Universidad Zaragoza

Trabajo Fin de Grado

Diseño, Despliegue y Monitorización de un Simulador Distribuido de Eventos Discretos

Design, Deployment and Monitoring of a Distributed Discrete Event Simulator

Autor

Paul Hodgetts Isarría

Director

Unai Arronategui Arribalzaga

Universidad de Zaragoza / Escuela de Ingeniería y Arquitectura
Grado Universitario en Ingeniería Informática

2022

AGRADECIMIENTOS

Quisiera agradecer, en primer lugar, al director de este trabajo Unai Arronategui por haberme dado la oportunidad de realizar este proyecto, si no fuera también por la excesiva paciencia que ha tenido conmigo en el transcurso del mismo. Me ha dotado de mucho conocimiento en el área de la simulación distribuida, así como de otras cuestiones que verán en esta memoria, aunque siempre poniendo por encima el cuidado personal.

También me gustaría agradecerle a mis amigos, familia y todas aquellas personas que pese a extrañarse por el significado del título de este trabajo, han dado un apoyo moral constante. Han jugado un papel esencial en lo que ya puede resumirse como una etapa de mi vida, y me han ayudado a que este proyecto se haga realidad. Quizá el más efectivo para mitigar el estrés haya sido mi gato Pixel, aunque sin su constante llamada de atención probablemente habría acabado el proyecto más rápido.

A todos les agradezco de nuevo su apoyo, y gracias a ellos las palabras que se recogen a continuación serán las de un trabajo que esperan de mí haberme convertido en un informático, un ingeniero, y una mejor persona.

Gracias.

RESUMEN

Las Redes de Petri se emplean en la industria como una herramienta matemática para la descripción y análisis de sistemas concurrentes y distribuidos. En este caso, es capaz de describir un modelo ejecutable en un simulador de eventos discretos. Tal modelo comprendería de un alto número de eventos para redes de gran escala, que implicaría la necesidad de un **simulador distribuido**.

Se presenta la oportunidad de definir un nuevo paradigma para la composición de problemas sobre eventos distribuidos; y como corresponde, una nueva escena de desarrollo para este modelo de computación. Son numerosas las posibles optimizaciones y mejoras que se pueden plantear para la ejecución de este entorno. Para ello, se asientan cuestiones de **diseño** del simulador distribuido.

Como es natural, esta serie de simulaciones distribuidas no pueden ser probadas sin unas herramientas de **despliegue** adecuadas. Se deberá de tener en cuenta que la versatilidad requiere flexibilidad a la hora de poner en marcha el sistema. Un despliegue debe de conocer la completa estructura de nodos de simulación, que vendrá marcada por una definición de red.

De la misma manera, no se puede asertar un correcto funcionamiento del sistema sin las necesarias herramientas de **monitorización**. Se añade a la problemática de despliegue nuevas nociones para la infraestructura de la monitorización, y el correcto tratamiento sobre el trazado y métricas del simulador, manteniendo las capacidades de escalabilidad del mismo.

En conclusión, se parte de una perspectiva en la que se debe completar la noción del diseño de un simulador, para luego definir su metodología de despliegue, y finalmente, dar un énfasis para establecer los componentes de monitorización que expondrán el comportamiento del sistema.

Índice

1. Introducción

En esta memoria se recogen unos conceptos de diseño de un simulador distribuido de eventos discretos de trabajos previos, necesarios para el desarrollo de la metodología de despliegue y de un sistema de monitorización escalable. Será en estas dos cuestiones donde se ponga especial énfasis. Todo ello, ha sido cumplimentando el límite de las 10.000 palabras propio de un Trabajo Fin de Grado.

1.1. Motivación

El empleo de Redes de Petri como modelo conceptual para definir un sistema presenta un nuevo paradigma entre las variantes de simuladores distribuidos de eventos discretos. No solo aporta una nueva forma de componer simulaciones distribuidas, sino que también introduce la posibilidad de integrar numerosas optimizaciones.

La lógica de una aplicación distribuida se reduce a una estructura de transiciones con un orden de sensibilizaciones marcadas por una función, denominada ***Linear Enabling Function*** (LEF). Esta simplificación permite hacer una fácil manipulación del comportamiento de la aplicación, y he aquí la motivación que establece el uso de este tipo de sistema.

Ahora bien, poco aporta un sistema distribuido que no pueda ser probado, con lo que se adhiere la necesidad de definir la metodología del despliegue. Éste deberá contar con las particularidades del sistema que podrán imponer una red de nodos dependiente de la simulación a ejecutar.

En la misma línea, el despliegue puede facilitar la interacción con casos reales del sistema, pero para asesorar la validez de los resultados se requiere de una infraestructura que extrapole la información necesaria para el entendimiento del comportamiento del sistema. Estas adiciones al simulador establecen lo que se denomina un sistema de monitorización.

1.2. Objetivos

Como es de remarcar, se habla de tres puntos esenciales para llevar a cabo la realización de este proyecto:

- Se ha de determinar un **diseño** que defina el simulador distribuido. Se escoge la palabra "diseño" deliberadamente, ya que las cuestiones de implementación del simulador son objetivo de otro trabajo. Aquí se van a establecer algunos conceptos que servirán como fundamento para las dos siguientes cuestiones.

- Se ha de solucionar la problemática del **despliegue** para el correcto uso del simulador. Como es natural, el sistema deberá de poder establecerse sobre varias máquinas al mismo tiempo.

- Finalmente, habrá que representar la operativa del simulador en lo que es el sistema de **monitorización**. Se deberá de poder realizar las necesarias consultas y comparativas para el análisis del comportamiento del simulador.

1.3. Contexto

Existe un esfuerzo previo en un trabajo[1] que trata de desarrollar un simulador de eventos discretos basado en Redes de Petri. La diferencia fundamental que separa dicho simulador del que se va a explicar en este trabajo es que este se enfocará en las capacidades de alta escalabilidad y un uso optimizado de recursos.

Como ya se ha podido ver, este proyecto emplea las Redes de Petri como modelo de implementación para el simulador distribuido. Las Redes de Petri han sido popularizadas como herramienta matemática para la descripción y el análisis de sistemas concurrentes y distribuidos. Son capaces de realizar una síntesis de un programa concurrente y/o distribuido para su mejor comprensión, pero es el foco de este trabajo dar un paso más allá y emplearlo como definición formal para una serie de simulaciones.

Este trabajo se apoya en, y ha apoyado, proyectos paralelos de otros alumnos[2][3]. Estos trabajos realizaron gran parte del esfuerzo del diseño e implementación del simulador distribuido, añadiendo sus respectivas optimizaciones.

Al partir de un proyecto programado en el lenguaje Rust, se genera la necesidad de aprenderlo para los cambios que se planteen, o su integración en las cuestiones de monitorización.

2. Conceptos y estado del arte

Se han de introducir y recoger todos los conceptos y todas las tecnologías que han sido objeto directo de un estudio previo, para luego plantear la operativa que marcará el desarrollo de este proyecto.

2.1. Simulación distribuida de eventos discretos

El simulador distribuido de este proyecto parte de un modelo basado en Redes de Petri. La motivación principal es que ofrece una simple representación de lo que sería una red de causalidad entre eventos, y realizar este esfuerzo de manera distribuida permite dividir la problemática en subredes, de tal manera que cada subred es capaz avanzar su propagación de eventos hasta encontrar la necesidad de ser propagado por, o propagar a, los eventos de otro nodo. La conceptualización de dicha propagación de eventos se define por lo que se denominan transiciones, como las que son propias en una Red de Petri.

La propagación de eventos entre transiciones ocurre cuando estas se ven sensibilizadas. Es decir, cuando se cumplen los requisitos de causalidad previos en la simulación para accionar la transición. La racionalidad de dicha transición cuando se encuentra sensibilizada se denomina disparo. Esto realizaría una propagación de eventos a las siguientes transiciones en el grafo de causalidad. Asimismo, el concepto de la sensibilización de una transición con su correspondiente disparo se puede representar formalmente por una función de activación, o ***Linear Enabling Function*** (LEF).

Al fin y al cabo, toda Red de Petri puede representarse como un grafo de LEFs, que acaba siendo una mayor simplificación para el modelo del simulador.

Recuperando la noción de que el simulador sea distribuido, la división de la simulación en las subredes de LEFs necesita acarrear la información de los nodos vecinos en los que realizar una propagación externa.

2.2. Los Tres Pilares de la Observabilidad

Antes de adentrarse a los diferentes términos que se emplearán a lo largo de la definición del sistema de monitorización, hay que aclarar que existe cierta controversia con el uso de los términos “Monitorización” y “Observabilidad”. El problema existe debido a que ambos términos se usan indistintamente cuando se encuentran bajo el foco de márketing, dificultando la difusión de una definición técnica y objetiva.

En resumen, la Monitorización engloba toda la noción de recuperación de información en un sistema informático para su posterior entendimiento, esto pudiera ser resuelto tradicionalmente por registros o *logs,* o más recientemente, con mecanismos de trazado y métricas. Por otra parte, la Observabilidad se puede establecer como una parte del estudio de la Monitorización, enfocado principalmente a la obtención de información para conocer el estado de una aplicación determinada. Es decir, se centra en el factor de la semántica de un programa, normalmente mediante el uso de trazas.

El “Trazado” no se libra tampoco de controversia, viéndose mezclado con el término de Telemetría. Este último término es más difícil de defender en una perspectiva técnica, quedando como un común indicio de vocabulario para la marquetería de esta área. El trazado debería definirse como un conjunto de estructuras de información que adhieren un contexto del sistema a lo largo de una de sus posibles operaciones.

A continuación, se introducen los “tres pilares de la Observabilidad”, que se resumen en las tres estructuras de información que conforman la noción de **trazas**.

2.2.1. Métricas

Se define métricas como la estructura de información para la Observabilidad que recoge un conjunto de claves-valor asociados a una estampilla de tiempo (*timestamp*). Permiten conocer el estado del sistema en un momento dado, o incluso representar el contexto de un evento.

2.2.2. *Spans* (o lapsos de tiempo)

Los *spans* o *lapsos de tiempo* son similares a las métricas en el sentido de que contiene una estructura de claves-valor, pero esta vez asociado a dos estampillas de tiempo: una que marca un principio de operación y otra de fin. Alternativamente, se pueden encontrar como una estampilla de tiempo de inicio y un campo con el transcurso de operación desde el momento de inicio. Este lapso de tiempo es el que suele marcar el tiempo de vida de una operación en el concepto de trazado.

2.2.3. Logs (o registros)

Esta estructura de información precede incluso a los conceptos de monitorización, siendo un clásico recurso de obtención de información sobre el comportamiento de un programa. No tienen porqué estar asociados a ninguna indicación temporal, pero su versatilidad reside en su representación que frecuenta como una línea de texto libre. Tampoco quiere decir que no sea una estructura manipulable y preparada a consultas, es tarea del desarrollador generar logs con una presentación consistente.

2.3. Aproximaciones y herramientas de Monitorización Distribuida

Conociendo las diferentes estructuras para representación del contexto de un sistema, se pueden poner en uso práctico con la herramienta adecuada. Para ello se hace un estudio de las diferentes tecnologías que se presentan en la actualidad. Se recuerda que en el foco principal del estudio, la monitorización ha de resolver prioritariamente las cuestiones de depuración y entendimiento del simulador distribuido.

2.3.1. Zipkin

Zipkin[17] es una herramienta para la definición y visualización del trazado de un sistema. Aunque se defienda como una solución para el análisis de

latencias de un sistema, ha aportado al escenario de la monitorización una nueva manera de presentar el contexto de un sistema distribuido. Si bien ha ayudado a sentar las bases sobre la instrumentación de una aplicación para su trazado, ésta ha influenciado a una alternativa, e incluso se ha visto reemplazada por ésta. Como se verá, se trata de OpenTelemetry (punto 2.3.3).

La herramienta sigue guardando relevancia por aquellos proyectos que la han adoptado desde un primer momento, y mantiene soporte con otras de las soluciones modernas que se van a mencionar en los siguientes puntos.

2.3.2. Prometheus

Este nombre puede responder a dos cuestiones: las herramientas que componen el sistema de métricas de Prometheus[15][16], o la definición abierta (y considerada *de facto* estándar) para las métricas. Ambos puntos son dignos de estudio pues ofrecen un sistema sobre el que asentar parte de la problemática de la monitorización, y resuelve la definición de dichas métricas para la instrumentación.

Esta tecnología realiza la recuperación de métricas mediante un componente *agente* que se encargará de realizar consultas frentes a dichos nodos instrumentados.

Las métricas de Prometheus cuentan con la particularidad de que se enfocan en un valor principal acompañado de etiquetas para su posterior procesado.

Prometheus adicionalmente cuenta con un lenguaje de consulta para las métricas, acompañado de funciones matemáticas, estadísticas y de agregación. Éste es PromQL.

2.3.3. OpenTelemetry

Se entiende como un estándar de Observabilidad para la definición de trazas apoyadas en *spans*, métricas asociadas y logs. Su funcionamiento de trazas se apoya en la decisión del desarrollador de incorporar en una cabecera el

TraceID, identificador de la traza, pudiendo recoger su misma noción el contexto de varios sistemas que trabajan en conjunto para resolver una operación del sistema.

Éste nace de la influencia de Zipkin y del antiguo estándar de métricas de Jaeger, OpenTracing. El proyecto de OpenTelemetry[5][6] cuenta con herramientas adicionales que emplean el estándar, como es el siguiente caso.

2.3.3.1. Colectores: Importadores, Procesadores, Exportadores

El colector[20][21] es un componente intermediario para el sistema de monitorización, que permite definir múltiples fuentes de trazas. Opcionalmente se incluyen una serie de operaciones de procesamiento para dicha información. Finalmente, se añaden diferentes puntos de exposición: visualizadores, almacenamiento, o incluso otro colector para formar una jerarquía de colectores.

2.3.4. Jaeger

Fuera del catálogo comercial, es la opción más popular para la representación y visualización de trazas[7][8]. Se defiende como una herramienta de trazado punto-a-punto. En la comparativa hubiera formado parte el que fuera el anterior estándar de métricas que Jaeger empleaba: OpenTracing. Este estándar fue descartado y ya no se soporta desde este mismo año 2022.

Jaeger parece ser un componente esencial para el sistema de monitorización, ya que se requiere de poder visualizar de las trazas, pero se pondrán a prueba sus capacidades en futuros apartados.

