gcloud compute --project efp-tfm ssh vm-server1
docker attach xdevs-Processor
java -jar /socket.jar Atomic Node2 1 30 gpt.xml \
    "C;192.168.2.5;5000;6000#G;192.168.2.2;5001;6001#P;192.168.2.3;5002;6002#T;192.168.2.4;5003;6003" 192.168.2.3 5002 6002

# gcloud compute --project efp-tfm ssh vm-server2
docker attach xdevs-Transductor
java -jar /socket.jar Atomic Node3 1 30 gpt.xml \
    "C;192.168.2.5;5000;6000#G;192.168.2.2;5001;6001#P;192.168.2.3;5002;6002#T;192.168.2.4;5003;6003" 192.168.2.4 5003 6003

# gcloud compute --project efp-tfm ssh vm-server2
docker attach xdevs-Coordinator
java -jar /socket.jar Simulator Node4 1 30 gpt.xml \
    "C;192.168.2.5;5000;6000#G;192.168.2.2;5001;6001#P;192.168.2.3;5002;6002#T;192.168.2.4;5003;6003" 192.168.2.5 5000 6000

-- Delete VM's
gcloud compute instances delete vm-server1 vm-server2
```

***Figura 3.28**: Comandos en Google Cloud Shell y Docker para el despliegue del EF-P en contenedores Docker.*

Asimismo, esta es la salida del despliegue en la nube de las cuatro máquinas virtuales después de correr el software de simulación distribuida en cada una de ellas:

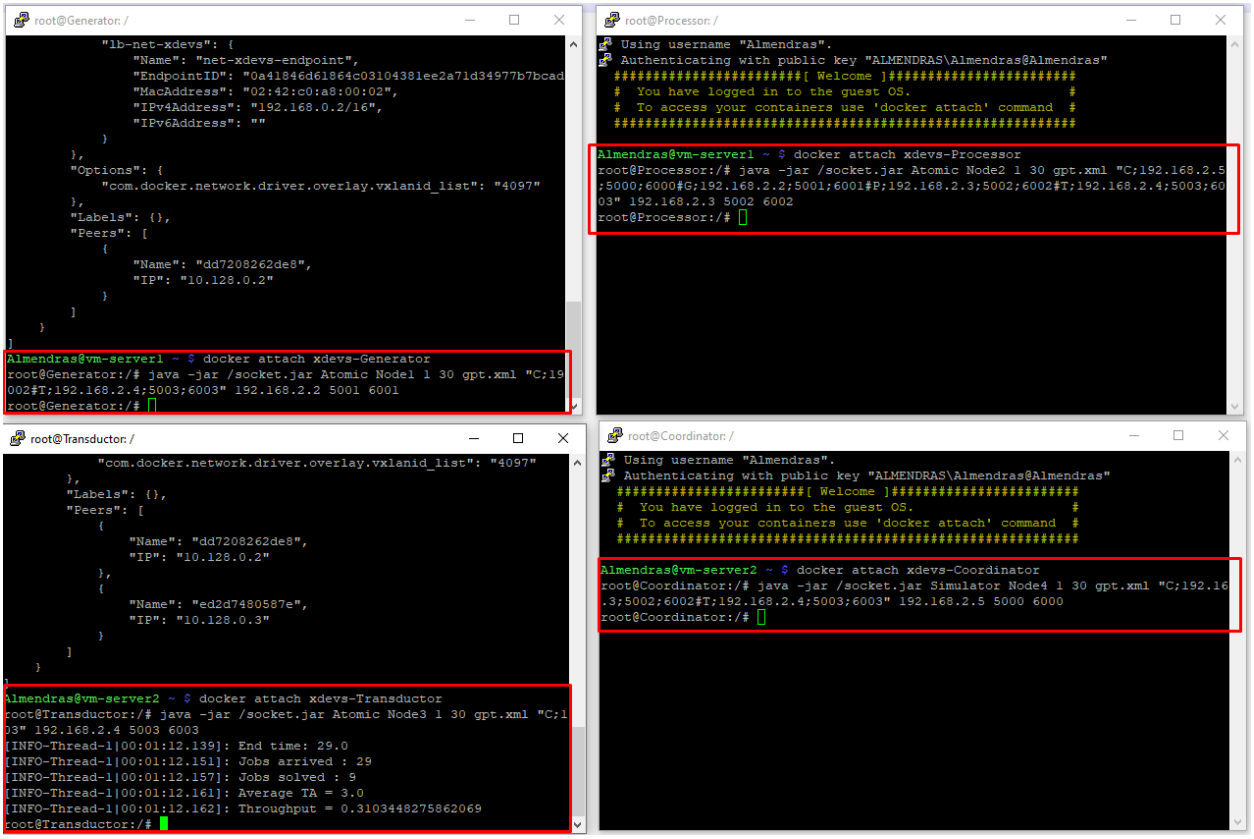

***Figura 3.29**: Despliegue del EF-P distribuido en cuatro contenedores Docker en la nube.*

3.7 Despliegue en una arquitectura de cluster con Kubernetes

En este apartado se explica el despliegue del modelo EF-P utilizando una infraestructura de clúster con Kubernetes [25] en la nube; para esta prueba se crea un clúster de tres nodos (equivalentes a tres máquinas virtuales), con cuatro contenedores. Cada contenedor Docker lleva consigo un elemento del modelo de simulación distribuido tal como se muestra en la figura siguiente:

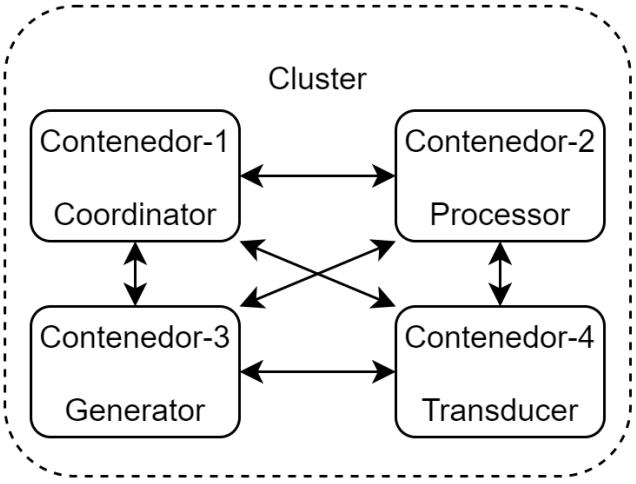

***Figura 3.30**: Arquitectura de clúster Kubernetes desplegadas en la nube con el software de simulación distribuida xDEVS.*

Para lograr ejecutar el modelo EF-P en la arquitectura de clúster, es necesario seguir los siguientes pasos:

- Inicializar el proyecto con el cual se trabajará en la nube. Además, se debe preparar los archivos de creación de Pods[3] en formato YAML con un contenido similar al siguiente:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: transductor
spec:
  containers:
  - name: shell
    image: gcr.io/efp-tfm/img-xdevs:v1
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
```

  Como cada Pod necesita de al menos un contenedor Docker, se deberá tener lista la imagen en la nube con el software necesario para ejecutar el simulador/coordinador distribuido. Finalmente se deberá habilitar la "**Kubernetes Engine API**".

[3] Los **pods** son los objetos más pequeños y básicos que se pueden implementar en Kubernetes; representan una instancia única de un proceso en ejecución en el clúster y contienen uno o más contenedores Docker.

- Crear un clúster de contenedores (con 3 nodos de hardware) usando Kubernetes mediante los siguientes comandos GCS:

  gcloud container clusters create cluster-xdevs --num-nodes=3

  gcloud container clusters get-credentials cluster-xdevs

  Después de ejecutar los anteriores comandos se deberá contar con una infraestructura de hardware de tres nodos o tres máquinas virtuales como la siguiente:

```
f:\luis\maestrias\ingenieria informatica\TFM\test cloud\20191213-Model3>gcloud container clusters list
NAME          LOCATION       MASTER_VERSION  MASTER_IP      MACHINE_TYPE   NODE_VERSION    NUM_NODES  STATUS
cluster-xdevs  us-central1-a  1.14.10-gke.36  104.197.78.71  n1-standard-1  1.14.10-gke.36  3          RUNNING

f:\luis\maestrias\ingenieria informatica\TFM\test cloud\20191213-Model3>gcloud compute instances list
NAME                                          ZONE           MACHINE_TYPE   PREEMPTIBLE  INTERNAL_IP  EXTERNAL_IP      STATUS
gke-cluster-xdevs-default-pool-8bf12b69-0x33  us-central1-a  n1-standard-1               10.128.0.7   34.67.34.32      RUNNING
gke-cluster-xdevs-default-pool-8bf12b69-1mz7  us-central1-a  n1-standard-1               10.128.0.6   35.232.177.184   RUNNING
gke-cluster-xdevs-default-pool-8bf12b69-c5tw  us-central1-a  n1-standard-1               10.128.0.8   35.225.94.124    RUNNING
```

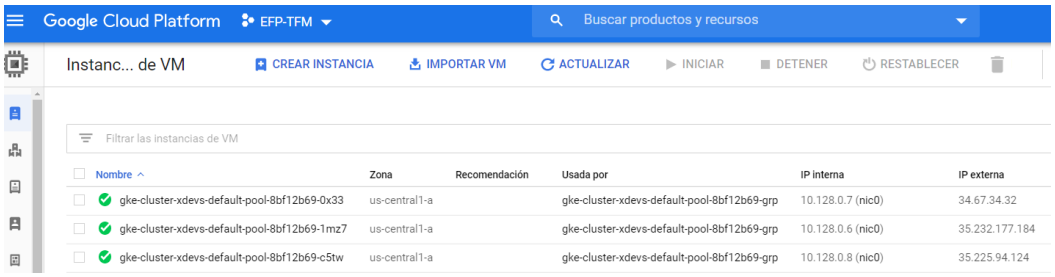

***Figura 3.31**: Vista resumen (GCS/Web Console) del clúster generado con tres máquinas virtuales.*

- Sobre el clúster generado, se debe crear los cuatro pods usando un contenedor Docker por cada pod. Estos contenedores serán generados con el comando Kubernetes similar al siguiente:

  kubectl create -f podG.yaml

- Conectarse al pod y ejecutar el software de simulación distribuida. La conexión y la ejecución del software pueden realizarse con los siguientes dos comandos:

  kubectl exec generator -c shell -i -t – bash

  **java -jar /socket.jar Atomic Node1 1 30 gpt.xml **

**"C;10.60.1.5;5000;6000#G;10.60.2.3;5001;6001#P;10.60.1.4;5002;6002#T;10.60.2.4;5003;6003" **

10.60.2.3 5001 6001

- Verificar los resultados de la ejecución.
- Borrar el clúster creado con el comando GCS:

gcloud container clusters delete cluster-xdevs

El conjunto de comandos en Google Cloud Shell y Kubernetes para tener un despliegue EF-P sobre un clúster en la nube se muestra a continuación:

```
-- Initialization
gcloud config configurations activate tfm2020
gcloud config set project efp-tfm
gcloud auth configure-docker
gcloud config set compute/zone us-central1-a
gcloud config set compute/region us-central1

-- Create Kubernetes Container Cluster
gcloud container clusters create cluster-xdevs --num-nodes=3
# gcloud container clusters list
# gcloud compute instances list
# kubectl describe nodes
gcloud container clusters get-credentials cluster-xdevs

-- Create pod Generator starting from a docker file (10.60.2.3)
# gcloud container clusters get-credentials cluster-xdevs
kubectl create -f podG.yaml
kubectl exec generator -c shell -i -t -- bash
java -jar /socket.jar Atomic Node1 1 30 gpt.xml \
    "C;10.60.1.5;5000;6000#G;10.60.2.3;5001;6001#P;10.60.1.4;5002;6002#T;10.60.2.4;5003;6003" 10.60.2.3 5001 6001

-- Create pod Processor starting from a docker file (10.60.1.4)
# gcloud container clusters get-credentials cluster-xdevs
kubectl create -f podP.yaml
kubectl exec processor -c shell -i -t -- bash
java -jar /socket.jar Atomic Node2 1 30 gpt.xml \
    "C;10.60.1.5;5000;6000#G;10.60.2.3;5001;6001#P;10.60.1.4;5002;6002#T;10.60.2.4;5003;6003" 10.60.1.4 5002 6002

-- Create pod Transducer starting from a docker file (10.60.2.4)
# gcloud container clusters get-credentials cluster-xdevs
kubectl create -f podT.yaml
kubectl exec transductor -c shell -i -t -- bash
java -jar /socket.jar Atomic Node3 1 30 gpt.xml \
    "C;10.60.1.5;5000;6000#G;10.60.2.3;5001;6001#P;10.60.1.4;5002;6002#T;10.60.2.4;5003;6003" 10.60.2.4 5003 6003

-- Create pod Coordinator starting from a docker file (10.60.1.5)
# gcloud container clusters get-credentials cluster-xdevs
kubectl create -f podC.yaml
kubectl exec coordinator -c shell -i -t -- bash
java -jar /socket.jar Simulator Node4 1 30 gpt.xml \
    "C;10.60.1.5;5000;6000#G;10.60.2.3;5001;6001#P;10.60.1.4;5002;6002#T;10.60.2.4;5003;6003" 10.60.1.5 5000 6000

-- Delete cluster-xdevs
gcloud container clusters delete cluster-xdevs
```

***Figura 3.32**: Comandos en Google Cloud Shell y Kubernetes para el despliegue del EF-P distribuido en un clúster.*

Asimismo, esta es la salida del despliegue del EF-P en la nube de los 4 pods sobre tres máquinas virtuales en clúster:

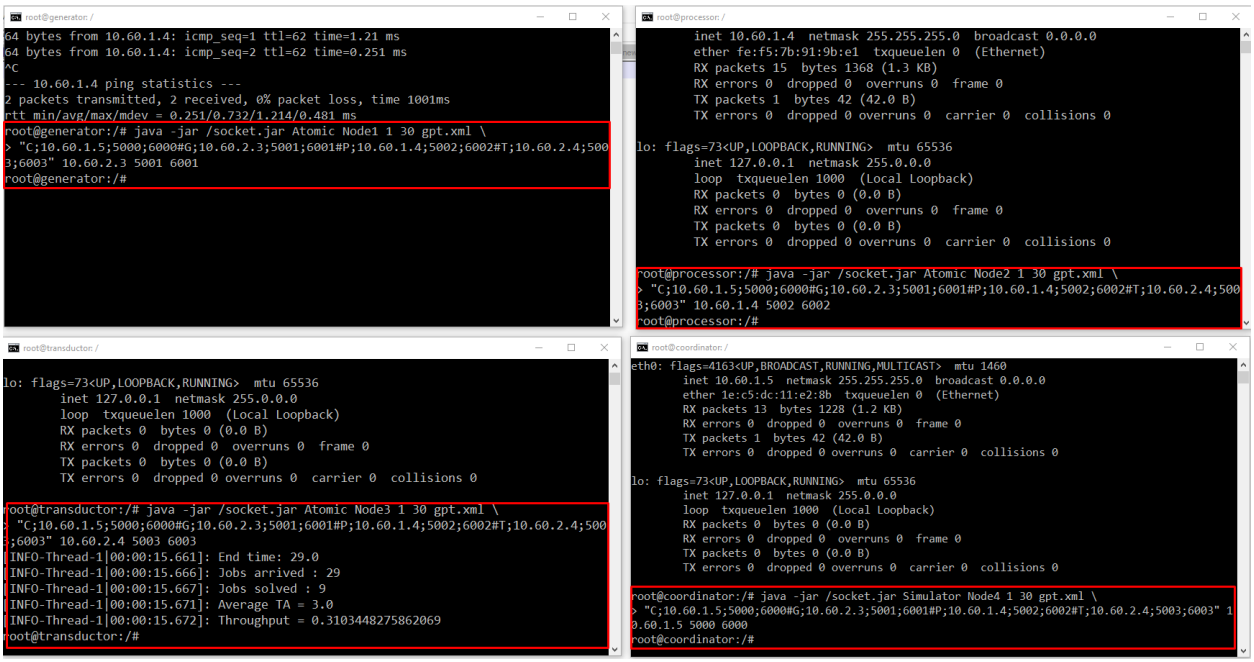

***Figura 3.33**: Salidas de la arquitectura de simulación distribuida usando un clúster de tres nodos y cuatro Pods.*

3.8 Conclusiones

A partir de las pruebas realizadas usando los servicios de Google Cloud Platform y los tipos de infraestructura (IaaS) que se pueden usar en la misma, se puede notar que un contenedor Docker o al menos una imagen Docker siempre está presente y es un elemento fundamental en los servicios que se ofrecen en la nube. Por otro lado, es necesaria una conectividad de red entre máquinas virtuales como también entre los contenedores Docker que cada máquina contiene, para permitir el paso de mensajes entre simuladores y coordinador y lograr la simulación distribuida.

Con el primer escenario, en donde se usan máquinas virtuales desplegadas a partir de una imagen Docker, es posible la utilización de todos los recursos de hardware disponibles en cada máquina virtual para el consumo del o los elementos de simulación (simulador o coordinador) que se estén ejecutando en la misma. Se debe tener en cuenta que los elementos de la simulación que se ejecuten en una máquina virtual lo hacen bajo una versión específica de sistema operativo y software instalado/configurado para uso de todos los procesos que usen dicha máquina.

En cambio, si la necesidad indica que cada simulador o coordinador requiere de una configuración específica de software/librerías o versión de sistema operativo diferente, la segunda arquitectura de simulación con contenedores Docker es la que se puede utilizar. Se debe notar que cada máquina virtual bajo este escenario debe tener instalado un servidor Docker; este permitirá la creación de contenedores que permitirán la independencia requerida. Claro está que al agregar una capa de software adicional a cada máquina este deberá ser tomado en cuenta a la hora de realizar pruebas de rendimiento y su incidencia en la simulación distribuida.

En la arquitectura de máquinas virtuales como en la de contenedores Docker, quien decide dónde se ejecuta cada elemento de la simulación es la persona o el equipo que planifica la ejecución de la simulación distribuida. Al ser de esa manera, también se decide implícitamente acerca de los recursos que cada máquina virtual proporcionará a cada elemento de la simulación que correría sobre su dominio. Si el objetivo es la automatización y autogestión de los recursos de las máquinas virtuales que participan en la simulación, la arquitectura clúster es la sugerida. Esta solo necesita saber cuántas máquinas virtuales o nodos va a tener el clúster (es posible modificar ese número en adelante sin afectar la arquitectura de la simulación ya establecida). En esta arquitectura solo se debe pensar en qué contenedores colocar cada elemento de la simulación distribuida sin preocuparnos de qué maquina se ocupará de cada contenedor.

Cada simulación distribuida puede ejecutarse a simple necesidad, y el coste económico de uso de recursos es por el tiempo que dura la simulación.

El reto posterior está en hacer la automatización y portabilidad de los comandos Docker, Kubernetes y GCS, para que estos puedan funcionar en cualquier proveedor de servicios en la nube, esta tarea podría ser resuelta con la herramienta Terraform [26], muy utilizada en ambientes Devops (Automatización y despliegue a demanda).

Capítulo 4 - Evaluación experimental

Este capítulo comienza con una introducción a un conjunto de modelos estándar denominado DEVStone. Este se usa para evaluar el rendimiento de motores de simulación. A continuación, se presenta la ejecución de un modelo DEVStone, denominado HO, usando los motores de simulación secuencial, paralelo, y distribuido diseñados en este trabajo. Para ello se usa siempre la misma máquina virtual de referencia, configurada en Google Cloud. La ejecución distribuida se presenta en las diferentes arquitecturas disponibles: máquinas virtuales, contenedores Docker y clúster con Kubernetes. Finalmente, el capítulo concluye con varias reflexiones acerca de las pruebas de rendimiento realizadas.

4.1 Evaluación de rendimiento con DEVStone

DEVStone es un conjunto de modelos estándar diseñado para medir el rendimiento de motores de simulación orientados a eventos. En los últimos años ha sido usado especialmente para evaluar simuladores DEVS. Esta herramienta se caracteriza por generar de manera automatizada una variedad de modelos en tamaño y forma. Cada modelo DEVStone se define usando cinco parámetros [27]:

- **Tipo**: Estructura diferente y esquemas de interconexión entre los componentes del modelo.
- **Ancho:** Este parámetro se basa en el número de componentes en cada modelo acoplado intermedio.
- **Profundidad**: El número de niveles en la jerarquía del modelo.
- **Tiempo de transición interno**: El tiempo de ejecución empleado por cada función de transición interna.
- **Tiempo de transición externo**: El tiempo de ejecución empleado por cada función de transición externa.

Se debe tener en cuenta que para consumir los tiempos de ejecución definidos para las funciones de transición interna y externa, DEVStone ejecuta Dhrystone, un benchmark sintético para mantener ocupada la CPU [28].

DEVStone maneja cinco tipos de modelos: **LI** (Low level of interconnections), **HI** (High Input couplings), **HO** (HI model with numerous Outputs), y **HOmod** y **HOmen** (variantes más complejas de HO). Dado que HOmod y HOmen consumen muchos recursos y a menudo son imposibles de ejecutar, el modelo HO será el que se use en este trabajo, que sigue el patrón mostrado en la figura 4.1:

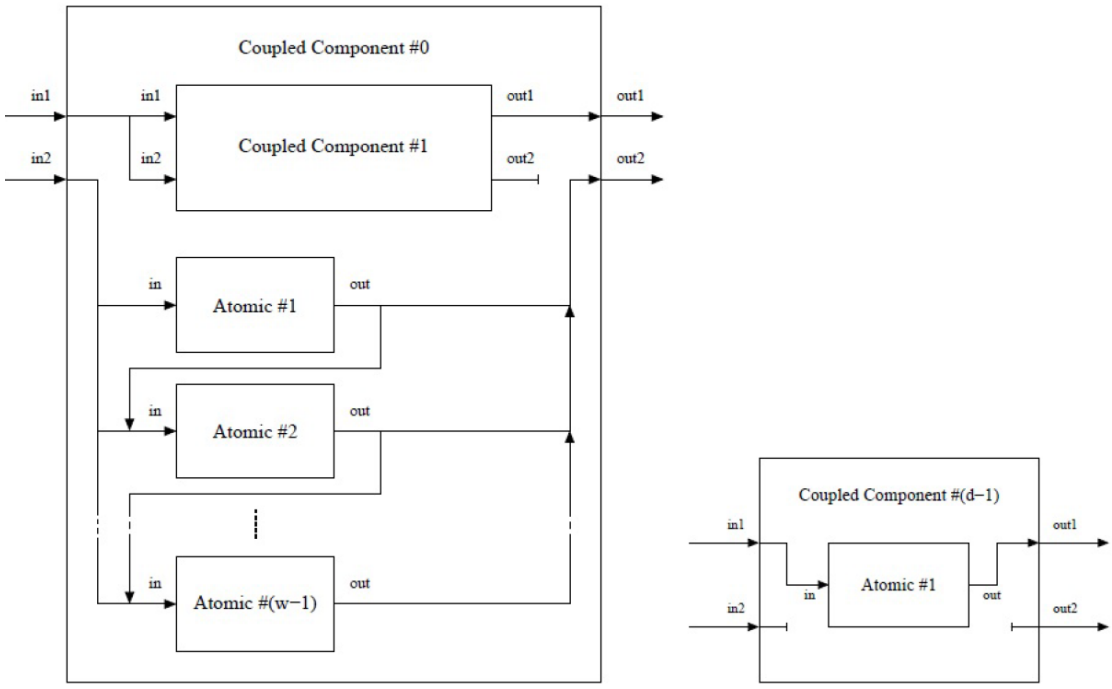

***Figura 4.1**: Modelo DEVStone HO, a la izquierda un ejemplo de disposición regular de modelos acoplados, a la derecha un ejemplo de modelo aplanado [27].*