2.3.5. Grafana

De entre todas las herramientas de visualización para la monitorización, destaca una propuesta abierta: Grafana[13][14]. Disfruta numerosas integraciones para la obtención y representación de la información, lo que la hace popular para el propósito de monitorización de infraestructura. Aunque suele configurarse para aceptar tecnologías de métricas como Prometheus, es capaz de representar los *Tres Pilares de la Observabilidad* (punto 2.2).

2.4. Herramientas y entornos de despliegue distribuido

Antes de adentrarse a realizar una comparativa de diferentes entornos de despliegue, hay que aclarar las competencias necesarias para el despliegue del simulador.

En otros proyectos, el despliegue se podría reducir a un apartado comúnmente llamado “puesta en marcha”. Sin embargo, en este proyecto se desarrolla esta cuestión a lo largo de toda la memoria por dos principales motivos:

- El despliegue es uno de los tres focos principales establecidos en los objetivos de este proyecto.
- **El despliegue depende** en gran medida en decisiones que se han de tomar en la etapa **del diseño del simulador**. Esta correlación se resalta por infrecuencia, pero es necesaria.

También se debe mencionar el uso de la palabra despliegue, y ante el foco de otros proyectos que también traten de este tema, aquí se hace justicia al uso de la palabra. En numerosos contextos el concepto de despliegue se expande para relacionarse con lo que realmente se denomina orquestación. Es decir, se suele adherir artificialmente los elementos de gestión de recursos, balanceo de carga, o replicación a esta noción, pero aquí no serán necesarios.

Aquí se recoge la palabra despliegue con el único propósito de gestionar los componentes que conformarán la arquitectura del sistema, sobre la maquinaria disponible. En resumen, **sólo es necesario establecer que los diversos componentes se pongan en marcha con la arquitectura esperada**.

Una vez aclaradas estas ideas, se podrá realizar una comparativa de las tecnologías que pueden resolver la problemática del despliegue.

2.4.1. Vagrant

Esta herramienta permite definir un entorno de máquinas virtuales mediante el uso de *scripts*, usualmente denominados *Vagrantfile*. Es compatible con diferentes proveedores de virtualización (*libvirt*, *VirtualBox*, *VMWare*, *HyperV*,

entre otros). Portando gran versatilidad, los *scripts* basados en Ruby permiten definir los recursos de las diferentes máquinas virtuales bajo una lógica programada. Para escenarios *multi-máquina* (virtuales, no *hosts*) es común definir una lista de configuraciones, con la que se facilita establecer los parámetros de su entorno, como puede ser de interés: simple manejo de la red.

Ahora bien, está pensado para realizar despliegues sobre una única máquina *host*, lo que la hace ideal para definir un entorno de pruebas, pero no para un despliegue en producción.

2.4.2. Docker (Compose, Swarm)

Es la tecnología de contenerización más popular actualmente (aunque en recesión), y es compatible con la mayoría de entornos de despliegue populares[9][10]. Los contenedores se crean a partir de *imágenes* que residen en un repositorio, y éstas son generadas a partir de un proyecto que contenga un *script* de construcción denominado *Dockerfile*.

Se pueden realizar despliegues de múltiples contenedores gracias a la herramienta de **Docker Compose**. Mediante un fichero de especificación se definen los diferentes contenedores a lanzar, incluyendo su imagen base, sus argumentos, configuración de red y volúmenes de almacenamiento, entre otras opciones. Esta herramienta está incluída en la instalación base de Docker. Al igual que Vagrant, está pensada para despliegues sobre una única máquina, haciendo la opción apta sólo para entornos de prueba y desarrollo.

Para aprovechar la anterior definición de Compose en una arquitectura multi-máquina de *hosts*, se incluye otra herramienta: **Docker Swarm**. Ya manejando cuestiones de orquestación, Swarm es capaz de definir *servicios*, que los entiende como las diferentes posibles instancias de un contenedor (réplicas). En su documentación defiende la posibilidad de establecer un entorno multi-máquina con asignación de IP fija para los contenedores. Esta cuestión cobrará relevancia en futuros apartados.

Para finalizar, queda mencionar otra herramienta que aprovecha las definiciones de Compose para realizar un despliegue en un entorno multi-máquina. Es **Overnode**, que en la comparativa con Swarm, éste no

posee de las típicas capacidades de orquestación pero sí permite una fácil gestión del despliegue de los contenedores en el entorno descrito. Esta herramienta es independiente a las anteriores, y aunque parezca fuera de lugar, también cobrará su sentido en futuros apartados.

2.4.3. Kubernetes

Actualmente el orquestador de contenedores por excelencia, Kubernetes[18][19] ofrece un entorno completo con la máxima modularización de sus componentes para adaptarlo a las necesidades de los posibles despliegues de los que se requiera. Pero, posibilidad no significa facilidad ni simplicidad. Kubernetes acapara incluso más funcionalidades de orquestación que Swarm, y acaban siendo cuestiones que no cobran relevancia para el desarrollo de este proyecto.

Hay que mencionar que en este área se está acercando al modelo de *microservicios*, una arquitectura de componentes heterogéneos que conforman el sistema distribuido en cuestión. Se establece gracias a la previa decisión de disgregar en gran medida los programas en lo que son los microservicios, siendo tolerantes a reinicios en cualquier momento, y por tanto, independientes de un estado interno.

Una vez más, esto queda fuera del interés de este proyecto, ya que como se verá, aquí se parte de un modelo homogéneo de nodos de simulación adheridos a un contexto, y por el momento, no tolerantes a reinicios.

3. Análisis de Requisitos

El desarrollo de este proyecto debería ofrecer un sistema de monitorización escalable para el simulador distribuido, contando con todos sus aspectos de despliegue. De aquí nace la necesidad de introducir una serie de requisitos para cumplimentar las expectativas.

3.1 Requisitos Funcionales (RF)

Se recogen los siguientes requisitos funcionales, que serán mencionados en su debido cumplimiento en la etapa de Diseño**:**

- **RF1** – El usuario podrá desplegar el simulador en un entorno de múltiples máquinas.
- **RF2** – El usuario podrá visualizar las trazas de ejecución de nodo(s) del simulador.
- **RF3** – El usuario podrá comparar visualmente las trazas de múltiples nodos al mismo tiempo.
- **RF4** – El sistema guardará las trazas en almacenamiento persistente.
- **RF5** – El usuario podrá visitar trazas de ejecuciones pasadas.
- **RF6** – El usuario podrá filtrar las trazas disponibles por tiempo (fecha y hora de recepción de las trazas).
- **RF7** – El usuario podrá filtrar las trazas disponibles por los nodos de simulación participantes en dicha traza.
- **RF8** – El usuario podrá visualizar los valores de los atributos necesarios para la depuración del sistema correspondientes a tiempo del sistema en las trazas.
- **RF9** – El usuario podrá opcionalmente visualizar las trazas en base al tiempo del simulador.
- **RF10** – El usuario podrá desplegar el sistema de monitorización escalable a la medida del despliegue del simulador.
- **RF11** – El usuario podrá realizar una traducción de una especificación formal de Redes de Petri a un formato apto para el simulador.

3.2 Requisitos No Funcionales (RNF)

Se añaden estas otras consideraciones como requisitos no funcionales:

- **RNF1** – El sistema de simulación validará y superará su correspondiente entorno de tests.
- **RNF2** – Todos los componentes del sistema deberán de poder escalar en número manteniendo el funcionamiento esperado.
- **RNF3** – El sistema priorizará el uso de recursos[1] para la ejecución de la simulación.

[1] Se define como recursos a la disposición de capacidad de cómputo (CPU), disponibilidad de memoria volátil, ancho de banda en red, y latencia en red.

4. Diseño del Sistema

A continuación, se dividen las problemáticas de diseño en dos secciones principales que rodearán el correspondiente estudio para la arquitectura del simulador distribuido, y para el sistema de monitorización. Queda por mencionar las cuestiones de despliegue que se reparten en un doble esfuerzo por comprender los elementos que conforman el despliegue en ambos apartados principales. Es decir, se desarrollan los conceptos de diseño del simulador, después su despliegue; entonces se pasa el foco al sistema de monitorización, y finalmente se reintroducen conceptos de despliegue para definir la resultante operativa de la total arquitectura del sistema.

4.1. Diseño de la Arquitectura del Simulador

En la mira de los objetivos establecidos (*apartado 1.2*) se recoge que el hincapié en la problemática del diseño se acota para introducir los conceptos necesarios en las consecuentes cuestiones de despliegue y monitorización.

La concepción de un simulador distribuido de eventos discretos que toma por modelo Redes de Petri favorece la división de la representación total de la simulación en subredes. Dichas subredes ponen como objetivo avanzar todas las transiciones del simulador posibles dentro de cada nodo, hasta que requieran de la propagación de transiciones externas (residentes en otros nodos). Con lo cual, se requiere identificar toda relación de transición externa con la IP y puertos en los que se expone la instancia de simbot (nodo de simulación) que contiene dicha transición.

El comportamiento esperado sería entonces recibir una especificación de la subred que el nodo pueda interpretar para generar una representación interna de sus transiciones como LEFs, y además que permita definir las diferentes conexiones que el nodo deba establecer para el envío de mensajes (cuando se requiera de una transición externa).

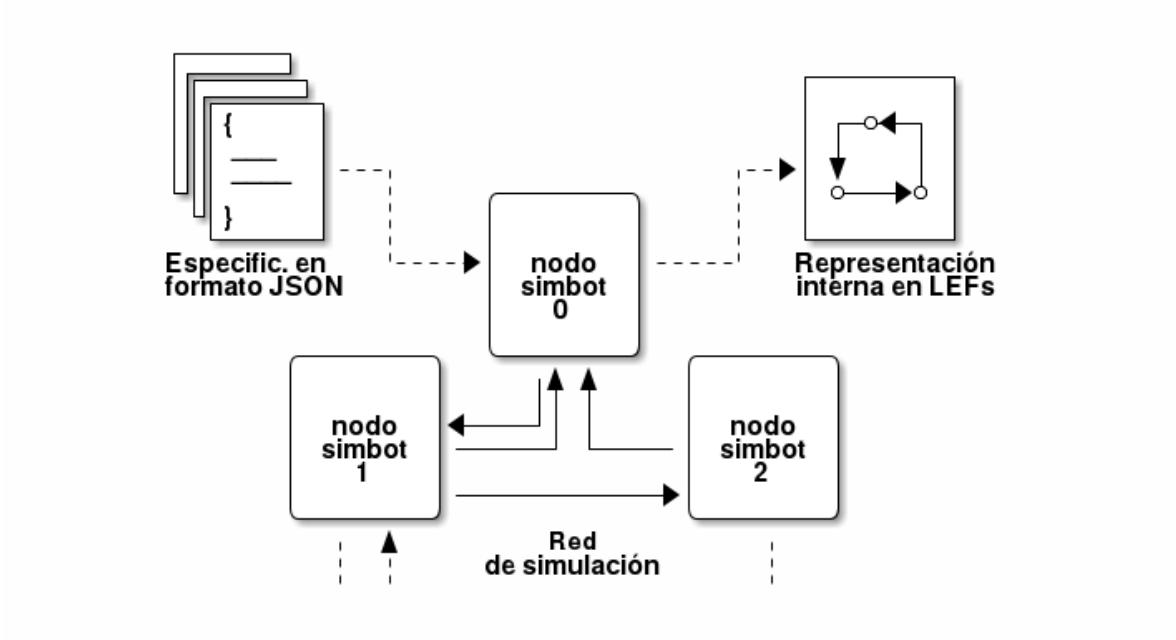

Diagrama 1. Arquitectura de nodos de simulación.

Como se puede observar en el anterior ejemplo (*Diagrama 1*), el nodo *simbot* 0 ha recogido su especificación de subred de un fichero, para luego generar una representación interna del modelo en LEFs. También, ese mismo fichero le especificaría una referencia a una transición externa, que se encuentra en el nodo *simbot* 1. Los otros nodos comparten la misma lógica y ambos por sus especificaciones acaban requiriendo de transiciones que se encuentran en el nodo *simbot* 0.

4.1.1. Especificación de redes a simular

Existe una especificación de Redes de Petri originaria de un proyecto pasado[1], que permite dividir una red en subredes. Este concepto es algo que se trata de replicar en este proyecto, y por tanto se toma tal modelo, contando con ejemplos existentes. Pero para poder ser interpretado por el simbot, hace falta dividirlo en las subredes necesarias, y realizar una compilación a una especificación más cercana a la representación interna del *simbot* en base a LEFs. [RF11]

Adicionalmente, se decide en esta etapa que la especificación de los nodos que contengan transiciones externas deba ser mediante IP y puerto, a

oposición de usar un nombre DNS. Esto se debe a que la operación de resolución de nombres es muy costosa en tiempo para los órdenes de magnitud en tiempo en las quiere trabajar el simulador. Realizar una traducción en cada envío es costoso en tiempo, y una traducción de nombres a IP previa a la ejecución es una operativa que se puede relajar al tiempo de despliegue (*apartado 4.1.4*).

Otra cuestión involucrada en la compilación de la especificación de Redes de Petri, es que se debe realizar una traducción de las transiciones "globales" a transiciones equivalentes externa o internamente. Esto quiere decir que cuando se divide una Red de Petri en diferentes subredes, los índices de las transiciones varían a una identificación interna en la subred, y también, las aquellas transiciones que se propagan a través de diferentes subredes, requieren una referencia externa.

4.1.2. Arquitectura dependiente de la simulación

Al realizar un despliegue de una simulación concreta, se están creando instancias de un mismo programa (*simbot*), donde la comunicación entre los diferentes nodos se verá definida por el propio fichero de especificación de la simulación. Esto quiere decir que no existe una conexión total entre todos los nodos de manera directa, pero sí una relación indirecta. Así, las conexiones entre los diferentes nodos dibujarían un grafo en el que existe por lo menos un camino para relacionar dos nodos cualesquiera. Es, por tanto, necesario considerar que se construirá una arquitectura para el despliegue en base a un sistema de nodos de simulación con conexiones arbitrarias.

4.1.3. Despliegue del Simulador

En este proyecto se ha de recalcar la dependencia que tiene la problemática del despliegue con las decisiones desarrolladas en la etapa de diseño del simulador distribuido.

Dicha dependencia existe principalmente en la definición de la red para un caso de simulación. Como ya se ha explicado, cada nodo del simulador debe conocer las IPs/puertos de los nodos involucrados en las transiciones externas de su subred. Por tanto, entra en efecto una nueva problemática: definir en

despliegue una correspondencia de IP/puerto a cada nodo de simulación y que sea consistente con las especificaciones de subredes de simulación.

Tal cuestión no es problema para las tecnologías de Vagrant y Compose, que pueden establecer una red sobre una interfaz virtual. Esto es, sólo en una máquina, que relega las opciones para ser entornos de prueba o desarrollo.

Mientras tanto, Docker Swarm sí promete una configuración de red en la que se asigne a cada contenedor *simbot* una IP/puerto, independientemente de la máquina en la que se encuentre. Sin embargo, a pesar de que se menciona el soporte para este caso, no es posible aún recogerlo en los ficheros de despliegue (heredados de Compose). Existe una discrepancia entre la especificación en fichero de texto y la versión en argumentos por comando para la creación de la red en Docker Swarm, y todavía no ha sido resuelta.

Es por esta razón que se ha decidido probar por la alternativa de Overnode, que a pesar de ser poco conocida, resulta más simple y acotada a las necesidades de este proyecto. Éste emplea una tecnología para la gestión de red en contenedores que frecuenta en Kubernetes: Weave. [RF1]

4.2. Diseño de Monitorización del Simulador

Una vez recogidos todos los conceptos y requisitos adheridos al contexto, se ha de desarrollar el diseño del sistema. Se introducirán todos los componentes del sistema en una vista inicial a alto nivel, en lo que es el estudio de la Arquitectura. Posteriormente, se detallará en profundidad el papel que han de jugar dichos componentes.

4.2.1. Arquitectura del sistema de monitorización

Como bien se ha descrito en el anterior apartado, se ha de desarrollar una arquitectura en torno a un sistema existente. Por tanto, se introducen nuevos componentes que irrumpan lo más mínimo en la comunicación entre los nodos de simulación. Estos son los **clientes** que emiten las trazas (los nodos de simulación a instrumentar), un componente de **visualización** de las trazas,

un punto de **almacenamiento** persistente de las trazas, y por último un árbol de componentes intermedios conocido como **colectores**.

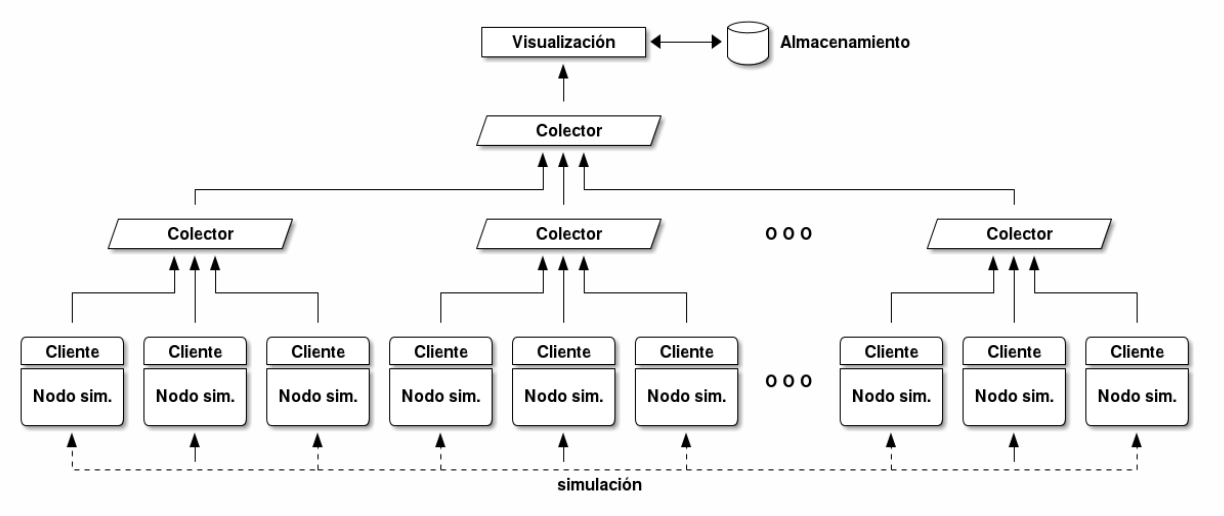

Diagrama 2. Arquitectura de trazado con un ejemplo de simulación.

Como se puede apreciar en el ejemplo (*Diagrama 2*), se describe que, para un número determinado de nodos, existe una jerarquía correspondiente de colectores. Los colectores fronterizos recolectan las trazas que emiten los clientes, y las propagan a colectores superiores, hasta llegar a un colector raíz que finalmente emite el conjunto de trazas al componente de visualización. En el momento en el que el componente de visualización procesa las trazas, las guarda en un sistema de almacenamiento persistente.

4.2.2. Instrumentación para la observabilidad remota

En este sistema, se comportan como clientes, o emisores de trazas, los nodos de simulación. En otras palabras, son los componentes que se han de instrumentar empleando las tecnologías disponibles para ello. OpenTelemetry es la especificación de observabilidad que se va a establecer el formato de las trazas.

Un gran punto de la problemática es decidir qué información se debe de recoger en las trazas[RF8]. Poca información no ayuda a formar unas trazas con las que evaluar el estado de la simulación. Por contra, mucha información, aparte de dificultar la recabación de datos significativos, puede suponer un mal uso de los recursos de los que se dispone. Al escalar la

simulación, ésta puede dar grandes saltos en órdenes de magnitud en el número de eventos emitidos. Todo exceso de información que se incluya en la instrumentación es exceso que tomará buena medida de los recursos.

Para el estudio de este proyecto se van a incorporar los siguientes elementos a trazar:

> **La vida del nodo de simulación**. Es necesaria una traza que permita entender el tiempo de vida de cada nodo, incorporando atributos que permitan identificarlo, así como los parámetros de invocación. Esta traza será la que acapare al resto de elementos a trazar en su respectivo nodo.
>
> **Envío de eventos**. Un nodo de simulación puede establecer una causalidad en otro nodo de la misma simulación. Interceptar esta información facilita el entendimiento de la comunicación entre los diferentes nodos. En palabras de sistema de simulación: se recoge todo evento que realice el disparo de una transición externa.

Cabe destacar que el sistema de simulación que se emplea en este proyecto podrá incorporar cambios o mejoras, por lo que se contemplará al llevar a cabo la implementación, una fácil incorporación en la instrumentación de cualquier nuevo elemento necesario.

4.2.3. Visualización

Si bien Jaeger se define como la herramienta libre por excelencia para el trazado distribuido, se le pondrá a prueba en sus capacidades. Esta herramienta deberá mostrar las trazas que los clientes han emitido[RF2], para posteriormente compararlas y comprender la ejecución de la simulación. Adicionalmente, deberá de poder realizar el filtrado necesario para consultar las trazas deseadas, sea por nodo participante, operación o fecha. [RF6] [RF7]

Habiendo definido previamente la información que se transmite en las trazas desde los clientes, se puede prever en cierta medida qué se debería recibir en la herramienta de visualización.

La problemática aquí reside en dos cuestiones fundamentales:

1. Poder comparar visualmente las trazas de diferentes nodos de simulación. [RF3]
2. Poder escoger si visualizar las trazas en base a tiempo real, o en base a tiempo de simulación. [RF9]

Ambas cuestiones no se pueden resolver con Jaeger. La primera resulta un problema mayor, que puede afectar a cualquier otro proyecto de características similares al simulador distribuido. Esto se explica en el siguiente apartado. Mientras, la segunda cuestión resulta algo más específica y simplemente Jaeger no da opción nativa para cambiar la línea temporal.

4.2.3.1 Aspecto de comparativa de trazas

Se ha mencionado que se pondría a prueba las capacidades de Jaeger, y es aquí donde se ha de desvelar el verdadero papel que juega este componente en cualquier arquitectura de un sistema de trazado.

Jaeger se defiende con ejemplos de sistemas **heterogéneos**, donde se trata de obtener trazas de operaciones que se llevan a cabo a lo largo de diferentes servicios de un sistema. No ofrece ninguna opción para realizar una comparativa directa entre diferentes trazas sobre una línea de tiempo.

Un ejemplo habitual

Un usuario solicita darse de alta en un sistema web. Una petición llega al servidor web y aquí comienza la traza. En conjunto con las peticiones internas del sistema, se propaga el identificador de traza (*TraceID*), como puede ser a un sistema caché intermediario, para finalmente realizar el registro en una base de datos. Se emiten los mensajes de confirmación al volver. En Jaeger se visualiza el recorrido de las diferentes peticiones que forman parte de la operación “dar de alta al usuario”, con cada servicio mostrando sus trazas en un color diferente.

Es el programador quien ha de asegurarse que el *TraceID* se propague entre los diferentes servicios, y que estos lo adopten en su instrumentación, para finalmente poder mostrarlos bajo una misma traza en Jaeger.

El caso del simulador distribuido

En la arquitectura del simulador sólo existen nodos de *simbot*, y por tanto, se define como un sistema **homogéneo**.

En los nodos de simulación, la traza no comienza bajo una petición, sino bajo el tiempo de vida de dicho nodo. Por tanto, cada nodo inicializa una traza con un identificador diferente. Para poder imitar el comportamiento de un sistema heterogéneo, habría que establecer un algoritmo de consenso entre todos los nodos. Esto no es efectivo ya que no está asegurado que ni todos los nodos se desplieguen al mismo tiempo, ni que todos existan a la vez.

En definitiva, la solución para la comparativa de trazas de diferentes nodos no se desarrolla ni en los clientes, ni en el componente de visualización. Es uno de los papeles que tendrá que jugar el **colector**.

4.2.4. Colectores

Normalmente, el colector suele emplearse como intermediario para transmitir trazas a lo largo del sistema distribuido. Este punto despierta varias facilidades a considerar:

1. **Alta escalabilidad**. Para evitar que un alto número de nodos sobrecargue un único colector, se pueden desplegar varias instancias de éste de manera jerárquica para ir transmitiendo las trazas de los nodos a colectores, de colectores a colectores, hasta acabar en un colector raíz que transmita las trazas al componente de visualización.
2. **Independencia tecnológica**. Como se detallará a continuación, el colector facilita la recepción de trazas de diferentes fuentes, y la emisión a diferentes destinatarios si fuera necesario. Esto es, con el soporte para diversas tecnologías en ambos puntos.
3. **Procesamiento personalizado**. Como también se explicará, los colectores son capaces de realizar una serie de operaciones sobre el flujo de trazas que transmite.

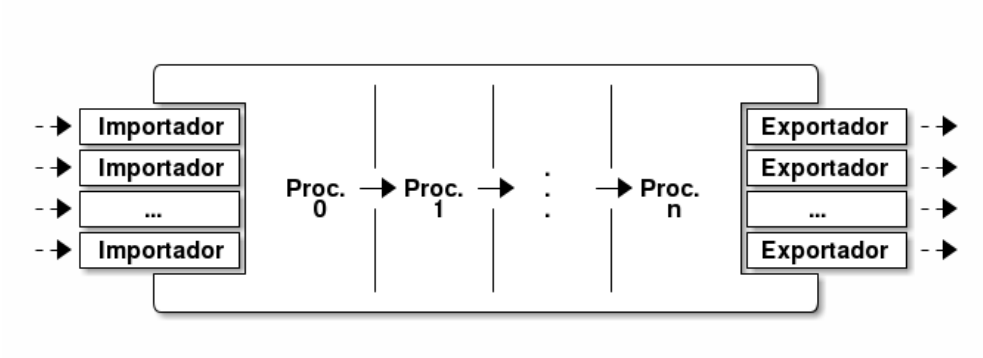

Diagrama 3. Modelo de un *pipeline*.

Esta serie de ventajas las proporciona OpenTelemetry Collector. Una instancia de éste se compone de múltiples flujos de procesamiento o ***pipelines*** (*ver Diagrama 3*). Es decir, un colector no está asociado a un único comportamiento con su flujo de trazas, sino que puede adoptar diferentes flujos de trabajo para las trazas. Una *pipeline* está compuesta por los tres módulos principales del colector: **importadores**, **procesadores** o **exportadores**.

4.2.4.1. Importación

Los importadores, como bien indica su nombre, se encargan de recibir e importar las trazas a una *pipeline* determinada. En el caso de este proyecto interesa establecer un importador de métricas en el formato de OpenTelemetry, que ofrecerá un puerto de red esperando la llegada de dichas trazas.

4.2.4.2. Procesamiento

Los procesadores otorgan una funcionalidad de tratamiento de datos adicional. Permite realizar diferentes operaciones sobre las trazas recibidas: filtros, alteración de atributos, aglomeración para no congestionar la red al emitir, entre otros muchos.

Es en este punto donde se pueden resolver los problemas introducidos por el componente visualizador (*apartado 4.2.3*). Al tener control de las diferentes

trazas, y una posible manipulación, se pueden introducir dos nuevos procesadores para su implementación:

- Un procesador que realice la agrupación de las trazas bajo una única, que se denominará como “adopción”. Éste se conocerá como ***adoptprocessor****. [RF3]

- Un procesador que transforme las medidas de tiempo real por tiempo de simulación, con lo que un segundo en el visualizador correspondería a una unidad temporal de simulación. Este procesador se conocerá como ***vtimelineprocessor****. [RF9]

* Estos nombres siguen la nomenclatura del proyecto de OpenTelemetry Collector.

4.2.4.3. Exportación

En el último paso de tratamiento de las trazas en una *pipeline,* éstas serán emitidas a otro servicio. En este caso, será o bien a otro colector para completar la jerarquía, o al componente de visualización.

4.2.5. Almacenamiento

El componente visualizador requiere de un punto de almacenamiento para las trazas, pero es de interés establecer que éste sea persistente. Esto permite un estudio sobre ejecuciones de simulaciones pasadas. [RF4][RF5]

Partiendo de que este componente acompaña al visualizador Jaeger, se opta por Badger. Éste es una base de datos empotrada basada en estampillas de tiempo, común en aplicaciones Go de este área como es el propio Jaeger.

4.2.6. El sistema de Monitorización en Despliegue

Retomando de nuevo la problemática de despliegue, hay que incorporar los nuevos componentes que conforman la arquitectura final del sistema. [RF10]

Se recuerda que en un intento de solucionar la problemática de red para un despliegue de contenedores, se apuesta por la solución de Overnode. La inclusión de colectores y del componente de visualización no introduce nuevas cuestiones para la operativa, más allá de lo que pueda ser las configuraciones concretas en su despliegue. Es decir, no aparecen nuevas restricciones para la tecnología de despliegue, son capaces de adecuarse a la gestión de IP por contenedor a pesar de soportar resolución de nombres.