En la anterior figura se puede apreciar que los modelos HO tienen dos puertos de entrada y dos de salida en cada nivel. Además, el segundo puerto de entrada de cada modelo acoplado está conectado a la entrada de cada modelo atómico. Por otro lado, la salida de cada modelo atómico está conectada a la segunda salida de su modelo acoplado principal. El número de modelos atómicos, funciones de transición y eventos generados en los modelos HO son exactamente los mismos que en el modelo HI con la diferencia de que en HO hay mayor tiempo de ejecución y consumo de memoria, debido a las conexiones de entrada externa adicionales. El cálculo del número de modelos atómicos, funciones de transición y numero de eventos en HO puede realizarse usando las siguientes fórmulas [27]:

$$
\begin{aligned}
\#\text{Atomic} &= (w-1)\cdot(d-1)+1 \\
\#\delta_{\text{int}} &= \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1 \\
&= \left(\frac{w^2-w}{2}\right)\cdot(d-1)+1 \\
\#\delta_{\text{ext}} &= \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1 \\
&= \left(\frac{w^2-w}{2}\right)\cdot(d-1)+1 \\
\#\text{Events} &= \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1 \\
&= \left(\frac{w^2-w}{2}\right)\cdot(d-1)+1
\end{aligned}
$$

***Figura 4.2**: Formulas HO para el cálculo de numero de modelos atómicos, numero de transiciones y numero de eventos [27].*

4.1.1 Estimación teórica de rendimiento de HO en los motores secuencial, paralelo y distribuido

Para realizar una estimación de tiempo teórico que durará la ejecución de un modelo DEVStone HO en un motor de simulación DEVS secuencial, se puede usar la fórmula siguiente:

$$t_s = \#\delta_{int} \cdot t(\delta_{int}) + \#\delta_{ext} \cdot t(\delta_{ext}) + t_m \qquad \textbf{(4.1)}$$

, donde:

- t_s es tiempo de ejecución, en segundos, del modelo HO ejecutado en un motor de simulación DEVS secuencial.
- $\#\delta_{int}, \#\delta_{ext}$ representan el número de transiciones internas y externas respectivamente, definidas en la Figura 4.2.
- $t(\delta_{int}), t(\delta_{ext})$ representan el tiempo empleado en cada transición interna y externa, respectivamente.

- t_m es el tiempo empleado por el motor de simulación en el paso de mensajes.

Para estimar el tiempo de ejecución de un modelo en el motor de simulación secuencial y compararlo con el tiempo medido real, despreciaremos el tiempo de paso de mensajes t_m:

$$t_s \approx \#\delta_{int} \cdot t(\delta_{int}) + \#\delta_{ext} \cdot t(\delta_{ext}) \qquad \textbf{(4.2)}$$

Para realizar una estimación de tiempo teórico que durará la ejecución de un DEVStone HO en un motor de simulación DEVS **paralelo**, se puede usar la fórmula siguiente:

$$t_p = \frac{\#\delta_{int}}{\#H} \cdot t(\delta_{int}) + \frac{\#\delta_{ext}}{\#H} \cdot t(\delta_{ext}) + t_H \ + \ t_m \qquad \textbf{(4.3)}$$

, donde:

- t_p es tiempo de ejecución, en segundos, del modelo HO ejecutado en un motor de simulación DEVS paralelo.
- $\#\delta_{int}, \#\delta_{ext}$ representan el número de transiciones internas y externas respectivamente, definidas en la Figura 4.2.
- $t(\delta_{int}), t(\delta_{ext})$ representan el tiempo empleado en cada transición interna y externa, respectivamente.
- $\#H$ representa el número de hebras utilizado en la simulación paralela.
- t_H es el tiempo empleado por el sistema operativo en la gestión de hebras.
- t_m es el tiempo empleado por el motor de simulación en el paso de mensajes.

Para estimar el tiempo de ejecución de un modelo en el motor de simulación paralelo y compararlo con el tiempo medido real, despreciaremos el tiempo por gestión de hebras t_H y el tiempo de paso de mensajes t_m:

$$t_p \approx \frac{\#\delta_{int}}{\#H} \cdot t(\delta_{int}) + \frac{\#\delta_{ext}}{\#H} \cdot t(\delta_{ext}) \qquad \textbf{(4.4)}$$

En el caso de la estimación teórica de tiempo de ejecución del modelo DEVStone HO en un motor de simulación DEVS distribuido, hay que tener en cuenta que cada modelo atómico se simula como un proceso en sí. Por tanto, tenemos:

$$t_d = \left(\max_{i \in \#Atomic} \#\delta_{int}^{i}\right) \cdot t(\delta_{int}) + \left(\max_{i \in \#Atomic} \#\delta_{ext}^{i}\right) \cdot t(\delta_{ext}) + t_{cloud} + t_m \qquad \textbf{(4.5)}$$

, donde:

- t_d es tiempo de ejecución, en segundos, del modelo HO ejecutado en un motor de simulación DEVS distribuido.
- $\#\delta_{int}^{i}, \#\delta_{ext}^{i}$ representan el número de transiciones internas y externas respectivamente, del modelo atómico i-ésimo.
- $t(\delta_{int}), t(\delta_{ext})$ representan el tiempo empleado en cada transición interna y externa, respectivamente.
- t_{cloud} es el tiempo empleado por el sistema en la nube para la gestión y paso de mensajes entre máquinas virtuales, contenedores, etc.
- t_m es el tiempo empleado por el motor de simulación en el paso de mensajes.

Para estimar el tiempo de ejecución de un modelo en el motor de simulación distribuido y compararlo con el tiempo medido real, despreciaremos el tiempo por gestión cloud t_{cloud} y el tiempo de paso de mensajes t_m

$$t_d \approx \left(\max_{i \in \#Atomic} \#\delta_{int}^{i}\right) \cdot t(\delta_{int}) + \left(\max_{i \in \#Atomic} \#\delta_{ext}^{i}\right) \cdot t(\delta_{ext}) \qquad \textbf{(4.6)}$$

En los siguientes apartados se mostrarán los resultados de ejecutar un modelo DEVStone HO en los motores xDEVS secuencial, paralelo y distribuido, tanto los tiempos reales de ejecución en las máquinas como sus homónimos teóricos.

4.2 Pruebas de rendimiento en la versión secuencial

Para todas las pruebas se ha optado por usar una máquina virtual de referencia usando Google Cloud Platform con una imagen Linux ubuntu-1804-lts, 4 vCPUs y 3.75 GB de RAM. Los detalles de la creación de la máquina virtual y la carga de software

además de los comandos GSC para la ejecución de la simulación se pueden observar en las figuras 4.3 y 4.4:

```
-- Initialization
gcloud config configurations activate tfm2020
gcloud config set project efp-tfm
gcloud auth configure-docker
gcloud config set compute/zone us-central1-a
gcloud config set compute/region us-central1

-- Create virtual machine
gcloud compute instances create vm-sequential --project=efp-tfm \
    --image-family ubuntu-1804-lts --image-project ubuntu-os-cloud \
    --subnet=default --private-network-ip=10.128.0.2 \
    --metadata-from-file startup-script=install.sh

-- Copy main files to instance: jar and run.sh
gcloud compute scp socket.jar vm-sequential:/tmp/java.jar
gcloud compute scp run.sh vm-sequential:/tmp/run.sh

-- Connect to VM with root user and run simulators and coordinator
gcloud compute ssh --project efp-tfm root@vm-sequential \
    --command="bash /tmp/run.sh"

-- Connect to VM and to see the results
gcloud compute ssh --project efp-tfm root@vm-sequential

-- Delete VM's
gcloud compute instances delete vm-sequential
```

***Figura 4.3**: Comandos GSC y SSH para preparar la máquina virtual para su posterior uso en la ejecución del modelo HO en su versión secuencial.*

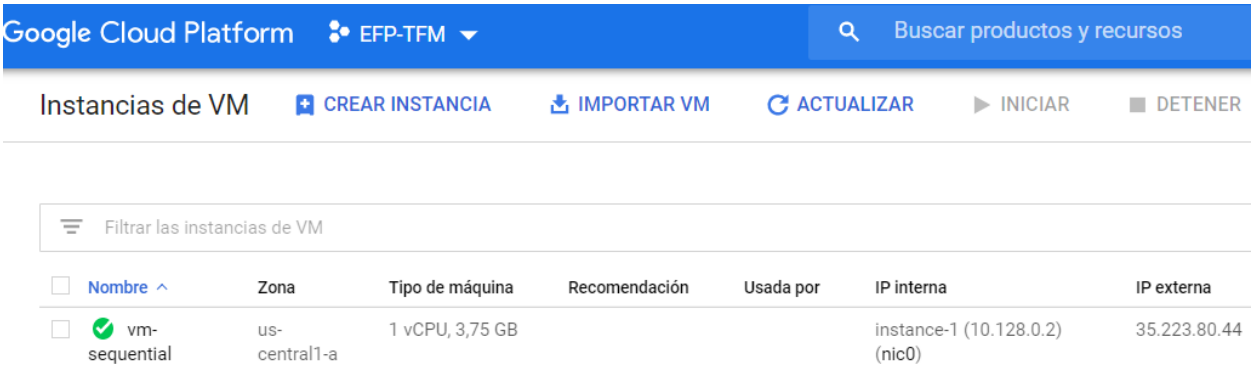

***Figura 4.4**: Máquina virtual vista desde la consola GCP.*

Luego se ha procedido a la ejecución del modelo de simulación DEVStone HO en el motor xDEVS secuencial con los siguientes datos:

width	depth	$t(\delta_{int})$	$t(\delta_{ext})$
6	4	1.618 s	1.618 s

***Tabla 4.1**: Parámetros para la ejecución del modelo de simulación DEVStone HO*

Se debe tomar en cuenta que tanto las pruebas de rendimiento de la versión secuencial y paralela como la distribuida se realizarán sobre un modelo aplanado para ser justos a la hora de comparar resultados. En la siguiente figura 4.5 se muestra el resultado del aplanado de la versión Devstone HO:

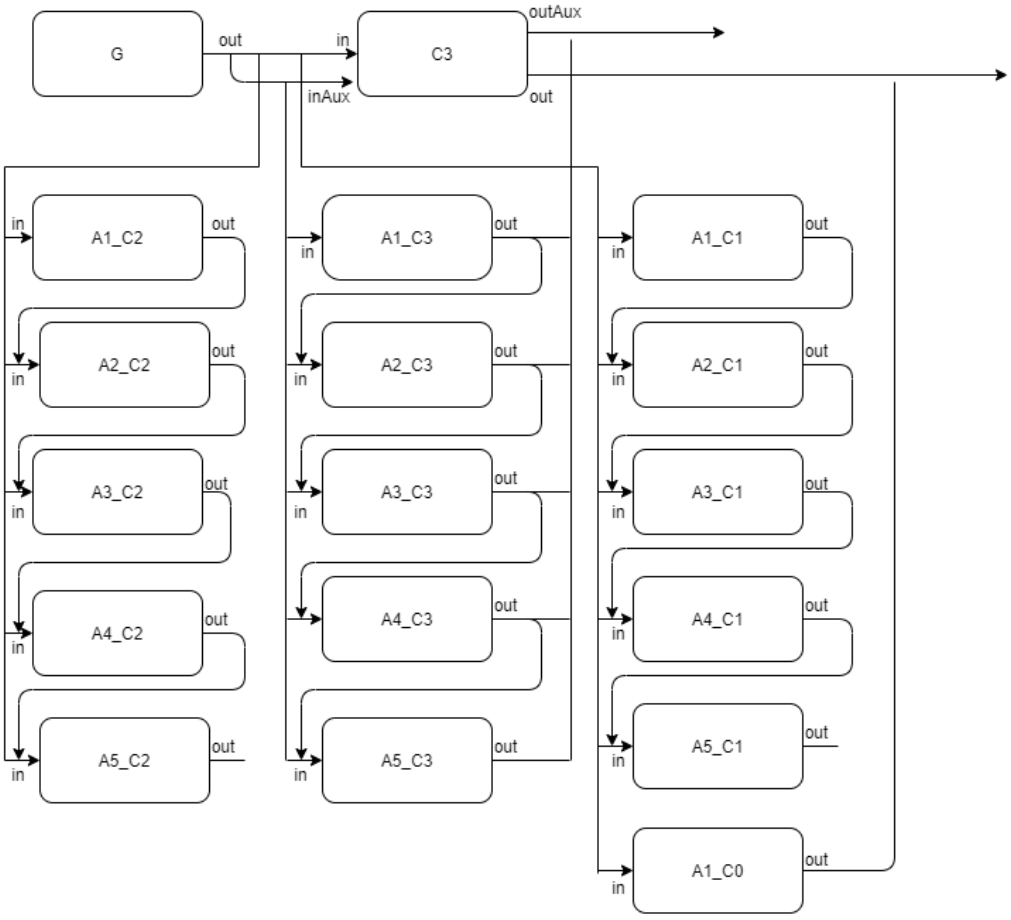

***Figura 4.5**: Modelo Devstone HO aplanado con width=6 y depth=4.*

Lo que arroja el siguiente número de modelos atómicos, funciones de transición y eventos, de acuerdo con la Fig. 4.2:

$$\#Atomic = 16$$

$$\#\delta_{int} = 46$$

$$\#\delta_{ext} = 46$$

$$\#Events = 46$$

Tras realizar la simulación, se obtiene el resultado reflejado en la siguiente tabla y su respectiva imagen de ejecución en la nube (Fig. 4.6):

Procesadores	**Tiempo real**	**Tiempo teórico t_s (4.2)**
4	148.88 s	148.86 s

***Tabla 4.2**: Resultados de la versión secuencial en la nube.*

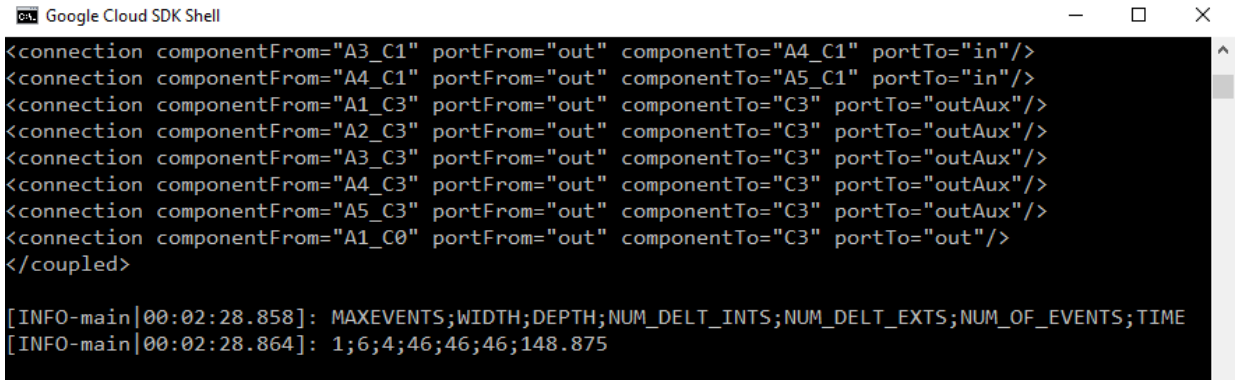

***Figura 4.6:** Ejecución del modelo Devstone HO secuencial en una máquina virtual con cuatro procesadores.*

Como se puede observar, el tiempo de ejecución real es muy similar al teórico calculado en (4.2). Por otro lado, dado que estamos lanzando las simulaciones en una máquina de referencia y en un entorno muy controlado (no hay variaciones por E/S ni por procesos adicionales cargados en la máquina virtual), las variaciones de una ejecución a otra son prácticamente nulas.

4.3 Pruebas de rendimiento en la versión paralela

De la misma manera que en la versión secuencial, en la versión paralela se ha utilizado la misma máquina virtual de referencia: Linux ubuntu-1804-lts, 1-4 vCPUs y 3.75

GB de RAM. Para la ejecución se tomaron los mismos datos de configuración del modelo DEVStone HO:

width	**depth**	$t(\delta_{int})$	$t(\delta_{ext})$
6	4	1.618 s	1.618 s

Tabla 4.3: Parámetros para la ejecución del modelo de simulación DEVStone HO.

La ejecución del modelo arrojó los siguientes resultados:

Figura	**Hebras (#H)**	**Tiempo real**	**Tiempo teórico t_p (4.4)**
4.7	1	148.88 s	148.86 s
4.8	2	77.70 s	74.43 s
4.9	4	42.09 s	37.21 s
4.10	8	25.91 s	18.61 s

***Tabla 4.4**: Resultados de la versión paralela en la nube.*

Como se puede observar en la tabla anterior, el rendimiento del motor de simulación DEVS paralelo mejora con el número de hebras, con una ganancia prácticamente lineal igual a x2. Cada vez hay mayor diferencia con el tiempo teórico por la gestión de hebras, que en 4.4 se desprecia por sencillez de cálculo. A pesar de la notable ganancia en la ejecución paralela, veremos que puede ser aún mayor en la ejecución distribuida.