La única cuestión que hay que asegurar es de exponer el componente de visualización a una red desde la que se desee acceder (mediante un navegador web).

5. Implementación del Simulador, su Despliegue y su Monitorización

Ya se han descrito todos los conceptos que marcan una decisión de alto nivel en el proyecto. Ahora se ha de realizar la implementación y de adentrarse a conocer la problemática que llega a hacer realidad el sistema. Se recuerda que entre los objetivos de este trabajo, se hace foco a la implementación del sistema de despliegue y el de monitorización, dejando para el simulador mejoras o cambios puntuales.

5.1. Implementación de funcionalidades del simulador

Entre otras cosas, se ha modificado en la implementación el mecanismo de identificación del nodo. Anteriormente, el nodo establecía la IP que empleaba para exponerse en la simulación mediante la recuperación de información de las interfaces de red de la máquina en la que se estaba ejecutando. Esto, a parte de ceder el control para decidir con qué IP se debe identificar, establecía un comportamiento inesperado para los casos en los que una máquina tuviera varias redes configuradas. La solución a este problema fue introducir un nuevo parámetro en la invocación del programa para un nodo de simulación: **el índice de nodo**. Como todo nodo ya recibe una lista con las IPs y puertos con los que se identifican los nodos que conforman la simulación, un índice ayuda a apuntarle al nodo cual es éste entre todos los que conoce.

5.1.1. Procesado de RdP a formato JSON

Recuperando del contexto de un proyecto previo[1], la definición de las Redes de Petri en una especificación formal en fichero de texto se heredó para realizar una fácil traducción e incorporación ya en formato de LEFs. Para ello se añadió a ese proyecto original una nueva clase Java (*ExportarRdP*) que exporta en formato JSON la instancia de objeto de la simulación procesada en un formato cercano a la definición de red de LEFs. De esta manera se pueden aprovechar todos los ejemplos de Redes de Petri que se usaron en ese proyecto, además de adoptar la especificación para nuevos experimentos.

5.2. Instrumentación de los nodos de simulación

La definición de trazas se vió facilitada por el uso de una librería denominada *tracing*[24], común en la instrumentación de aplicaciones Rust. Para poder emitir las trazas en el formato de OpenTelemetry, se emplean las librerías de *opentelemetry*[25] y *opentelemetry-otlp*. La conversión del formato entre librerías ocurre gracias a *tracing-opentelemetry*. La librería de *tracing* ofrece integración con otros formatos de trazas o métricas, como el propio de Jaeger o Prometheus.

Ahora queda responder la cuestión de qué instrumentar. Interesa establecer unas trazas bajo los criterios introducidos en la fase de diseño (*apartado 4.2.2*). Mediante un macro que ofrece la librería de *tracing*, se puede asignar un *span* al tiempo de vida en ejecución de una función. Esto resulta en la instrumentación de:

- *simulate_period(...)*: Quitando la toma de argumentos en invocación y la inicialización para el cliente de instrumentación, comprende la vida del nodo de simulación. Sus argumentos se agregan automáticamente mediante la macro a los campos de información de la traza, portando una fácil identificación de los nodos en la visualización.
- Envío de mensajes: Interesa establecer un *span* con un tiempo de vida acotado en la mayor medida posible. Se podría reducir a la ejecución de la función *send_msg(...)*, la función implementada de más bajo nivel dentro del proyecto en el envío de mensajes entre nodos, pero aún se puede ajustar un poco más. Se declara un *span* con una definición manual de comienzo y fin, rodeando las funciones de envío de mensaje que ya residen en librerías de terceros.

5.3. Nuevos procesadores para el colector

La problemática de la comparación de las trazas de diferentes nodos no se puede resolver directamente desde el componente de visualización. Del mismo modo, también se quiere poder mostrar la línea temporal de las trazas empleando los tiempos de simulación, en vez de los reales.

5.3.1. Agregación de trazas: *adoptprocessor*

Este procesador se introduce bajo la problemática de agregar la información de diferentes trazas como si fueran una única. Establecer el concepto de "una única traza" requiere entender cómo se identifica una traza, cómo herramientas como Jaeger son capaces de identificar qué *spans* pertenecen a una traza u otra. Todo se debe al campo ***Trace*ID**. Es una cadena de carácteres numérica y semialeatoria que se genera a partir de diferentes campos, tratando de establecer su unicidad. Esta generación mantiene una estructura cifrada que se emplea como método de comprobación de integridad. Aunque todos los campos de información adicionales sean correctos, asignar manualmente un valor arbitrario al *TraceID*, puede suponer una mala representación por fallos de comprobación. La *TraceID* debe ser generada correctamente.

Modificar el campo de *TraceID* para todos los *spans* que se procesen en el colector, con el fin de que todos coincidan en uno, permitiría reagruparse e identificarse como elementos de la misma traza en las herramientas de visualización. Pero como ya se ha descrito, no vale cualquier traza, se debe generar correctamente. Por ello, se obvia ese problema y en cambio se ***adopta***.

Adoptar es el concepto que se ha denominado al proceso de alteración del campo de *TraceID* para todo *span* bajo uno válido de todos los recibidos. Es decir, una de las trazas prevalecerá con su *TraceID* y lo establecerá al resto, haciendo en efecto una *adopción*.

El consenso bajo el cual se rige la *adopción* de trazas es muy simple: El colector inicializa un *adoptante* vacío, la primera traza comprobará el valor del *adoptante* y al encontrar que está vacío, lo sustituirá y registrará su propia *TraceID* como la de *adoptante*. Las consecuentes trazas obtendrán un valor no vacío para el *adoptante*, y por tanto se *dejarán adoptar* modificando su *TraceID* por el del *adoptante*. En resumen, la primera traza que llegue al colector, *adopta* al resto.

Este es un comportamiento que se propaga sin problema a los escenarios de alta escalabilidad, en los que existe una jerarquía de colectores. Se realizarían las diferentes adopciones por los colectores, hasta que en el colector raíz haya un adoptante que prevalezca sobre el resto y sirva como traza única a representar en el componente de visualización.

Adicionalmente, hay que considerar que en el proceso de adopción, el nombre de "operación" de traza relacionado con su correspondiente *TraceID* puede quedar con valor diferente a la del adoptante. Por ello, se aprovecha en el proceso de adopción de trazas para establecerles el contexto de operación con el nombre del nodo *simbot*, siguiendo la estructura "`simbot_IP:puerto`".

5.3.2. Traducción a tiempo de simulación: *vtimelineprocessor*

Como ya se conoce del anterior apartado, se puede alterar los campos de las trazas a procesar. El tiempo de simulación se conoce por la información adicional que provee la traza. Se alteran los campos de tiempo real realizando una conversión para la representación de tiempo de simulación, guardando los reales como nuevo campo de información adicional a la traza. Un segundo en la línea temporal en la herramienta de visualización, equivaldrá a una unidad de tiempo de simulación.

Por tanto, aunque aparentemente se esté empleando una línea temporal real, se estará realizando una equivalencia para representarlo. Los tiempos comenzarán en la medida de del tiempo en el que se haya iniciado el colector. De esta manera la herramienta de simulación tomará como más reciente la simulación que se está ejecutando actualmente. Cada segundo a partir de ese instante representará el avance de tiempo de simulación en una unidad.

5.4. Desarrollo de Despliegue del Sistema

Para poder disfrutar de una solución versátil de despliegue, se empaquetan los componentes desarrollados (*simbot* y *collector*) en imágenes de contenedor. Para ello se desarrollan los ficheros *Dockerfile* en los directorios raíz de las fuentes de ambos componentes. Las imágenes que se generan pueden depender de otras total o parcialmente, ya que se mantiene una generación intermedia de *deltas* para cada imagen. Se añaden las herramientas necesarias para la construcción del proyecto y el código del proyecto. Se genera el binario correspondiente y se copia a un *delta* que parte de una imagen sólo con lo necesario para la ejecución del binario, resultando en un tamaño de imagen final reducido. Este último paso es importante, se está hablando de diferencia abismal en tamaño. Por ejemplo, con *simbot*, si no se

realizara ese último paso de higiene, la imagen pesaría poco más de 1 GB. Con el paso del binario a una capa limpia se reduce el tamaño a un total de 35 MB.

Se hace una primera prueba de operativa con estas imágenes (más la oficial de Jaeger) mediante un fichero de especificación de Docker Compose. En dicho fichero se establecen las diferentes instancias de los contenedores a lanzar con los parámetros necesarios (argumentos de invocación, imagen base, red), generando un primer caso de arquitectura en local. Para habilitar el sistema de almacenamiento persistente para las trazas, se debe proveer Jaeger con algunas variables de entorno dedicadas a Badger.

Una vez comprobada su correcta puesta en marcha, se aprovecha tal definición de Docker Compose para traducirse en lo que serán los servicios de Overnode; y con esto se podría realizar un despliegue en un entorno multi-máquina.

5.5. Puesta en marcha

Primero se ha de hacer un rápido repaso de todas las dependencias para la operativa del sistema. Se requiere de Docker, las imágenes Docker del *simbot*, el colector adaptado y Jaeger; como Docker Compose y Overnode presente en las máquinas de las que se disponga. Para un correcto funcionamiento, se espera el uso de una distribución GNU/Linux con los módulos de kernel *dummy* y *openvswitch*, necesarios para la operativa de Weave, que gestionará la red de los contenedores.

A continuación se crea el cluster de Overnode con la correspondiente orden de unión. En una de las máquinas, (cualquiera, no hay noción de máquina principal) se deben de incorporar los ficheros de especificación derivados de Compose y ahora adaptados a Overnode. Para proveer de los ficheros de subredes de la simulación, estos se deberán especificar como volúmenes de punto de montaje con el correspondiente formato de Docker Compose.

Una orden de puesta en marcha de Overnode se encargará de copiar los ficheros de especificación para los contenedores que residirán en las correspondientes máquinas, y finalmente los iniciará.

Con ello, Jaeger debería de poder ser accesible como web en el nodo en el que se encuentre desplegado. Se recuerda que debido a la naturaleza de los colectores, se puede especificar múltiples puntos de salida, y por tanto podrían existir múltiples instancias de Jaeger si fuera necesario.

En el Anexo A, *Guía de la metodología*, se explica en detalle este proceso con las debidas configuraciones y órdenes, acompañado de un ejemplo.

6. Experimentación

En este punto se van a recoger las pruebas que se han realizado con el fin de demostrar la utilidad del sistema que se ha desarrollado en este proyecto. Se recuerda que previamente a toda experimentación, existe un entorno de pruebas con un ejemplo funcional de tres nodos *simbot*, un colector, y Jaeger. Para ver en detalle la configuración de estos casos, se recomienda revisar los anexos A, B, C y D.

Se han realizado las correspondientes acciones para iniciar la operativa de 6 máquinas físicas (*ver Anexo E*) situadas en un laboratorio del centro, que servirán como entorno de experimentación.

6.1. Extendiendo la monitorización

Para un primer experimento se ha decidido establecer una extensión al actual sistema de monitorización. Las implicaciones previstas son las de poder recoger métricas en la infraestructura de colectores, y poder adquirir un nuevo componente de visualización para las trazas. El propósito de este experimento es demostrar la capacidad de disgregación e independencia de los componentes que componen el sistema de monitorización.

Se va a emplear un ejemplo de infraestructura para la monitorización basada en métricas que provee Overnode. Dicho ejemplo se compone de varios exportadores de métricas (*node_exporter* para las máquinas, *cadvisor* para los contenedores), un servidor Prometheus para la recogida de métricas, y finalmente, un componente de visualización Grafana.

Dicho ejemplo está pensado para un caso de 3 máquinas, extenderlo para este escenario de 6 máquinas sólo requiere especificar en la configuración de Prometheus que deberá rescatar las métricas de 6 puntos diferentes.

Ahora ya, se pueden realizar las adaptaciones del sistema para aprovechar estos nuevos componentes.

En el caso del colector, simplemente se ha de definir un nuevo apartado en su configuración dedicado a la "*Telemetría*":

```
service:

  [...]

  telemetry:
    metrics:
      address: 0.0.0.0:8888
      level: basic
```

Con esto el colector publica sus métricas en el puerto 8888. De nuevo en la configuración de Prometheus, se ha de añadir un campo para recuperar las métricas del colector:

```
- job_name: collector
  scrape_interval: 5s
  static_configs:
  - targets:
    - collector:8888
```

Esto hará que Prometheus realice su acción de recogida de métricas del colector cada 5 segundos.

En Grafana, ya con Prometheus configurado como fuente de datos, se puede crear un nuevo *dashboard* con paneles que empleen las métricas de los colectores (*ver Anexo H, Imagen 4*).

Por otra parte, se puede hacer la visualización de trazas también desde Grafana (emplean un *fork* de Jaeger). Ahora bien, se puede añadir como fuente de trazas Jaeger o, lo que resultaría más tentativo para no depender de otro componente de visualización, se podría configurar el sistema de almacenamiento de trazas de Grafana: Tempo. Con éste último habría que configurar el colector raíz para poner como punto de exportación a Tempo, pero se escapa del interés para experimentación al ser un producto comercial.

Aun así, queda la opción de emplear Jaeger como fuente. El soporte para esto viene habilitado por defecto, sólo se requiere especificar nombre y puerto del contenedor de Jaeger. Entonces en la sección de “Explorar” se puede acceder a una interfaz casi idéntica a la de Jaeger (*ver Anexo H, Imagen 5*).

6.2. Preparación de Monitorización de alta escala

En este segundo experimento se va a poner a prueba la teoría sobre la que se sienta la arquitectura de colectores del sistema de monitorización en escala. Para ello, se va a realizar un caso de 6 nodos *simbot*, 3 colectores (2 hojas, y uno raíz), y una instancia de Jaeger.

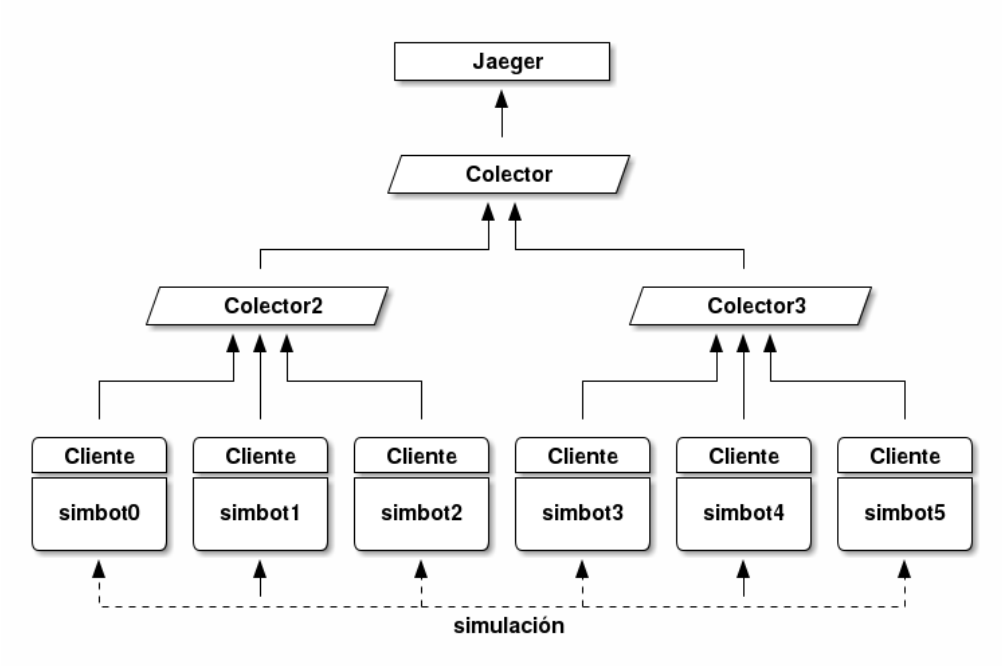

Diagrama 3. Ejemplo del sistema de monitorización en escala.

Primero, se ha de generar la definición para los tres colectores. Uno (la raíz) mantendrá la misma configuración que en el experimento anterior, esperando una fuente de datos de OpenTelemetry en el puerto por defecto, empleando el procesador *adoptprocessor*, y emitiendo las trazas al contenedor de Jaeger.

Los otros dos colectores, pueden tomar esta configuración para hacer sólo un cambio: en vez de emitir las trazas a Jaeger, las ha de emitir al colector raíz:

```
exporters:
  otlp:
    endpoint: collector:4317
    tls:
      insecure: true
```

Para aprovechar los esfuerzos del primer experimento, se habilita la opción de *Telemetría* también para estos colectores, y se añaden también a la configuración de Prometheus.

Esto permite ver que los colectores reciben y emiten las trazas como es esperado. Por ejemplo, un despiste en un fichero de configuración enseguida se hace notar viendo como 3 colectores reciben trazas, pero solo 2 las emiten.

De cualquier modo, con un correcto funcionamiento del sistema de colectores se puede hacer un análisis de la carga que éstos reciben. Si bien los colectores hoja reciben trazas del mismo número de nodos, no implica que reciban el mismo número total de trazas. Cada nodo tiene un comportamiento determinado en la simulación, y éste no tiene porqué ser simétrico al resto de nodos. El panel de métricas para colectores desarrollado en el experimento anterior ofrece la información de que el colector 2 está recibiendo muchas más trazas que el colector 3 (*ver Anexo H, Imagen 6*).

Con ello se concluye que para este experimento, para establecer una carga de trazas sobre los colectores más equitativa, habría que reasignar algunos de los tres primeros nodos de simulación al colector 3. Similares conclusiones se podrían llevar a cabo con casos más extremos, con alto número de *simbots*, poniendo a prueba las capacidades y recursos de cada colector, requiriendo de un reajuste como el que se ha visto aquí.

6.3. Síntesis de una simulación errónea

Finalmente queda plantear el que sea el experimento de mayor relevancia, o por lo menos, aquel que muestre el propósito del sistema de monitorización que se ha desarrollado. Es decir, hacer un análisis de un posible comportamiento erróneo en el simulador.

Véase el siguiente caso, se está ejecutando una simulación con la misma arquitectura que la del experimento anterior. Se corrobora que el sistema de colectores funciona adecuadamente; emiten las mismas trazas que reciben. Sin embargo, el componente de visualización de trazas Jaeger ofrece unos resultados extraños, que se muestran en tiempo de simulación (Ver *Anexo I, Imagen 7*).

A simple vista se puede determinar que ha habido un fallo en la simulación que ha impedido que ninguno de los nodos finalice su ejecución. Esto se debe a la falta de trazas con el tiempo de vida del nodo, que solo se emiten cuando ha conseguido finalizar la ejecución con éxito.

Queda por tanto ver el estado en el que ha quedado el simulador, gracias a las trazas de los envíos de eventos:

Nodo origen	**Tiempo de simulador**	**Nodo destino**
simbot0	11	simbot5
simbot1	5	simbot0
simbot2	5	simbot0
simbot3	4	simbot1
simbot4	5	simbot0
simbot5	5	simbot0

Tabla 1. Último estado de los nodos de una simulación fallida.

Contrastando estos resultados (Tabla 1) con la Red de Petri de la que originan las subredes de la simulación, desvela que existe un comportamiento inesperado en uno de los nodos. Como se aprecia en la especificación de la Red de Petri, la primera subred (asignada a *simbot* 0) establece una sensibilización de transiciones externas al resto de subredes, siendo éstas idénticas en forma. Estas subredes (de la 1 a la 5) deberían avanzar del mismo modo en la simulación. Sin embargo, esto no encaja con los resultados obtenidos, ya que el *simbot* número 3 se encuentra una unidad de tiempo de simulación atrasado al resto (del 1 al 5). Se puede ver que está tratando de realizar una sensibilización de una transición externa en el *simbot* 1, cuando en ningún momento de la especificación de la Red de Petri, la subred 3 realiza una sensibilización de la subred 1.

Recogiendo la información anterior, se decide estudiar el contenido del fichero de subred 3 en formato de LEFs en JSON:

```
  "ia_trans_ext": [
    {
      "ia_trans": 1,
      "ia_red": 0,
      "ia_addr": "10.35.0.11:20000"
    }
  ],
  "ia_addr": "10.35.0.13:20003"
}
```

Ahí se puede observar que trata de sensibilizar una transición en el nodo 1 (10.35.0.11), en vez del valor esperado, nodo 0 (10.35.0.10). Reemplazando este valor por el correcto permite obtener el comportamiento esperado de la simulación (Ver *Anexo I, Imagen 8*).

7. Conclusiones

Este trabajo ha comenzado en una etapa temprana al diseño del simulador distribuido de eventos discretos, e incluso ha dado la oportunidad de participar en el artículo de investigación en los aspectos de depuración, despliegue y experimentación del simulador distribuido[4]. Aunque el énfasis aquí se establece sobre las implicaciones que tiene plantear su problemática de despliegue y monitorización. Para ello se ha requerido del aprendizaje de nuevas tecnologías como Rust[26], o el uso de otras ya conocidas como Golang[27].

Debido a la naturaleza de tal simulador, se han presentado varias cuestiones para la definición de un sistema de despliegue. En concreto, aquí se demuestra un estudio que impone una comparativa entre las tecnologías más populares, que librándose de complejidad, ha dado a lugar una solución sin cuestiones clásicas de orquestación. Así, Overnode ha logrado defenderse como la apuesta más simple para la gestión en contenedores en un entorno multi-máquina.

En cuanto a la monitorización, la problemática se sitúa a medio camino entre los requisitos de depuración para el simulador, y la oferta de funcionalidades de las herramientas habituales para la observabilidad. Jaeger ha demostrado resolver bien poco en cuanto a la presentación de las trazas del simulador en un punto único de comparación. Si bien se define como una herramienta para el trazada distribuido, la noción de distribuido la ha de imponer el desarrollador del sistema a instrumentar. Además, los ejemplos que se recogen comúnmente para demostrar la funcionalidad de este componente, suelen tratarse de arquitecturas heterogéneas en servicios, con control lineal del paso de mensajes.

Una vez más, en el caso del simulador, es necesario tener en cuenta las diferentes posibles configuraciones de conexión entre nodos, con lo que se ha de plantear una solución diferente. Por suerte, dicha solución reside en un componente que también solucionará las implicaciones de escalado para el sistema de monitorización: el colector. Introduciendo lo que se denominan procesadores de colector, se ha podido realizar una agregación de las trazas en una identificación única (proceso de adopción); permitiendo la comparativa de trazas en un punto único. Esta cuestión resuelve una

problemática de monitorización para todos los sistemas que no tengan una operativa de interacción lineal entre servicios.

Adicionalmente, se ha desarrollado un procesador que permite realizar una traducción de las trazas al uso de tiempos de simulación. También debido a la naturaleza del colector, se ha ganado en independencia tecnológica para introducir nuevos componentes de procesado, nuevas fuentes de datos, o nuevos puntos de emisión (visualizado y/o almacenamiento).

Todo ello se ha puesto a prueba en tres experimentos que demuestran los puntos de fácil adaptación y extensión del sistema de monitorización en despliegue, un correcto escalado del sistema de monitorización con una jerarquía de colectores, y finalmente, se ha realizado un caso del propósito del sistema de monitorización: identificar fallas en la simulación.

7.1. Planteamientos a futuro

Como es natural, este proyecto no está exento de mejoras o extensiones. Da lugar a un trabajo futuro dentro de las propias competencias de despliegue y monitorización. En primer lugar, si la adopción de los componentes de monitorización hubiera requerido de menos esfuerzo en desarrollo, hubiera existido un foco mayor en la instrumentación del simulador. Se recuerda que el simulador puede trabajar en altos órdenes de magnitud en eventos, con lo que del mismo modo, recuperar toda la operativa en trazas puede ser muy costoso para el almacenamiento. Un proceso más afinado o selectivo sobre qué instrumentar mitigaría esto, o incluso simplificaría el proceso de obtención de conclusiones desde el visualizador.

Por otra parte, los esfuerzos del primer experimento sólo son una breve demostración de lo que podría ser un sistema de monitorización con la ya definida observabilidad, pero adicionalmente con un uso de métricas para el control de recursos. Los paneles que se han desarrollado carecen de complejidad y se reducen al entendimiento de los colectores. Un buen uso de las métricas podría permitir tener una total imagen del contexto del sistema, tanto del estado de la simulación, como de la disposición de recursos.

8. Esfuerzo dedicado y Evaluación Personal

El total de las horas se recoge a continuación (ver Tabla 2), en lo que ha sido un esfuerzo distribuido en un periodo de tres años (2019-2022), con especial implicación en este último. Esto no se debe a ninguna de las implicaciones de trabajo en sí, sino que remite a cuestiones personales que han dificultado el desarrollo del trabajo en un lapso de tiempo habitual, y que han cesado en el transcurso del último año.

Tarea	Horas
Estudio de Bibliografía y Documentación	30
Aprendizaje de lenguaje Rust	25
Familiarización con código base	15
Diseño e implementación de exportador de Redes de Petri	2
Diseño e implementación de proc. de adopción de trazas	80
Diseño e implementación de proc. de tiempo de simulación	20
Diseño e implementación del sistema de despliegue	40
Diseño e implementación del entorno de pruebas	20
Depuración	50
Experimentación	70
Reuniones	60
Memoria	100
Total	512

Tabla 2. Dedicación de horas.

Aprovecho ya el uso de la primera persona para comentar la evaluación personal.

Este trabajo ha vivido lo suficiente como para ver la concepción del simulador hasta su implementación con la inclusión de ciertas optimizaciones. Desde el primer momento, el proyecto del simulador bajo el cual se sustenta este trabajo, ha sido merecedor de mi atención. El amplio rango de posibilidades que ofrece tal proyecto ha sido el fruto de mi motivación para realizar este trabajo.

Lo que mayor satisfacción me aporta sobre el trabajo es el problema fundamental para la comparativa de trazas. Algo que pudiera parecer tan obvio como “un botón” en Jaeger para diferentes trazas en una línea temporal, no existe. Esto es un problema que rápidamente he podido ver que se propagaba por su comunidad. Es por ello, que el haberme topado con este problema más amplio para la escena de la monitorización, y haberle aportado una solución, hace confiar en el fruto de mis esfuerzos.

Espero personalmente que todo lo que se expone en este trabajo sea de gran utilidad para todos los alumnos que puedan tomar un trabajo en el proyecto del simulador.

9. Bibliografía

[1] Sergio Herrero Barco. Desarrollo de un framework de simulación de sistemas de eventos discretos complejos. Trabajo fin de grado, Universidad de Zaragoza, 2020.

[2] Álvaro Santamaría de la Fuente. Diseño e Implementación de un Simulador Distribuido de Eventos Discretos con Mecanismos de Balanceo de Carga. Trabajo fin de grado, Universidad de Zaragoza, 2021.

[3] Fidel Reviriego Navarro. Diseño e implementación de un simulador distribuido de alta escala de sistemas de eventos discretos. Trabajo Fin de Máster, Universidad de Zaragoza, 2021.

[4] Paul Hodgetts, Hayk Kocharyan, Fidel Reviriego, Alvaro Santamaría, Unai Arronategui, José Ángel Bañares, and José Manuel Colom. Workload evaluation in distributed simulation of dess. *18th International Conference, GECON Online*, September 21-23, 2021.

[5] OpenTelemetry - https://opentelemetry.io/ Accessed: 2022-06-19

[6] Documentación de OpenTelemetry - https://opentelemetry.io/docs/instrumentation/rust/ Accessed: 2022-06-19

[7] Jaeger - https://www.jaegertracing.io/ Accessed: 2022-06-19

[8] Documentación de Jaeger - https://www.jaegertracing.io/docs/1.35/ Accessed: 2022-06-19

[9] Docker - https://www.docker.com/ Accessed: 2022-06-19

[10] Documentación de Docker - https://docs.docker.com/ Accessed: 2022-06-19

[11] Overnode - https://overnode.org/ Accessed: 2022-06-19

[12] Documentación de Overnode - https://overnode.org/docs/getting-started Accessed: 2022-06-19

[13] Grafana - https://grafana.com/oss/grafana/ Accessed: 2022-06-19

[14] Documentación de Grafana - https://grafana.com/docs/grafana/latest/ Accessed: 2022-06-19

[15] Prometheus - https://prometheus.io/ Accessed: 2022-06-19

[16] Documentación de Prometheus - https://prometheus.io/docs/ Accessed: 2022-06-19

[17] Documentación de Zipkin - https://zipkin.io/ Accessed: 2022-06-19

[18] Kubernetes - https://kubernetes.io/ Accessed: 2022-06-19

[19] Documentación de Kubernetes - https://kubernetes.io/docs/home/ Accessed: 2022-06-19

[20] OpenTelemetry Collector - https://github.com/open-telemetry/opentelemetry-collector Accessed: 2022-06-19

[21] OpenTelemetry Collector (versión comunitaria) - https://github.com/open-telemetry/opentelemetry-collector-contrib Accessed: 2022-06-19

[22] Alpine Linux - https://www.alpinelinux.org/ Accessed: 2022-06-19

[23] Ubuntu - https://ubuntu.com/ Accessed: 2022-06-19

[24] Librería tracing para Rust - https://docs.rs/tracing/latest/tracing/index.html Accessed: 2022-06-19

[25] Librería opentelemetry para Rust - https://docs.rs/opentelemetry/latest/opentelemetry/ Accessed: 2022-06-19

[26] Rust - https://www.rust-lang.org/ Accessed: 2022-06-19

[27] Golang - https://go.dev/ Accessed 2022-06-19

ANEXOS

Anexo A. Guía de la metodología de despliegue

En este anexo se tratará de explicar en detalle cómo lograr la operativa del sistema al completo, siendo el simulador más los componentes necesarios para la monitorización.