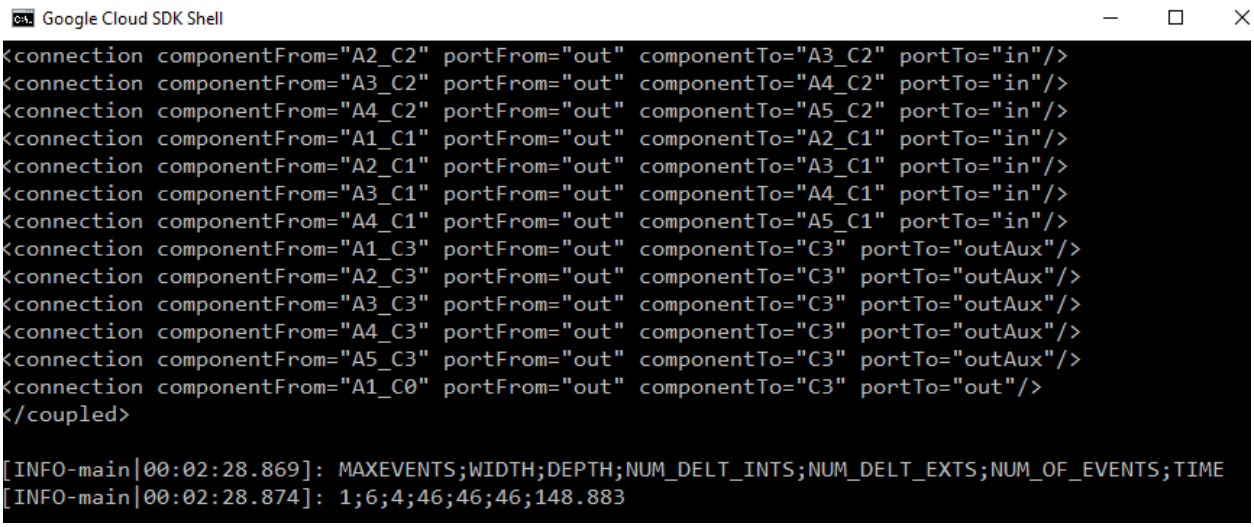

***Figura 4.7**: Ejecución del modelo Devstone HO en paralelo en la máquina virtual de referencia usando una hebra.*

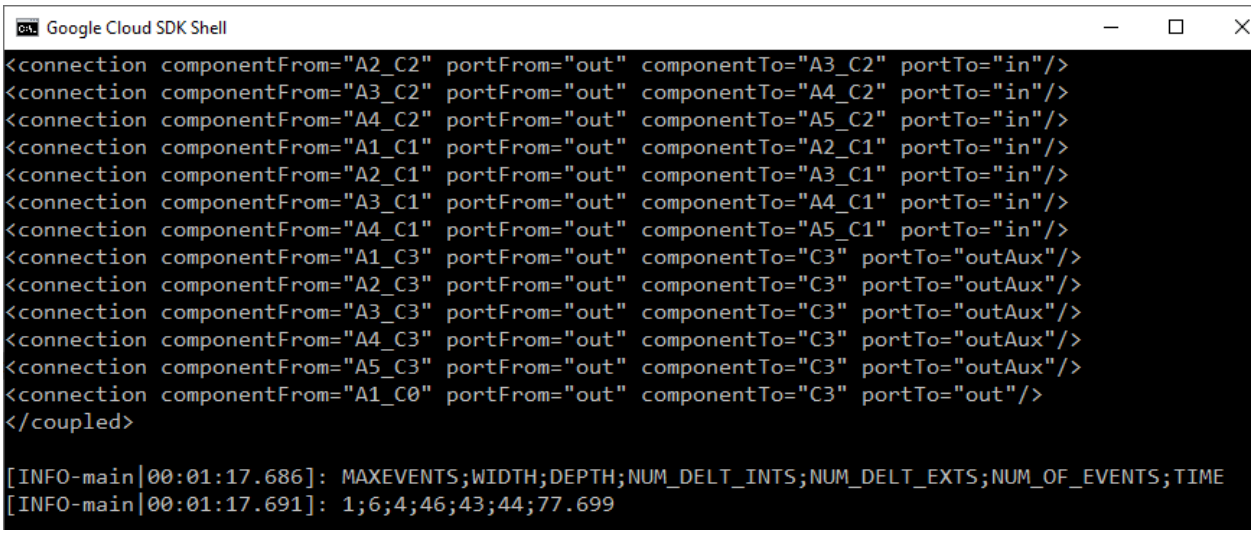

***Figura 4.8**: Ejecución del modelo Devstone HO en paralelo en la máquina virtual de referencia usando dos hebras.*

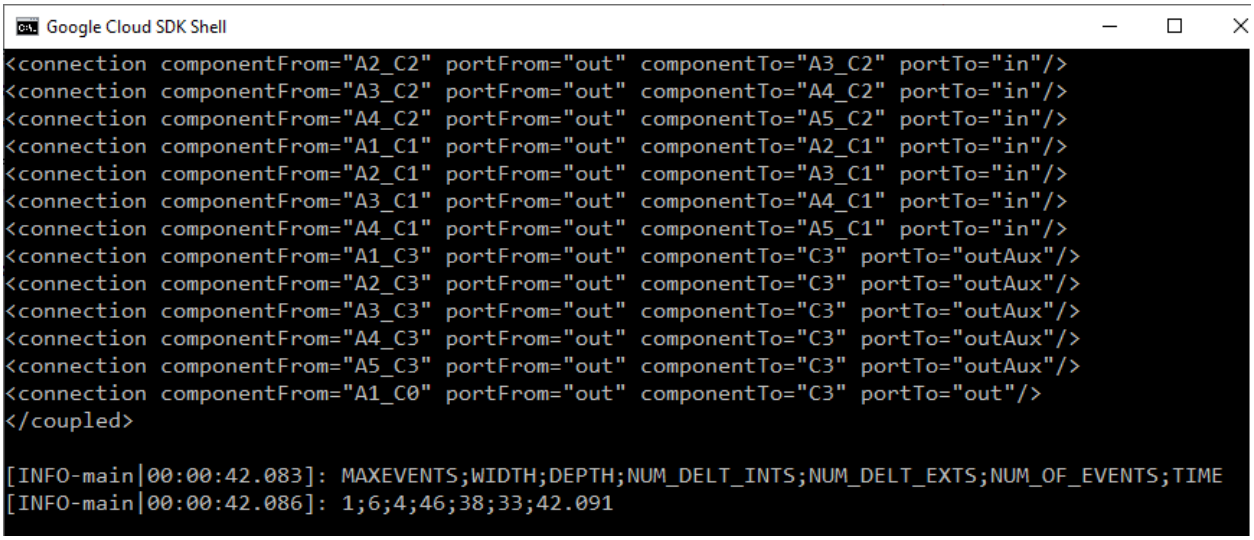

***Figura 4.9**: Ejecución del modelo Devstone HO en paralelo en la máquina virtual de referencia usando cuatro hebras.*

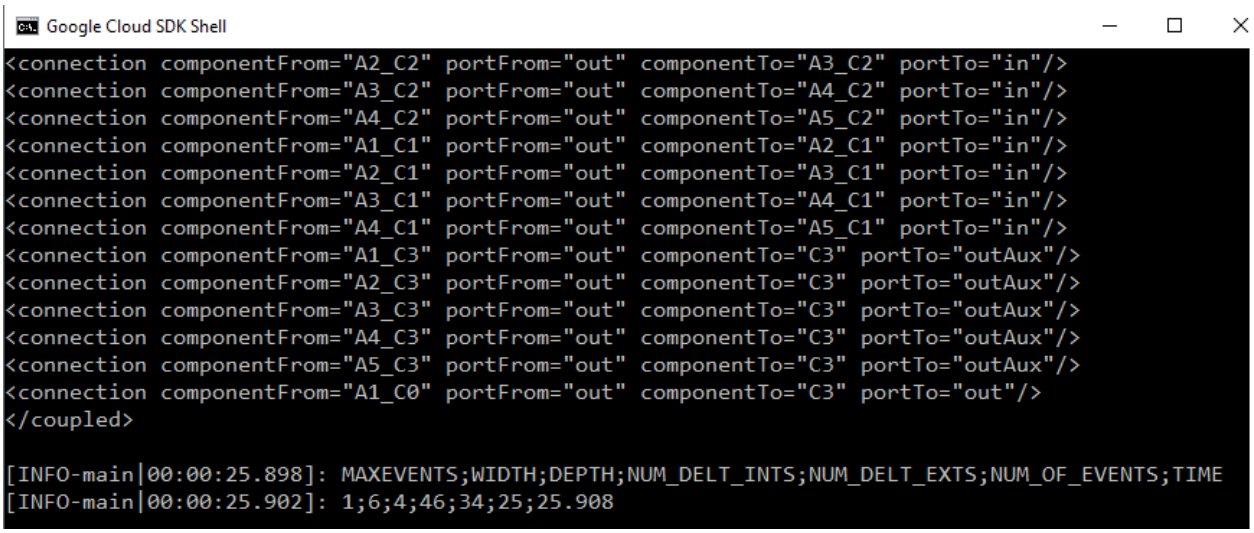

***Figura 4.10**: Ejecución del modelo Devstone HO en paralelo en la máquina virtual de referencia usando ocho hebras.*

4.4 Pruebas de rendimiento en la versión distribuida

En el caso de las pruebas de rendimiento de la versión distribuida, se procedió primeramente a implementar una clase DevStoneCoupledHODistributed que hereda de la clase Coupled e implementa la interfaz DistributedInterface (Ver figura 4.11). Tal como se indica en el capítulo 3, la ejecución distribuida implica preparar los modelos de simulación para ejecutarse de manera distribuida. **Esto no implica para nada cambiar**

la dinámica del modelo original, sino crear un wrapper intermedio. Además, se utiliza una clase Nodo desde la cual se ejecutarán los simuladores y el coordinador en cada máquina virtual o contenedor en la nube (Ver figura 4.12). Tampoco se debe olvidar que el modelo debe estar aplanado (Ver figura 4.13). Por lo demás se usan las clases DEVStone implementadas originalmente, sin cambiar ni una coma.

```
public class DevStoneCoupledHODistributed extends Coupled implements DistributedInterface{
    // Atributtes
    private String simulationPlane;
    // HO parameters:
    private double intDelayTime;
    private double extDelayTime;
    // Generator parameters:
    double preparationTime;
    double period;
    int maxEvents;

    // Constructors
    public DevStoneCoupledHODistributed(String name, String simulationPlane,
            double intDelayTime, double extDelayTime,
            double preparationTime, double period, int maxEvents) {
        this(simulationPlane, intDelayTime, extDelayTime,
                preparationTime, period, maxEvents);
        super.name= name;
    }

    public DevStoneCoupledHODistributed(String simulationPlane,
            double intDelayTime, double extDelayTime,
            double preparationTime, double period, int maxEvents) {
        super(DevStoneCoupledHODistributed.class.getSimpleName());
        this.simulationPlane = simulationPlane;
        this.intDelayTime = intDelayTime;
        this.extDelayTime = extDelayTime;
        this.preparationTime = preparationTime;
        this.period = period;
        this.maxEvents = maxEvents;
    }

    // Methods
    public Atomic returnModel(String ClassName) {
        if (ClassName.compareTo("DevStoneGenerator") == 0) {
            return new DevStoneGenerator(ClassName, this.preparationTime, this.period, this.maxEvents);
        } else{
            return new DevStoneAtomic(ClassName, this.preparationTime, this.intDelayTime, this.extDelayTime);
        }
    }

    public String getSimulationPlane() {
        return simulationPlane;
    }
}
```

***Figura 4.11**: Implementación de DEVStone HO para correrse de manera distribuida.*

```java
public class Node {
    // LOGGER
    private static final Logger LOGGER = Logger.getLogger(DevStoneCoupledHODistributed.class.getName());
    public static void main(String[] args) {
        DevsLogger.setup(Level.INFO);
        //name, simulationPlane, intDelayTime, extDelayTime, preparationTime, period, maxEvents
        DevStoneCoupledHODistributed dho = new DevStoneCoupledHODistributed(
                args[1],//name
                args[7],//simulationPlane
                Double.parseDouble(args[2]), //intDelayTime
                Double.parseDouble(args[3]), //extDelayTime
                Double.parseDouble(args[4]), //preparationTime
                Double.parseDouble(args[5]), //period
                Integer.parseInt(args[6]) //maxEvents
        );
        SimulatorDistributed p;
        CoordinatorDistributed c;
        if( args[0].compareTo("Atomic")==0 ){
            p = new SimulatorDistributed(dho, args[8],Integer.parseInt(args[9]),Integer.parseInt(args[10]));
        } else if( args[0].compareTo("Simulator")==0){
            DevsLogger.setup(Level.INFO);
            LOGGER.info("Run Coordinator ....");
            c = new CoordinatorDistributed(dho);
            long start = System.currentTimeMillis();
            c.initialize();
            c.simulate(Long.MAX_VALUE);
            c.exit();
            long end = System.currentTimeMillis();
            double time = (end - start) / 1000.0;
            LOGGER.info("TIME: " + time);
        }
    }
}
```

***Figura 4.12**: Clase Node que ejecutara un simulador o coordinador de manera independiente en cada máquina virtual o contenedor.*

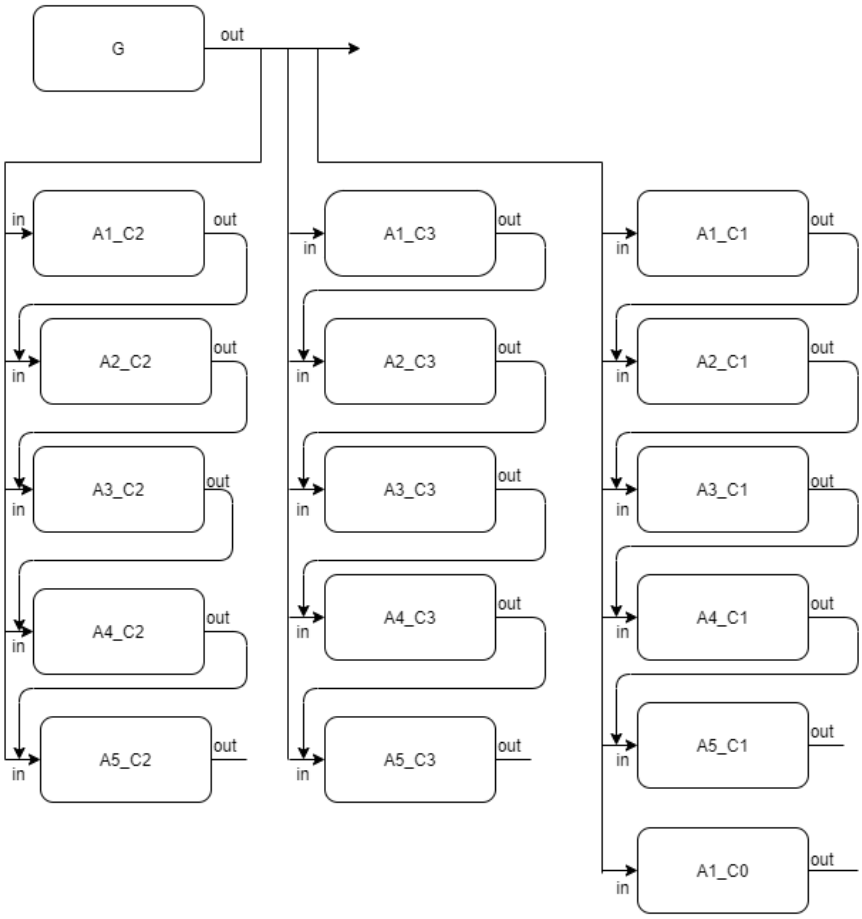

***Figura 4.13**: Modelo DEVStone HO aplanado que servirá de base para colocar cada modelo atómico en una máquina virtual o contenedor de manera distribuida.*