La operativa debería de poder lograrse en cualquier distribución GNU/Linux que tenga soporte para una versión reciente de Docker (aquí en uso versión 20.10). Las instrucciones tienen en cuenta una arquitectura de Intel/AMD X86 de 64-bits. El soporte de otras arquitecturas dependerá de las imágenes de los contenedores de terceros.

Si se desea, se pueden generar las imágenes Docker correspondientes para el *simbot* y el colector adaptado con una etiqueta propia, pero aquí se proveerá de las que se han generado a lo largo del proyecto.

Realizar un fichero de especificación para Docker Compose no es obligatorio, pero se recomienda para hacer una prueba inicial de la configuración de los contenedores, que de igual modo se aprovechará. Véase el caso con tres nodos *simbot*, un colector, y jaeger en el Anexo B. Los importantes campos a tener en cuenta son los de *command,* donde se establecen los argumentos del *simbot*; y el campo de red *ipv4_address* que permite la asignación de una IP estática.

Para probar en la máquina local un despliegue con Compose:

```
$ docker-compose up
```

Para acceder a Jaeger, según el ejemplo, usar la ruta `http://localhost:16686`.

Una vez probado, se puede traducir al formato de Overnode. Primero se genera un proyecto de Overnode con:

```
# overnode init
```

Esto generará tres ficheros: `overnode.yml`, `.env`, y `.overnodeignore`. Los dos últimos sirven para establecer variables de entorno y especificar ficheros a ignorar en el proyecto, respectivamente.

Dentro de este directorio de proyecto, se generará un directorio por contenedor, y cada uno de ellos contendrá la especificación concreta del contenedor, extraída del Compose. Por ejemplo, el colector se especificaría en `collector/collector.yml`:

```
version: "3.7"
services:
  collector:
    network_mode: bridge
    restart: unless-stopped
    environment:
      WEAVE_CIDR: 10.35.0.5/12
    image: registry.gitlab.com/paul.98.zgz/otelcol-contrib
    ports:
      - "4317:4317"
```

Aquí el cambio principal es la configuración de red, que ahora se debe especificar mediante una variable de entorno de Weave. Es necesario seguir el formato IP/máscara. Una vez se ha realizado la conversión para todos los contenedores, se han de listar en el fichero de proyecto `overnode.yml`:

```
id: overnode
version: 3.7

simbot:
  simbot0/simbot0.yml: 1
  simbot1/simbot1.yml: 2
  simbot2/simbot2.yml: 3
  collector/collector.yml: 2
  jaeger/jaeger.yml: 1
```

Los números que aparecen a la derecha de las rutas son los índices de las máquinas del cluster Overnode en los que se desplegarán los contenedores. Ya solo queda realizar el despliegue con:

```
# overnode up
```

El contenedor de Jaeger debería ser accesible empleando la **IP de la máquina** del cluster en el que se encuentra, en este caso la número 1.

Anexo B. Ejemplo de especificación de Docker Compose

```
version: '3'
services:
  simbot0:
    container_name: simbot0
    image: registry.gitlab.com/paul.98.zgz/simbot:latest
    command: 3subredes 100 0 [10.35.0.10:20000,10.35.0.11:20001,10.35.0.12:20002] 10.35.0.5:4317
    depends_on:
      - collector
    networks:
      dock_net:
          ipv4_address: 10.35.0.10

  simbot1:
    container_name: simbot1
    image: registry.gitlab.com/paul.98.zgz/simbot:latest
    command: 3subredes 100 1 [10.35.0.10:20000,10.35.0.11:20001,10.35.0.12:20002] 10.35.0.5:4317
    depends_on:
      - collector
    networks:
      dock_net:
          ipv4_address: 10.35.0.11

  simbot2:
    container_name: simbot2
    image: registry.gitlab.com/paul.98.zgz/simbot:latest
    command: 3subredes 100 2 [10.35.0.10:20000,10.35.0.11:20001,10.35.0.12:20002] 10.35.0.5:4317
    depends_on:
      - collector
    networks:
      dock_net:
          ipv4_address: 10.35.0.12

  collector:
    container_name: collector
    image: registry.gitlab.com/paul.98.zgz/otelcol-contrib
    ports:
      - "4317:4317"
    depends_on:
      - jaeger
    networks:
      dock_net:
          ipv4_address: 10.35.0.5

  jaeger:
    container_name: jaeger
    image: jaegertracing/all-in-one
    ports:
      - "16686:16686"
    environment:
      SPAN_STORAGE_TYPE: badger
      BADGER_EPHEMERAL: "false"
      BADGER_DIRECTORY_VALUE: /badger/data
      BADGER_DIRECTORY_KEY: /badger/key
    volumes:
      - ./badger:/badger
    networks:
      dock_net:
          ipv4_address: 10.35.0.6

networks:
  dock_net:
    driver: bridge
    ipam:
      config:
        - subnet: 10.35.0.0/24
```

Anexo C. Guía de la metodología de monitorización

Una vez se ha desplegado el sistema, podemos recuperar las trazas obtenidas en la simulación desde Jaeger. A continuación se muestran unas imágenes con los pasos a seguir para interactuar con las trazas obtenidas. Véanse los puntos en rojo marcados en las imágenes que acompañarán las explicaciones.

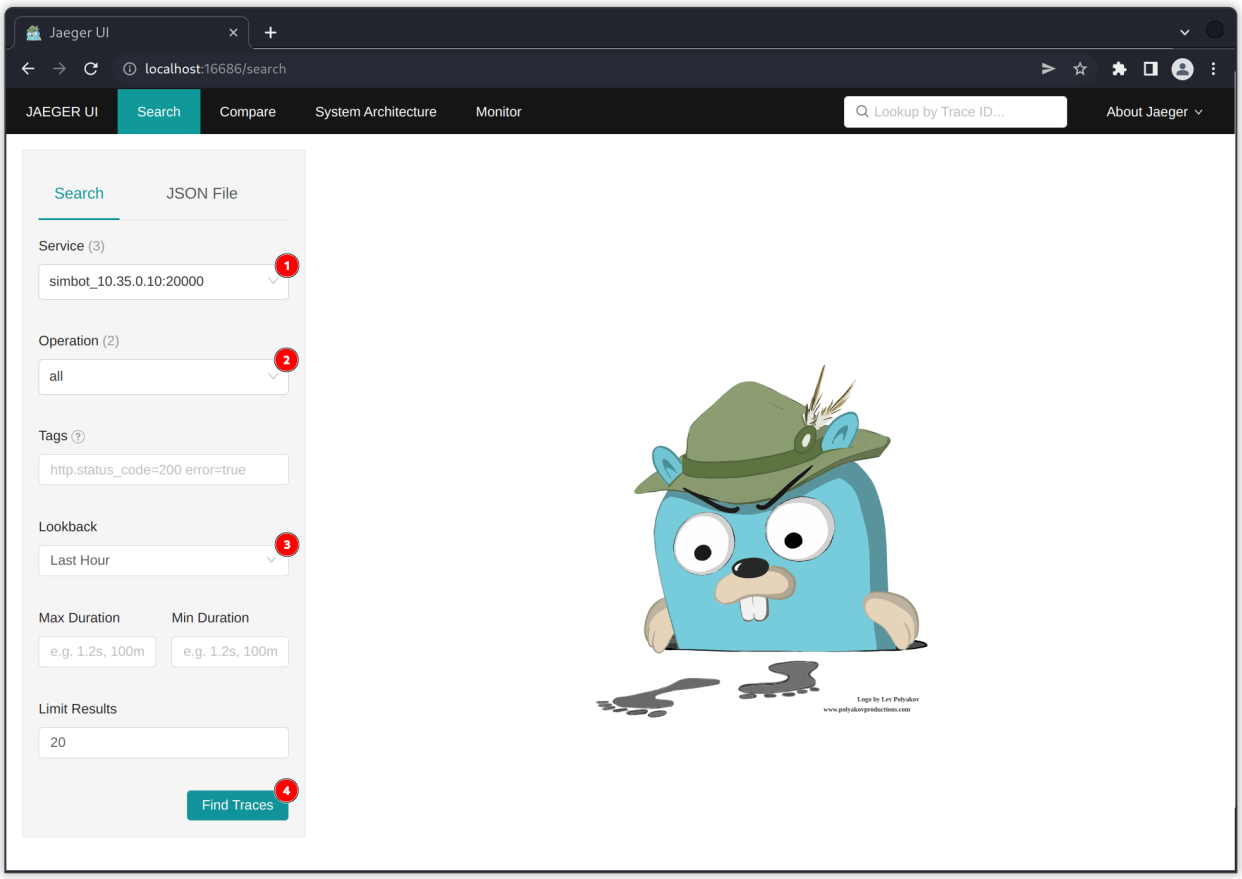

Imagen 1. Pantalla inicial del cliente web de Jaeger.

Por defecto, Jaeger muestra una pantalla sin resultado pero con todas las opciones de búsqueda disponibles. Para encontrar las trazas del simulador, se ha de realizar lo siguiente:

1. Seleccionar uno de los nodos *simbot* partícipes de la simulación.
2. Especificar de qué operación se quiere obtener las trazas.
3. Ajustar la búsqueda en un margen de tiempo.
4. Confirmar la búsqueda.

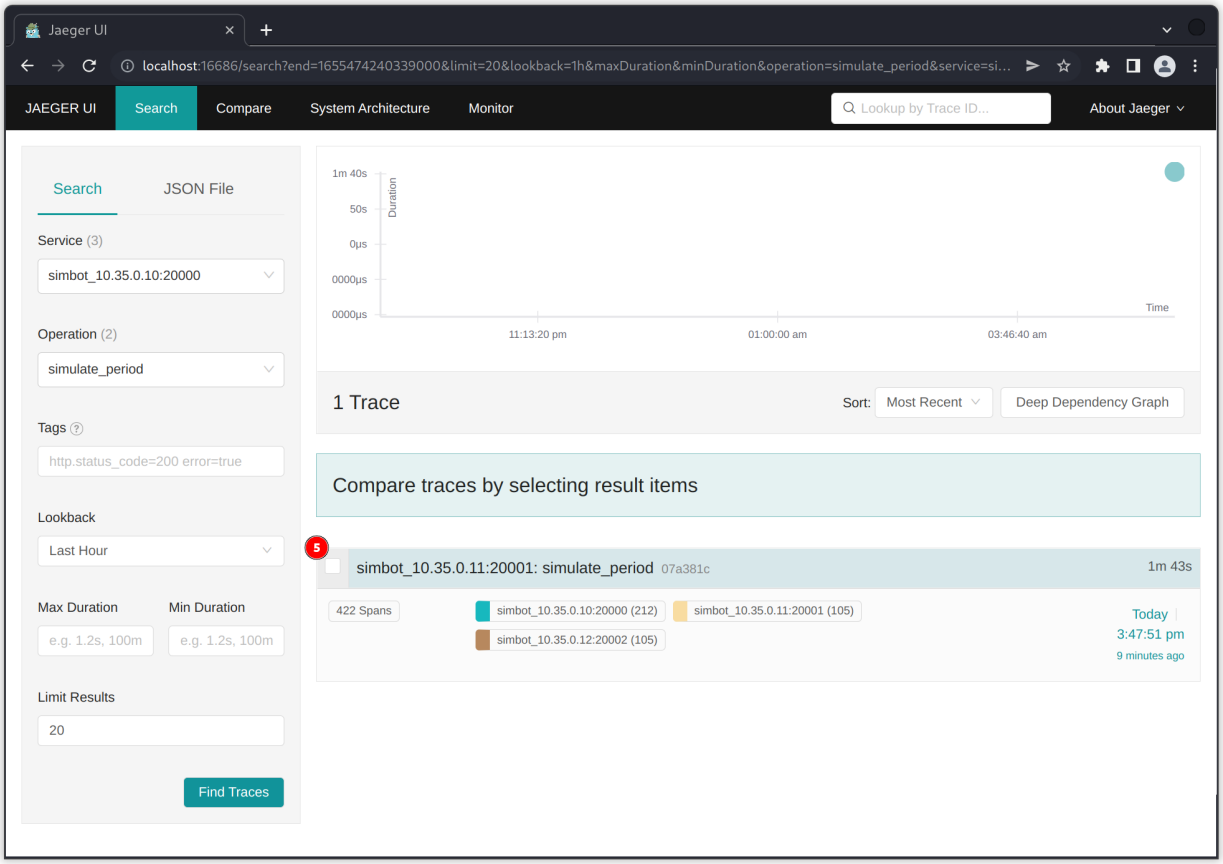

Imagen 2. Pantalla de resultados en una búsqueda de trazas.

Entre los resultados debería de aparecer la traza de interés. Se puede identificar mediante los nodos *simbot* que muestra, y la fecha de obtención.