Las pruebas de rendimiento se han ejecutado usando de nuevo la máquina virtual de referencia, con Linux ubuntu-1804-lts, 4 vCPUs y 3.75 GB de RAM. También se mantienen valores de configuración para el modelo DEVStone HO distribuido:

width	**depth**	$t(\delta_{int})$	$t(\delta_{ext})$
6	4	1.618 s	1.618 s

***Tabla 4.5**: Parámetros para la ejecución del modelo de simulación DEVStone HO.*

El **primer escenario** de pruebas se realizó usando hasta dos máquinas virtuales de referencia y ejecutando de manera balanceada el DEVStone HO anterior. La ejecución bajo este escenario arrojó los siguientes resultados:

Figura	Máquinas virtuales	Tiempo real	Tiempo teórico t_d (4.6)
4.14	1	17.50 s	16.18 s
4.15	2	17.02 s	16.18 s

***Tabla 4.6**: Resultados de la versión distribuida en la nube (Escenario 1: máquinas virtuales).*

Lo primero que llama la atención es la ganancia con respecto al motor secuencial (x8.51), pero también con respecto al motor paralelo (x2.41 en el caso de 4 hebras). Esto es así porque en el motor distribuido, cada modelo atómico se comporta como un proceso independiente. Siendo, por la Fig. 4.2, $\left(\max_{i\in \#Atomic} \#\delta_{int}^{i}\right) = \left(\max_{i\in \#Atomic} \#\delta_{ext}^{i}\right) = 5$. Se desprende que el tiempo teórico de ejecución será igual a $t_d = 16.18\,s$, independientemente del número de máquinas virtuales, contenedores, etc. Claro, esto no es realista, pues la ejecución distribuida consume sus recursos en cuanto a paso de mensajes y gestión de contenedores y procesos. No obstante, podemos observar claramente las bondades de la ejecución distribuida.

El segundo aspecto que llama la atención es que usando 2 máquinas virtuales se consume menos tiempo (aunque sea poco apreciable) que usando 1 máquina virtual. Creemos que es por el tiempo de gestión de hebras. Cuando se usa una única máquina virtual esta gestiona 16 hebras (tantas como modelos atómicos), mientras que, en el caso de 2 máquinas virtuales, cada una gestiona 8 hebras. Habría que realizar más pruebas, con modelos de distinto tamaño, para poder contrastar mejor estas hipótesis, pero este aspecto se deja como trabajo futuro, debido al coste económico que supone realizar este tipo de pruebas en Google Cloud.

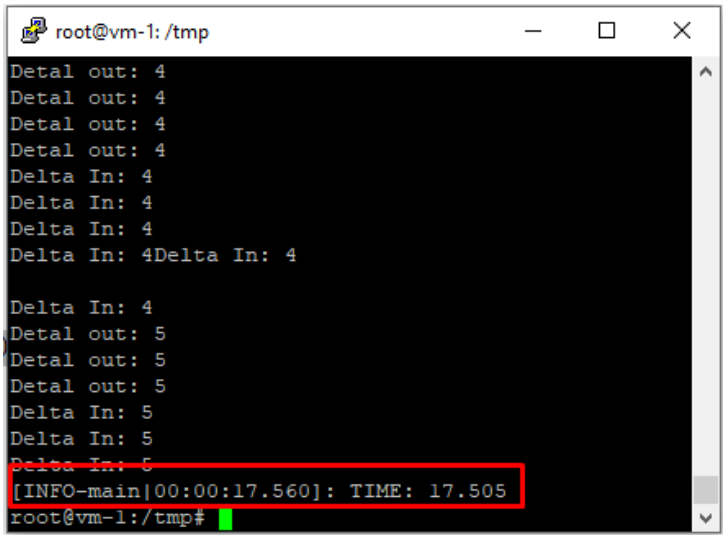

***Figura 4.14**: Ejecución del modelo Devstone HO distribuido en una máquina virtual.*

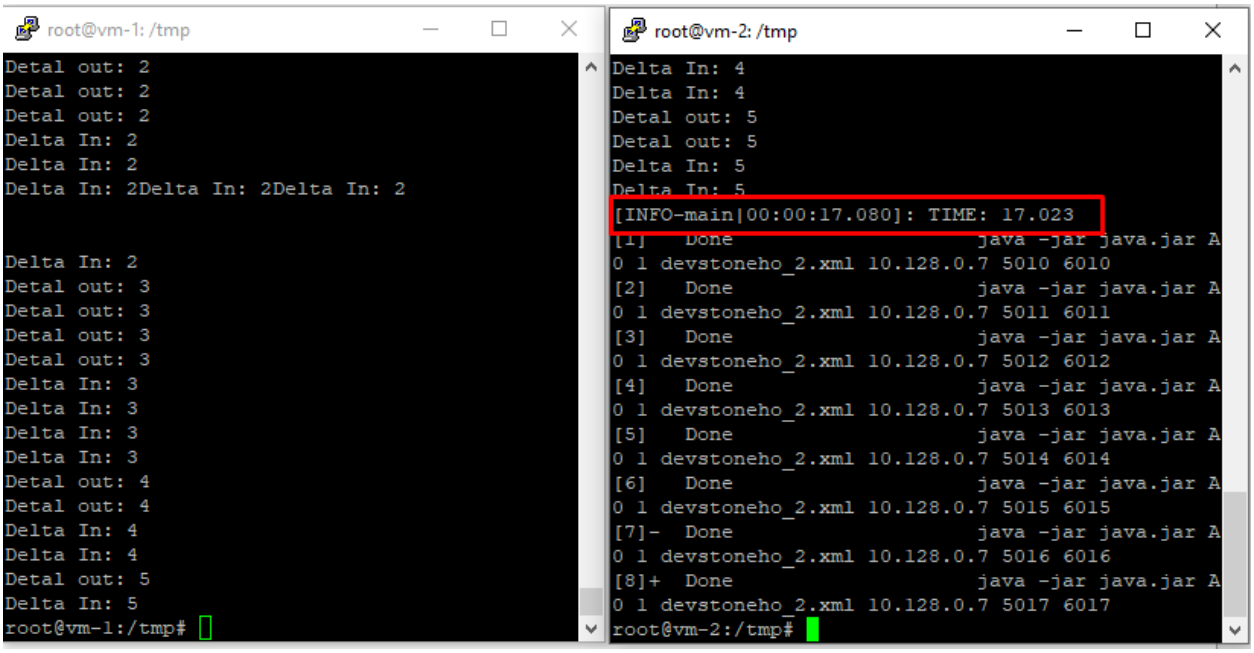

***Figura 4.15**: Ejecución del modelo Devstone HO distribuido en dos máquinas virtuales.*

El **segundo escenario** de pruebas se realizó usando hasta dos máquinas virtuales y colocando contenedores Docker en cada una de ellas, además de balancear los 18 elementos del modelo en cada contenedor. La ejecución del modelo arrojó los siguientes resultados:

Figura	Máquina virtual	Dockers (por MV)	Tiempo real	Tiempo teórico t_d (4.6)
4.16	1	18	17.92 s	16.18 s
4.17	1	1	17.71 s	16.18 s
4.18	2	2	17.11 s	16.18 s

***Tabla 4.7**: Resultados de la versión distribuida en la nube (Escenario 2: contenedores Docker).*

En este caso usando dieciocho Dockers en una máquina virtual (Fig. 4.16) tarda un poco más que si usamos un Docker en una máquina virtual (Fig. 4.17), está claro que el t_{cloud} es más notable a medida que se va incrementando el uso de contenedores Docker. Por otro lado, si la carga de los elementos de la simulación (simuladores/coordinador) se reparte de manera balanceada en dos máquinas virtuales (Fig. 4.18), el tiempo mejora, pero este no es mejor que si usáramos solamente máquinas virtuales (Fig. 4.15), y nuevamente esto se debe a que existe un t_{cloud} mayor cuando se usa contenedores Docker.

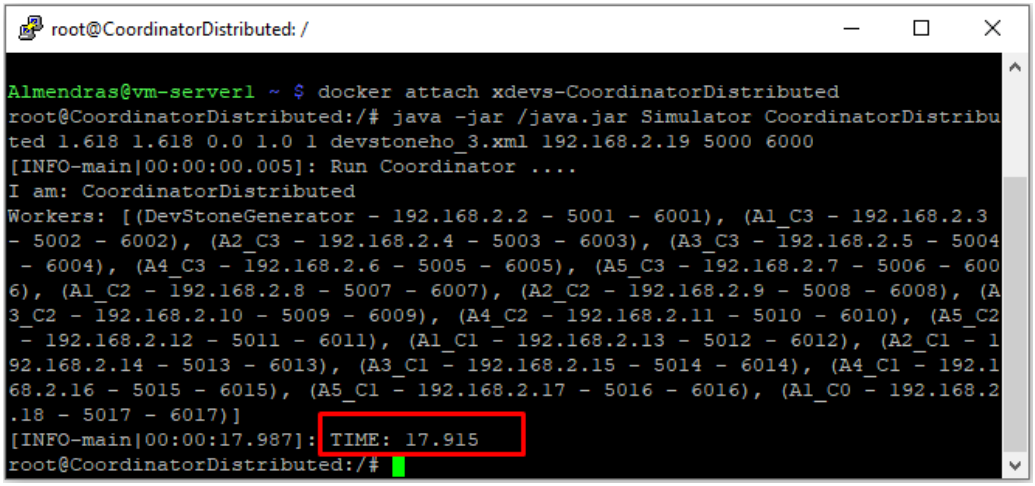

***Figura 4.16:** Ejecución del modelo Devstone HO distribuido en una máquina virtual con dieciocho contenedores.*

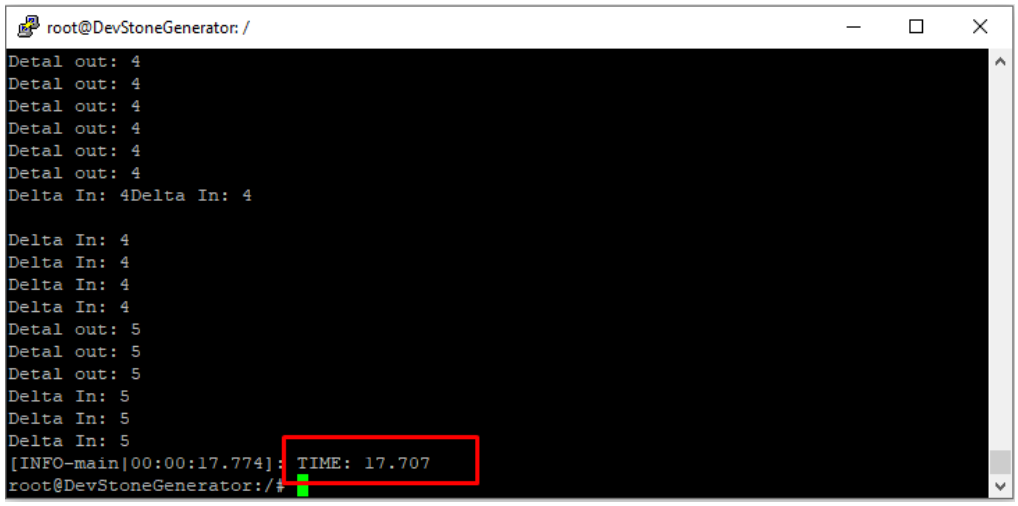

***Figura 4.17**: Ejecución del modelo Devstone HO distribuido en una máquina virtual con un contenedor.*

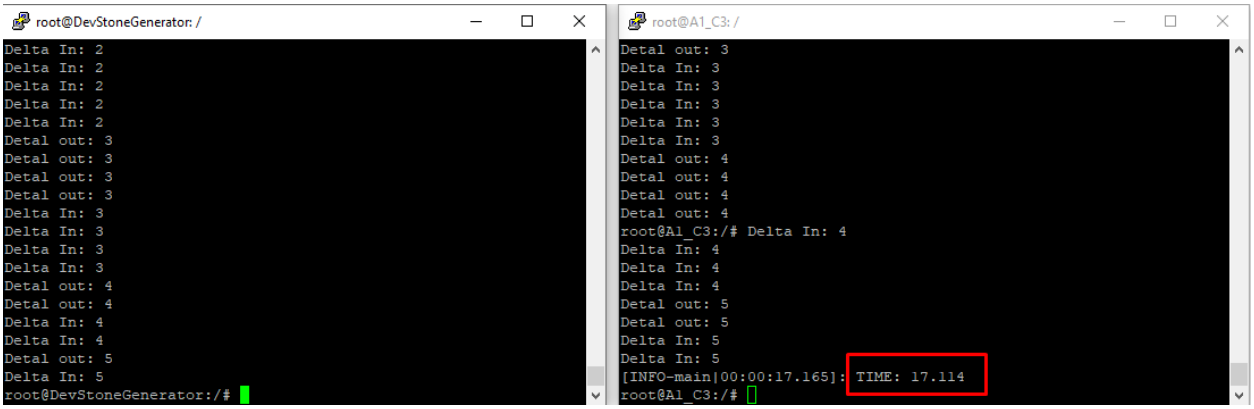

***Figura 4.18**: Ejecución del modelo Devstone HO distribuido en dos máquinas virtuales cada una con dos contenedores.*

El **tercer escenario** de pruebas se realizó usando un clúster con dos nodos o máquinas virtuales con las mismas características que los escenarios anteriores. En el clúster se crearon hasta dos contenedores Docker. Recordar que el uso de recursos es administrado por Kubernetes. La ejecución del modelo anterior arrojó los siguientes resultados:

Figura	**Nodos (MV)**	**Dockers (por clúster)**	**Tiempo real**	**Tiempo teórico t_d (4.6)**
4.19	2	1	17.66 s	16.18 s

4.20	2	2	17.64 s	16.18 s

***Tabla 4.8**: Resultados de la versión distribuida en la nube (Escenario 3: clúster con Kubernetes).*

Si bien las pruebas en este caso (fig. 4.19 y fig. 4.20) arrojan un tiempo mayor (aunque son centésimas y no señala contundencia con respecto a los resultados de la fig. 4.18), se esperaría un mejor tiempo dado que la gestión de recursos es realizada de forma automática por Kubernetes esperándose un t_{cloud} menor, sin embargo, esto hace pensar que existen otros escenarios talvez más complejos en donde aplicar clúster a una simulación distribuida puede lograrse tener un mejor tiempo de ejecución, estas pruebas podrían ser parte de un trabajo futuro.

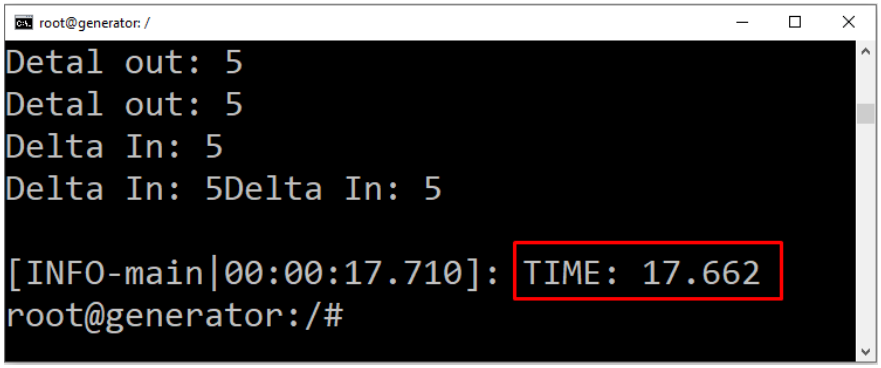

***Figura 4.19**: Ejecución del modelo Devstone HO distribuido en un cluster de dos nodos con un contenedor.*

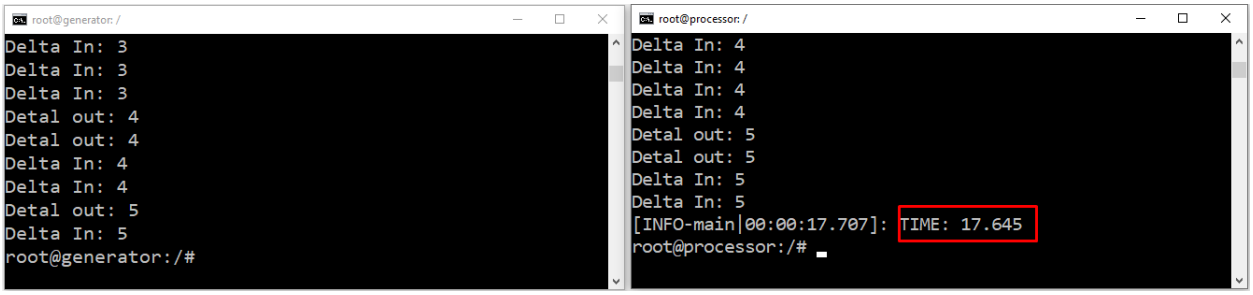

***Figura 4.20**: Ejecución del modelo Devstone HO distribuido en un clúster de dos nodos con dos contenedores.*

4.5 Conclusiones

Después de haber realizado las pruebas de rendimiento en diferentes escenarios se tienen las siguientes conclusiones:

- En este capítulo se han podido recopilar las fórmulas de cálculo de tiempo de ejecución teórico de cada uno de los modelos de simulación incluyendo el modelo de simulación distribuido.
- El sistema de simulación distribuido aprovecha mejor la arquitectura aplanada de un modelo DEVS, especialmente cuando la cantidad de modelos atómicos distribuidos es elevada (en el caso de las pruebas de este capítulo: dieciséis). Es por ello que se ve un mejor tiempo ejecutando el modelo de simulación distribuida que los otros.
- Se aprecia que la ejecución del modelo de simulación distribuido entre máquinas virtuales, o contenedores Docker o clúster con Kubernetes, el tiempo de ejecución prácticamente no se aleja en cualquiera de los tres casos, por lo que el tiempo de paso de mensajes y uso de contenedores Dockers o máquinas virtuales es muy pequeño y prácticamente no influye en decantarse por el uso de uno u otro. Esto demuestra la viabilidad de la solución propuesta en las tres arquitecturas de referencia.
- No hubo la posibilidad de hacer pruebas con más de dos máquinas virtuales por la limitación del número de procesadores que tiene la cuenta de GCP de pruebas que se usó. Se deja como trabajo futuro profundizar en este aspecto, ya que parece que la escalabilidad del motor distribuido implementado en este trabajo está prácticamente asegurada.

Capítulo 5 - Conclusiones y trabajo futuro

Este capítulo resume las conclusiones más importantes del trabajo, así como las direcciones futuras que se podrían tomar para mejorar lo abordado en este trabajo.

5.1 Conclusiones

Las conclusiones más relevantes del trabajo son las siguientes:

Se ha logrado desarrollar una extensión de la biblioteca de simulación DEVS conocida como xDEVS (https://github.com/dacya/xdevs). Esta extensión admite la simulación de modelos de forma distribuida, que añadida a los motores de simulación ya existentes tanto secuencial como paralela, permite desplegar simulaciones usando servicios de la nube. Para el caso del presente trabajo se demostró su uso utilizando los servicios de Google Cloud Platform.

Se ha aprendido a gestionar la infraestructura IaaS de GCP. Ese aprendizaje ha permitido desplegar modelos de simulación distribuida (EF-P y DEVStone) en tres arquitecturas: Máquinas virtuales, Dockers y Cluster con Kubernetes.

Se ha verificado el correcto funcionamiento de la extensión implementada usando del modelo DEVS de referencia por excelencia: el generador – procesador – transductor.

Además, se ha analizado el rendimiento y la escalabilidad de la solución propuesta. Para ello se ha utilizado el benchmark DEVStone y en particular su modelo HO para realizar pruebas de rendimiento en la nube y poder comparar los tiempos de ejecución de cada una de las versiones: secuencial, paralela y distribuida.

Para realizar todas las simulaciones en un entorno controlado, se han utilizado los servicios de infraestructura de la nube (IaaS) de GCP, que ayudaron a plasmar el despliegue de cada uno de los elementos del modelo de simulación distribuida (coordinador/simuladores) en máquinas virtuales, contenedores Docker y clúster con Kubernetes. Estos servicios son estándares de virtualización en la industria de la nube (Docker/Kubernetes/OpenStack). Al haber sido adoptados por los principales

proveedores de venta de servicios en la nube, el despliegue realizado en el presente trabajo podría ser aplicado de igual manera con otros proveedores distintos de GCP.

Con todo ello, se han cumplido los objetivos planteados en el presente trabajo, diseñando una plataforma de simulación de eventos distribuida versátil, sencilla de desplegar, y escalable.

5.2 Trabajo futuro

A continuación, se plantean algunas acciones que se abordarán en el futuro:

Es posible mejorar y formalizar aún más DEVStone para el sistema distribuido, por ejemplo, considerando distintos tiempos de ejecución en las funciones de transición y reemplazando el benchmark de uso de CPU (Dhrystone), que se ha quedado un tanto obsoleto.

Es necesario continuar con las pruebas de rendimiento en distintos escenarios de complejidad y mayor cantidad de elementos de simulación desplegados de manera distribuida, para de esa manera obtener pruebas mucho más contundentes acerca de cuál sería el mejor escenario de las tres presentadas para una mejor simulación distribuida.

Se puede mejorar la gestión de logs de xDEVS con la finalidad de realizar una mejor depuración y lograr observar con mayor detalle la ejecución de la simulación distribuida.

Se debe automatizar el despliegue de los modelos de simulación distribuida en la nube mediante el uso de herramientas DevOps como por ejemplo Terraform. Aunque el despliegue diseñado en este trabajo es en cierta forma intuitivo, se volvería inmanejable en modelos heterogéneos (DEVStone es bastante homogéneo y repetitivo en todas sus variantes).

Es recomendable pensar en evolucionar el software de simulación xDEVS especialmente para su uso distribuido tomando como base software creado bajo el manifiesto reactivo[29] (Responsivos, Resilientes, Elásticos y Orientados a mensajes), algunas herramientas que podrían usarse son Akka (https://akka.io) y Apache Kafka

(https://kafka.apache.org), lográndose mejorar el paso de mensajes y la gestión de modelos atómicos, especialmente si se desea escalar el modelo para manejar millones de modelos atómicos dentro del modelo de simulación distribuida.