5. Se selecciona dicha traza.

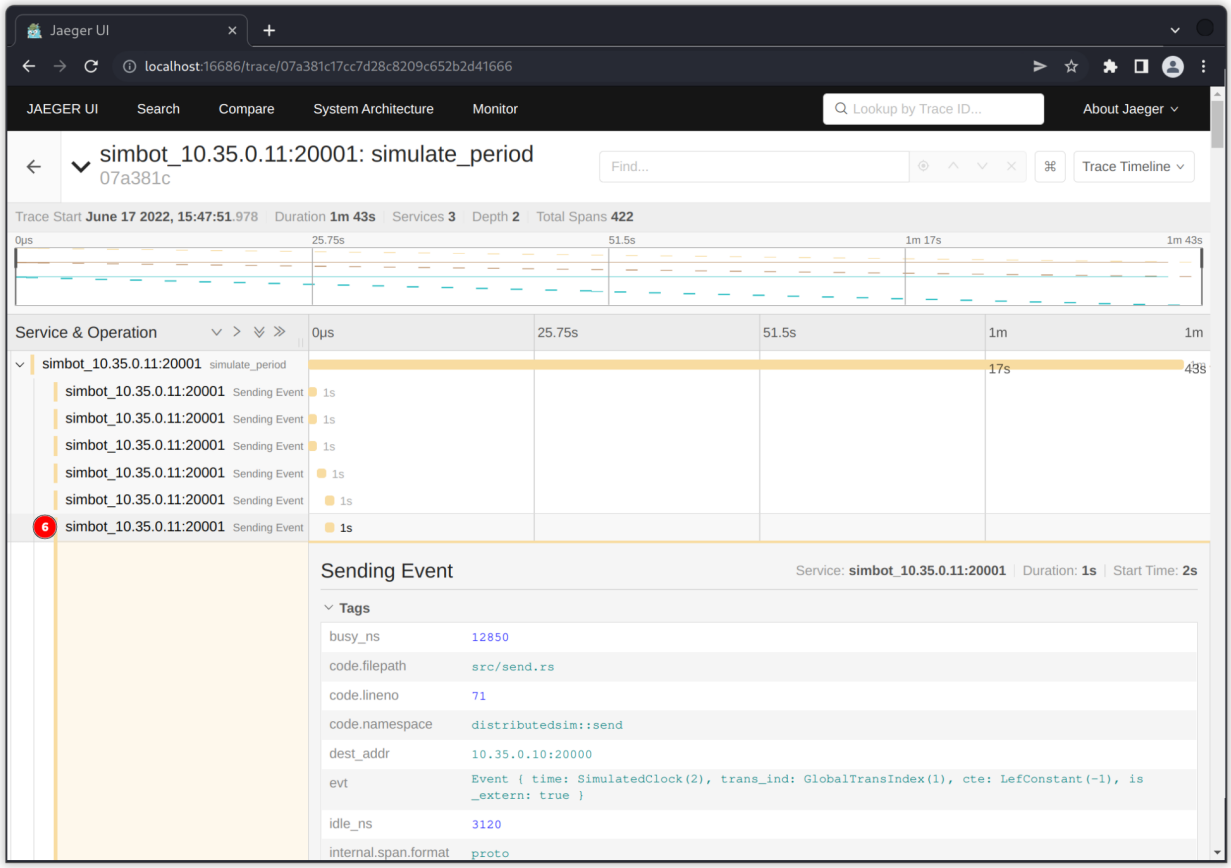

Imagen 3. Pantalla con la línea temporal de las trazas.

Aquí se muestra la traza deseada con sus *spans*.

6. Se selecciona un *span* para obtener información adicional del contexto.

Anexo D. Despliegue de entorno de experimentación

Para el entorno de pruebas, donde se tiene un control más manual para la operativa en tiempo de desarrollo, se cuenta con la herramienta Vagrant.

Esta herramienta es compatible con muchos sistemas operativos gracias a su integración con diferentes tecnologías de virtualización. En este caso se empleará VirtualBox como proveedor. En cuanto a las máquinas, se emplean imágenes oficiales ofrecidas por Vagrant, conocidas como *boxes*, de Ubuntu[23] 20.04 LTS. Se especifican en el fichero Vagrantfile, basado en Ruby, que permitirá definir las diferentes instancias de máquinas virtuales, su configuración y provisión de scripts de preparación.

En este proyecto se ofrece un Vagrantfile con un ejemplo similar al visto en el Anexo B: tres nodos *simbot*, un colector y Jaeger. Ahora bien, el colector y Jaeger convivirán con el primer *simbot*, cuya máquina viene preparada con más memoria virtual.

Para levantar el escenario, sólo hay que ejecutar lo siguiente en el directorio en el que se encuentre el Vagrantfile:

```
$ vagrant up
```

Nota: La primera vez que se ejecute puede que Vagrant necesite descargar la integración con VirtualBox. Si se confirma, lo hará automáticamente y lo instalará. Habrá que lanzar de nuevo la orden para cargar el escenario con éxito. También es probable que se requiera definir la red virtual para las máquinas desde la interfaz de VirtualBox, en este caso para `192.168.56.1/24`.

Para acceder a las máquinas, se puede hacer desde la interfaz de VirtualBox o bien se puede establecer una conexión SSH mediante la herramienta de Vagrant. Por ejemplo:

```
$ vagrant ssh node0
```

El servicio de Jaeger debería estar disponible accediendo desde un navegador web con la IP de la máquina virtual en la que se encuentre.

Anexo E. Personalización y extensión de la instrumentación de *simbot*

Como se ha mencionado anteriormente en la memoria, se ha aplicado la instrumentación en dos puntos del simulador, y de dos maneras diferentes.

En primer lugar, está el ejemplo de una macro que se ajustará al tiempo de vida de una función. Vease el caso de *simulate_period()*, en `src/lib.rs`:

```
#[tracing::instrument(fields(src_addr = list_nodes[index.0]))]
pub fn simulate_period(
    &mut self,
    lefs_name: &String,
    init_cycle: SimulatedClock,
    final_cycle: SimulatedClock,
    index: SubnetIndex,
    list_nodes: Vec<&str>,
) -> Result<(), Box<dyn Error>>{
```

La macro en cuestión es **`#[tracing::instrument(...)]`** que por sí sola basta para marcar la instrumentación, pero en el ejemplo se está añadiendo un argumento adicional: **`fields(src_addr = list_nodes[index.0])`**. Por defecto, la macro adhiere automáticamente los argumentos de la función asignada al contexto de la traza, pero con esta opción adicional en la macro se está añadiendo un nuevo campo para identificar la IP/puerto de origen del nodo que emite la traza.

En el segundo caso de instrumentación, se define manualmente el comienzo y fin de la traza. Vease el caso de *send_msg()* en `src/sed.rs`:

```
let span = tracing::info_span!("Sending Event", tag = ?msg.tag,
dest_addr = ?self.address, evt = ?msg.event, src_addr = ?msg.address);
let _enter = span.enter();

// Serialize and send the message
let encoded = bincode::serialize(&msg)?;
stream.write_all(&encoded)?;
stream.flush()?;

drop(_enter);
```

Con `tracing::info_span!(...)` se establece el contexto del *span*, así como el nombre de su operación. Después, con `let _enter = span.enter();` se marca el comienzo del *span*; y por último, se marca el fin con `drop(_enter);`. El tiempo de vida del *span* viene marcado en este caso por el tiempo de vida de la variable `_enter`.

Aunque no exista ningún caso en el código, es posible también establecer métricas asociadas a la traza. Éstas deben de existir bajo el contexto de un *span*. Un ejemplo sería:

```
let span = tracing::info_span!("...");
let _enter = span.enter();

// [...]
event!(Level::DEBUG, "Ejemplo de métrica asociada al span");
// [...]

drop(_enter);
```

Como se puede observar, basta con emplear una macro similar a la del contexto de un *span*. En este caso es: `event!(Level::DEBUG, "...");`.

Anexo F. Instalación de Overnode y consideraciones

Antes de instalar Overnode, se requiere de Docker. Éste está disponible para diferentes sistemas operativos (aunque se recuerda que para otros apartados se requiere de GNU/Linux). El método más simple para obtener Docker es:

```
$ curl https://get.docker.com | bash -
```

Esto deberá ser así para todas las máquinas del cluster. También se recuerda que para que Docker opere con usuarios no *root*, se deben incluir en el grupo de usuarios 'docker'.

Asimismo, Overnode también debe ser instalado en todas las máquinas, y también ofrece un método de instalación simple:

```
# curl https://overnode.org/install | bash -
```

Nota: Se conoce que Weave (gestor de redes en Overnode) tiene un *bug* en distribuciones basadas en Ubuntu. Se puede remediar comentando o eliminando la opción 'trust-ad' en el fichero `/etc/resolv.conf`.

Continuación con la instalación de Overnode, se han de añadir las máquinas al cluster. Aquí no existe un concepto de máquina principal o maestra, solo un grupo de máquinas indexadas. Al añadir la máquina se le puede especificar qué otras debe esperar que se conecten al clúster mediante su nombre DNS (éste ha de ser configurado previamente). Es importante recordar el índice de cada máquina, ya que este valor se empleará en los despliegues y además le permite conocer bajo qué nombre se identifica en el cluster (mediante la lista de máquinas que se ha provisto); y también establecer un *token* arbitrario común como medida de autenticación:

```
host1# overnode launch --id 1 --token token-arb host1 host2 host3
host2# overnode launch --id 2 --token token-arb host1 host2 host3
host3# overnode launch --id 3 --token token-arb host1 host2 host3
```

La interacción con el cluster se puede realizar ahora desde cualquiera de los nodos. Para comprobar su estado actual:

```
# overnode status
```

Anexo G. Operativa de máquinas del laboratorio

Se instaló una distribución liviana de GNU/Linux conocida como *Alpine*[22] en dichas máquinas bajo una red privada. El acceso a esta red se disponía a través de otra máquina del laboratorio (*dawson)*, ésta funcionaba como *gateway* a internet.

Con el motivo del plan de ahorro energético se ha configurado una unidad de distribución de alimentación (*Power Distribution Unit* o PDU) inteligente, administrada bajo la misma red. Se descartó la configuración que presentaba el PDU en primera instancia, por lo que se hizo un reseteo de fábrica y se partió de cero con el manual original. Para la conexión de *setup* inicial, se estableció una comunicación por puerto serie, aplicando la tasa en Baudios y opciones indicadas por el manual. Una vez logrado el acceso, se ajustó en su configuración de red que tomara IP mediante DHCP. La máquina *dawson* funcionaba también como servidor DHCP, y se congeló una IP para la dirección MAC de la PDU. De este modo se simplifica la operativa del PDU con IP estática, ahora prescindiendo de necesidad de conexión por serie, a favor de la comunicación por red *telnet*.

Las máquinas se han configurado desde la BIOS para ser iniciadas automáticamente en el momento que reciben alimentación; es decir, se elimina la necesidad de un encendido en físico.

Esto cumplimenta con el plan de ahorro energético porque permite en remoto hacer un apagado de las máquinas, que no solo implica menos consumo eléctrico de las máquinas, sino también en acondicionamiento del aire de la sala en la que ya operan varios nodos.

Anexo H. Paneles de Grafana

Aquí se muestran algunas imágenes con paneles de Grafana, informativos para el contexto del sistema de monitorización que se ha desarrollado.

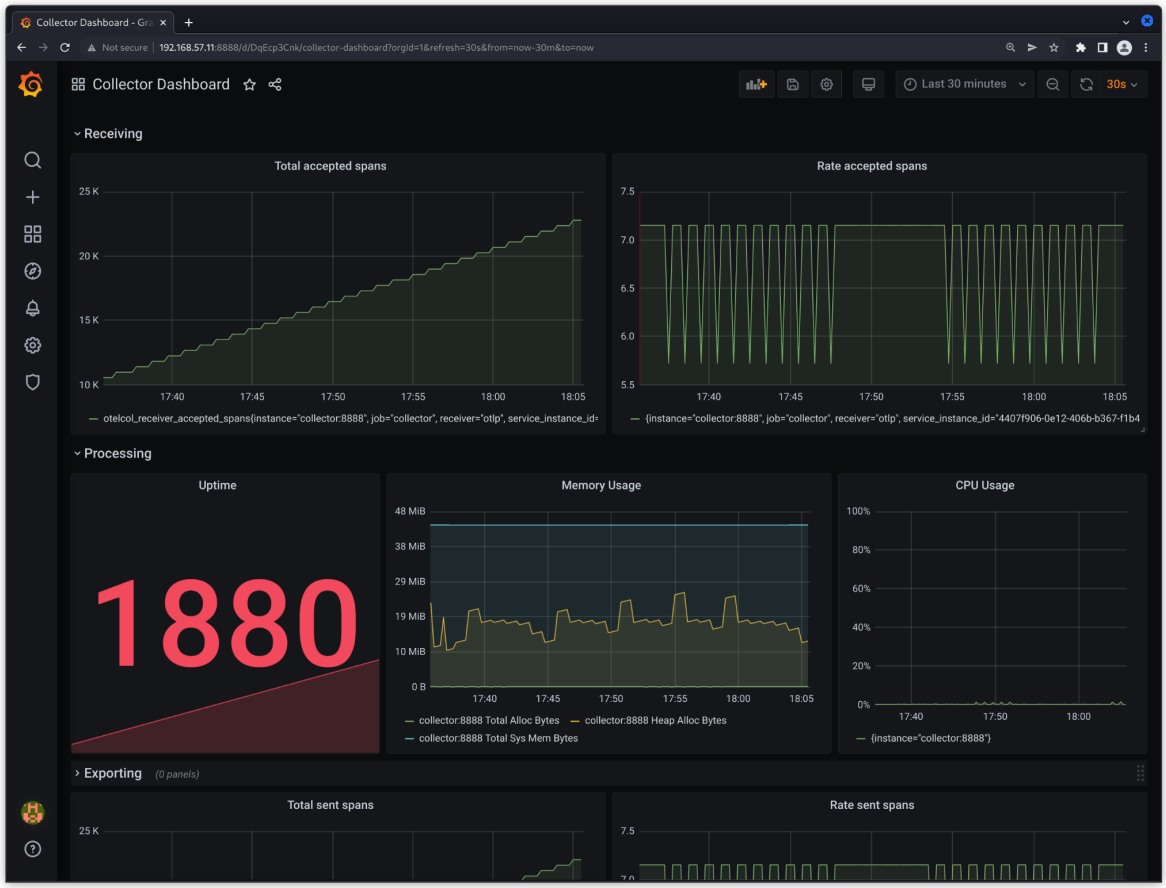

Imagen 4. Panel de métricas para los colectores.

El panel muestra el número de *spans* que se ha recibido en cada colector, métricas sobre el uso de memoria y CPU en el procesamiento, y finalmente, el número de *spans* emitidos.