Chapter - Introduction

This chapter aims to set out the objectives of this Work of Master Thesis, the work plan to be followed in order to achieve the stated objectives, as well as a summary of each of the chapters developed in this work.

Objectives

One of the main objectives is to build an extension to the xDEVS [1] simulation engine that allows models to be defined on distributed platforms with the capacity to run on cloud infrastructures.

Likewise, with the previous extension and using cloud services, to test different simulation architectures that contemplate communication between virtual machines, between containers and clusters.

On the other hand, use software that allows performance tests to be carried out to compare simulations made with xDEVS in its sequential, parallel format and the new distributed version, in order to obtain quantitative conclusions.

Finally, to translate the process and the results of the previous objectives into a master's degree finalization working paper.

Work plan

Employ 4 months to research the problem and learn the tools that will allow the coding and testing of the locally distributed xDEVS version, as well as use the GPT (generator-processor-transducer) simulation model as the base case for testing thereafter.

Likewise, in 2 months, testing of the distributed xDEVS version in a cloud service provider's infrastructure will be carried out using the communication capacities between virtual machines, containers and clusters that will allow the deployment of the distributed simulation system in these possible architectures.

Subsequently, in 2 months the performance tests of the sequential, parallel and new distributed version of xDEVS will be carried out to obtain quantitative conclusions and recommendations of the best scenario for each version.

Finally, in 2 months to carry out the final master working document with its respective review and approval by the directors.

Memory structure

The report consists of five chapters explained in summary form below.

Chapter 1: Introduction. The same that exposes the main objectives and steps to follow in the course of the project.

Chapter 2: Motivation and state of the art. This chapter covers and explains the main aspects that motivate the execution of this work, as well as the foundations and state of the art of high performance computing, together with the new paradigm offered by technological services for infrastructure, platforms and software in the cloud. The chapter ends with a review of the state of play of simulation, especially distributed and discrete-event based, and how classic sequential or parallel simulation can be moved to the cloud to take advantage of the underlying characteristics of the cloud.

Chapter 3: System architecture. Changes and enhancements to the xDEVS simulation tool to support distributed simulation capable of running in the cloud are detailed here. In addition, three distributed simulation architectures are discussed, all of which are compatible with the extension implemented in this paper.

Chapter 4: Experimental Evaluation. This chapter verifies the distributed platform, using a model considered standard in the literature, the Generator-Processor-Transducer (or GPT, also known as Generator, Processor, Transducer). It also details the benchmarking of the three simulation solutions offered by xDEVS: sequential, parallel and the new distributed option. To quantify this comparison, the benchmarks included in DEVStone are used, and the result will be the obtaining of aspects and scenarios in which each solution behaves in a better way.

Chapter 5. Conclusions and future work. In this chapter the conclusions of the work are finally presented as well as the recommendations or future suggestions derived from the experiments carried out and the experience obtained.

Chapter - Conclusions and future work

This chapter summarizes the most important conclusions of the work, as well as future directions that could be taken to improve what is addressed in this work.

Conclusions

The most relevant conclusions of the work are the following:

An extension of the DEVS simulation library known as xDEVS (https://github.com/iscar-ucm/xdevs) has been developed. This extension supports the simulation of models in a distributed way, which added to the existing simulation engines both sequential and parallel, allows the deployment of simulations using cloud services. In the case of the present work, its use was demonstrated using Google Cloud Platform services.

We learned how to manage GCP's IaaS infrastructure. This learning has allowed the deployment of distributed simulation models (EF-P and DEVStone) in three architectures: Virtual machines, Dockers and Cluster with Kubernetes.

The correct functioning of the implemented extension has been verified using the reference DEVS model par excellence: the generator - processor - transducer.

In addition, the performance and scalability of the proposed solution has been analyzed. For this purpose, the DEVStone benchmark has been used and in particular its HO model to carry out performance tests in the cloud and to be able to compare the execution times of each of the versions: sequential, parallel and distributed.

To carry out all the simulations in a controlled environment, GCP's Cloud Infrastructure Services (IaaS) have been used to deploy each of the elements of the distributed simulation model (coordinator/simulators) in virtual machines, Docker containers and clusters with Kubernetes. These services are virtualization standards in the cloud industry (Docker/Kubernetes/OpenStack). Since they have been adopted by the main providers of cloud services, the deployment carried out in this work could be applied in the same way with providers other than GCP.

With all this, the objectives set out in this work have been met, designing a versatile, easy-to-deploy and scalable distributed event simulation platform.

Future work

The following are some actions that will be addressed in the future:

DEVStone can be further improved and formalized for the distributed system, for example by considering different runtimes for the transition functions and by replacing the CPU usage benchmark (Dhrystone), which has become somewhat obsolete.

It is necessary to continue with the performance tests in different complexity scenarios and more simulation elements deployed in a distributed way, in order to obtain much more convincing evidence about which would be the best scenario of the three presented for a better distributed simulation.

The xDEVS log management can be improved in order to perform a better debugging and to be able to observe in more detail the execution of the distributed simulation.

The deployment of distributed simulation models in the cloud should be automated using DevOps tools such as Terraform. Although the deployment designed in this work is somewhat intuitive, it would become unmanageable in heterogeneous models (DEVStone is quite homogeneous and repetitive in all its variants).

It is recommended to think about evolving the xDEVS simulation software especially for distributed use taking as a base software created under the reactive manifesto [29] (Responsive, Resilient, Elastic and Message-oriented), some tools that could be used are Akka (https://akka.io) and Apache Kafka (https://kafka.apache.org), improving the message flow and the management of atomic models, especially if you want to scale the model to handle millions of atomic models within the distributed simulation model.

BIBLIOGRAFÍA

[1] J. L. Risco, «xDEVS Project», abr. 01, 2020. https://github.com/dacya/xdevs (accedido jun. 26, 2020).

[2] H. Hibino, Y. Fukuda, Y. Yura, K. Mitsuyuki, y K. Kaneda, «Manufacturing adapter of distributed simulation systems using HLA», en *Proceedings of the Winter Simulation Conference*, dic. 2002, vol. 2, pp. 1099-1107 vol.2, doi: 10.1109/WSC.2002.1166363.

[3] J. Venkateswaran y Y. J. Son, «Design and development of a prototype distributed simulation for evaluation of supply chains», *Int. J. Ind. Eng. Theory Appl. Pract.*, vol. 11, n.º 2, pp. 151-160, jun. 2004.

[4] S. J. E. Taylor, «Distributed simulation: state-of-the-art and potential for operational research», *Eur. J. Oper. Res.*, vol. 273, n.º 1, pp. 1-19, feb. 2019, doi: 10.1016/j.ejor.2018.04.032.

[5] R. M. Fujimoto, «Research Challenges in Parallel and Distributed Simulation», *ACM Trans. Model. Comput. Simul.*, vol. 26, n.º 4, pp. 1-29, may 2016, doi: 10.1145/2866577.

[6] A. Gokhberg, L. Ermert, J. Igel, y A. Fichtner, «Development of a combined cloud-HPC computing infrastructure for mapping seismic noise sources at the continental scale», *Geophys. Res. Abstr.*, vol. 21, pp. 1-1, ene. 2019.

[7] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, y R. Buyya, «HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges», *ACM Comput. Surv.*, vol. 51, n.º 1, pp. 1-29, ene. 2018, doi: 10.1145/3150224.

[8] Microsoft Azure, «Conceptos de la nube: Principios de la informática en la nube - Learn», *Conceptos de la nube: Principios de la informática en la nube*. https://docs.microsoft.com/es-es/learn/modules/principles-cloud-computing/ (accedido feb. 22, 2020).

[9] N. Kratzke y R. Siegfried, «Towards cloud-native simulations – lessons learned from the front-line of cloud computing», *J. Def. Model. Simul. Appl. Methodol. Technol.*, p. 154851291989532, ene. 2020, doi: 10.1177/1548512919895327.

[10] R. Buyya *et al.*, «A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade», *ACM Comput. Surv.*, vol. 51, n.º 5, pp. 1-38, nov. 2018, doi: 10.1145/3241737.

[11] O. Topçu, U. Durak, H. Oğuztüzün, y L. Yilmaz, *Distributed Simulation: A Model Driven Engineering Approach*. Cham: Springer International Publishing, 2016.

[12] K. Snively y D. J. McDonnell, «A Parallel DEVS Approach for Cloud Simulation Standards», 2016.

[13] RedHad, «HPC Moves to the Cloud - What You Need to Know», *insideHPC*, 2017. https://insidehpc.com/white-paper/insidehpc-guide-hpc-moves-cloud/ (accedido abr. 06, 2020).

[14] Gigalight, «How Much Do You Know About Supercomputing, High Performance Computing and Cloud Computing?», *Medium*, nov. 05, 2018. https://medium.com/@Gigalight/how-much-do-you-know-about-supercomputing-high-performance-computing-and-cloud-computing-5618db07fc45 (accedido jun. 27, 2020).

[15] Redhat, «El concepto de OpenStack». https://www.redhat.com/es/topics/openstack (accedido jul. 02, 2020).

[16] P. Calegari, M. Levrier, y P. Balczyński, «Web Portals for High-performance Computing: A Survey», *ACM Trans. Web*, vol. 13, n.º 1, pp. 1-36, feb. 2019, doi: 10.1145/3197385.

[17] Microsoft Azure, «¿Qué es DevOps? Explicación de DevOps | Microsoft Azure». https://azure.microsoft.com/es-es/overview/what-is-devops/ (accedido jul. 02, 2020).

[18] Wikipedia, «Computación en la nube - Wikipedia, la enciclopedia libre». https://es.wikipedia.org/wiki/Computaci%C3%B3n_en_la_nube (accedido mar. 29, 2020).

[19] A. Sunyaev, *Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies*. Cham: Springer International Publishing, 2020.

[20] B. P. Zeigler, T. G. Kim, y H. Praehofer, *Theory of Modeling and Simulation*. Academic Press, 2000.

[21] J. L. Risco Martín, «DEVS - Discrete Event System Specification», p. 52, jul. 2019.

[22] J. L. Risco Martín y S. Mittal, «xDEVS». sep. 03, 2015.

[23] Google, «Servicios de cloud computing», *Google Cloud*, jun. 27, 2020. https://cloud.google.com/?hl=es (accedido jun. 27, 2020).

[24] Docker, «Docker», jun. 27, 2020. https://www.docker.com/ (accedido jun. 27, 2020).

[25] Kubernetes, «Kubernetes», *Kubernetes*, jun. 27, 2020. https://kubernetes.io/ (accedido jun. 27, 2020).

[26] HashiCorp, «Terraform», *Terraform by HashiCorp*, jun. 27, 2020. https://www.terraform.io/index.html (accedido jun. 27, 2020).

[27] J. L. Risco-Martín, S. Mittal, J. C. Fabero Jiménez, M. Zapater, y R. Hermida Correa, «Reconsidering the performance of DEVS modeling and simulation environments using the DEVStone benchmark», *SIMULATION*, vol. 93, n.º 6, pp. 459-476, jun. 2017, doi: 10.1177/0037549717690447.

[28] Wikipedia, «Dhrystone», *Wikipedia*. jun. 02, 2020, Accedido: jun. 30, 2020. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Dhrystone&oldid=960301458.

[29] J. Bonér, D. Farley, R. Kuhn, y M. Thompson, «El Manifiesto de Sistemas Reactivos», sep. 16, 2014. https://www.reactivemanifesto.org/es (accedido mar. 29, 2020).