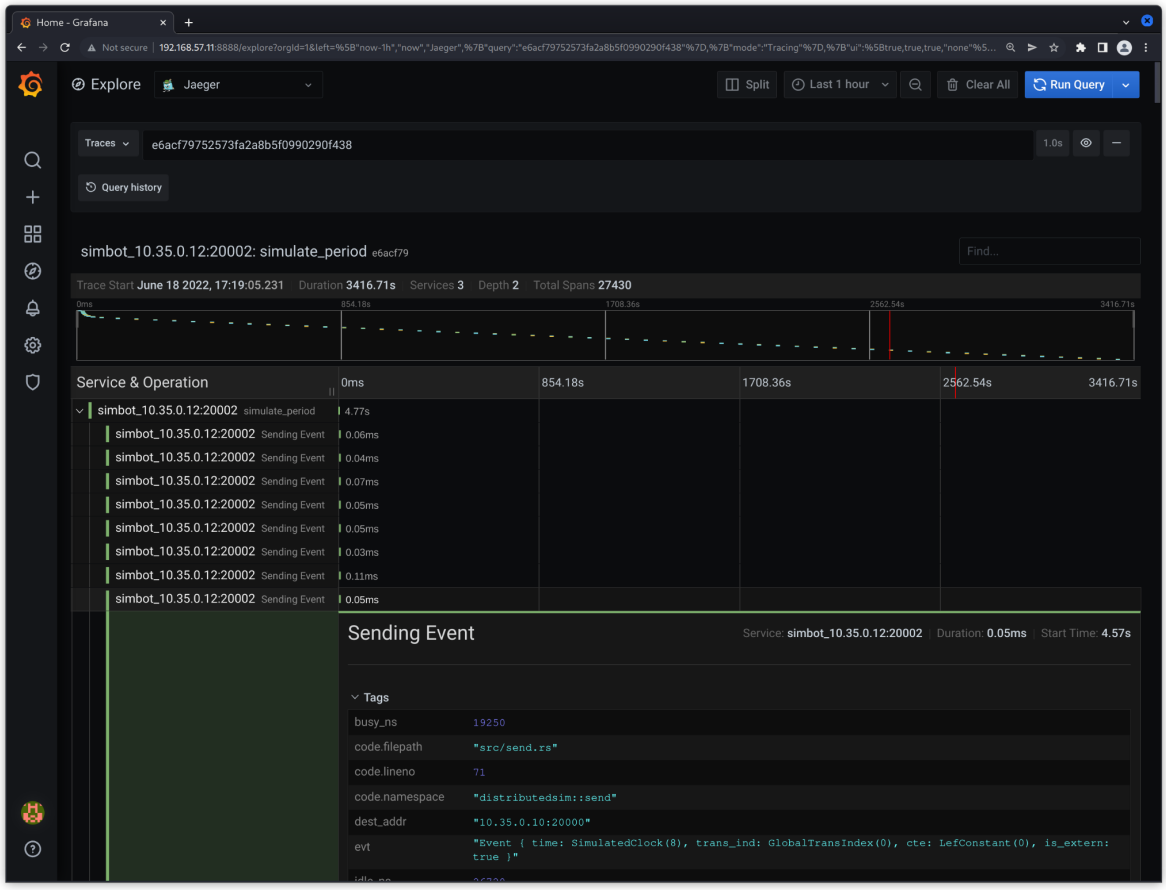

Imagen 5. Panel de trazas de Grafana.

El panel de trazas es claramente una ramificación del cliente web de Jaeger, con la única diferencia a destacar en que éste se encuentra en un tema oscuro que se integra con el resto del cliente web de Grafana.

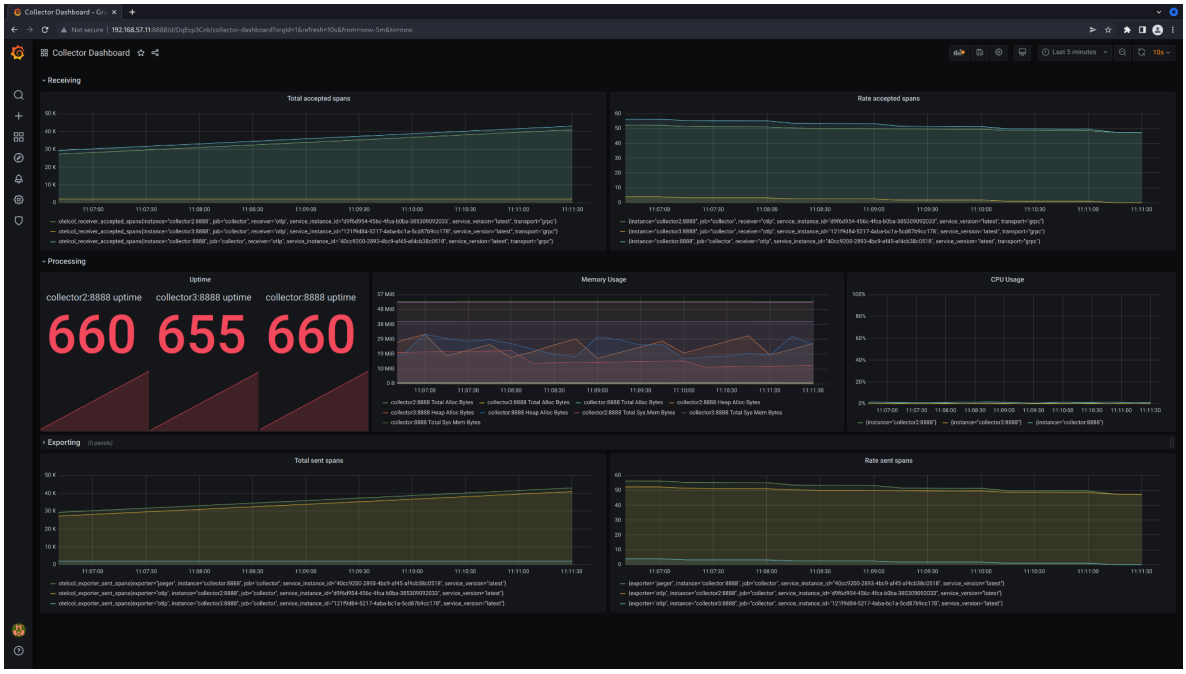

Imagen 6. Panel de métricas de colectores con 3 instancias reportando.

Con tres colectores reportando métricas se puede entender el reparto de carga que tienen estos. En este caso hay un colector hoja que recibe mucha más carga que el otro colector hoja; un claro indicio se que se ha de reajustar la jerarquía de los colectores.

Anexo I. Depuración del simulador con Jaeger

A continuación se muestran imágenes con las trazas de una ejecución de 6 nodos *simbot*, en un primer caso con un fallo de simulación, y en un segundo caso ya corregido. Se ha hecho uso del *vtimelineprocessor*, con lo que se muestran tiempos de simulación, no tiempos reales.

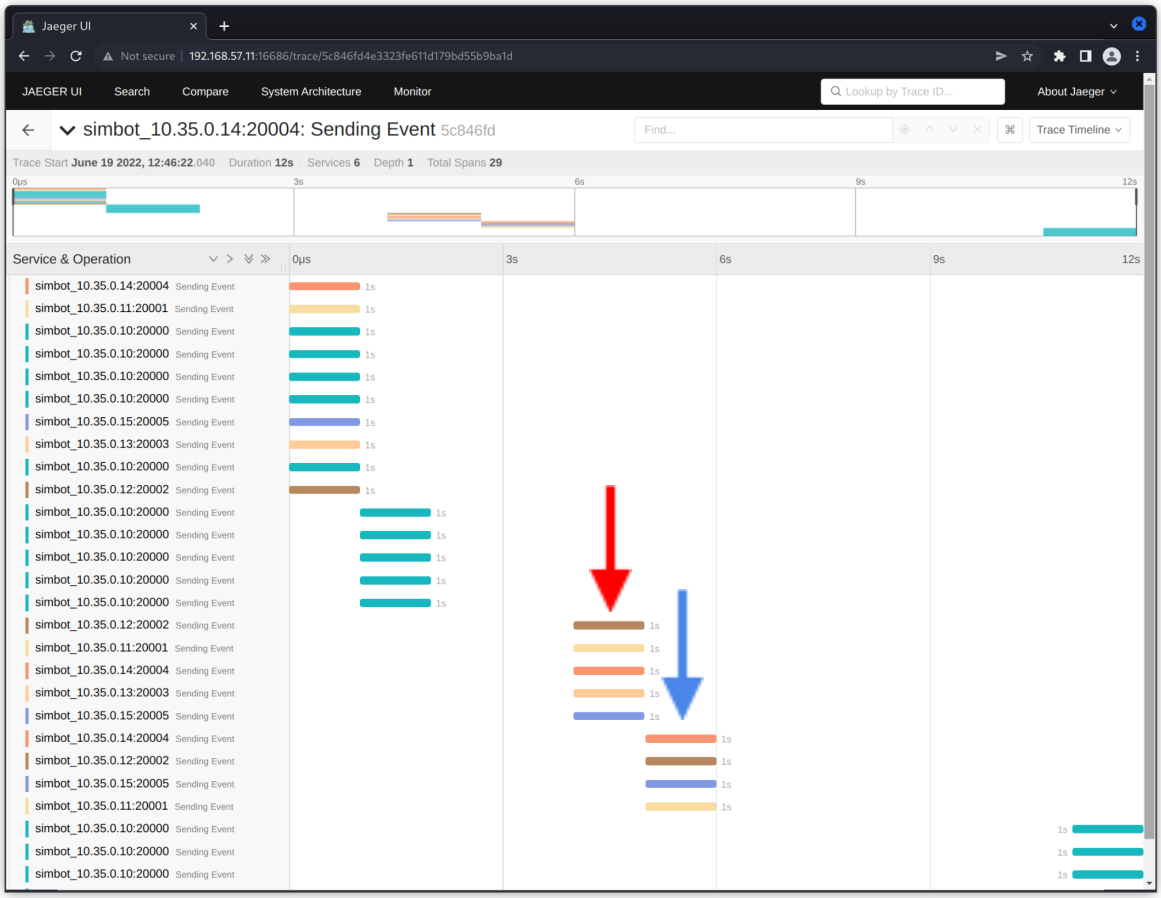

Imagen 7. Trazas de una simulación errónea.

Como se puede apreciar en la Imagen 7, a las trazas de cada nodo les corresponde un color: **simbot0**, **simbot1**, **simbot2**, **simbot3**, **simbot4** y **simbot5**. Se conoce que la naturaleza de esta Red de Petri es la de transiciones prácticamente idénticas entre los *simbots* 1 al 5, convergiendo en el *simbot* 0. Sin embargo, en el tiempo de simulación 4 existen trazas para los cinco nodos similares (flecha roja), mientras que en tiempo 5 sólo se pueden ver cuatro (flecha azul). Esto demuestra que el simbot 3 se haya quedado una unidad de tiempo de simulación atrasado y no pueda avanzar, indicativo del causante del fallo de la simulación.

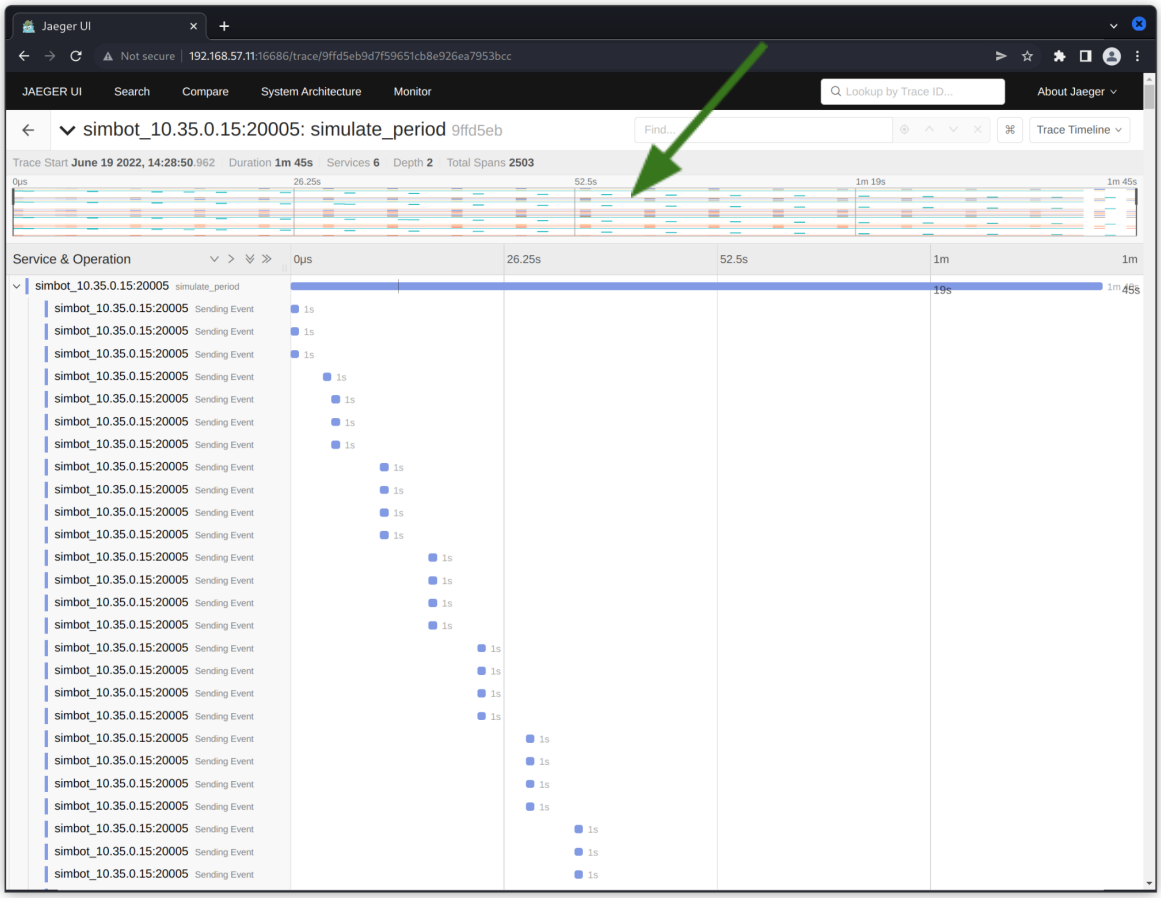

Imagen 8. Trazas de una simulación sin errores.

Ahora con el error solucionado, se puede contemplar que se han recibido las trazas con el completo tiempo de vida de los nodos *simbot*. En la Imagen 8, se muestra con claridad el tiempo de vida total del *simbot* 5 (1 minuto y 40 segundos, que corresponde a 100 segundos, que representan 100 unidades de tiempo de simulador), con alguno de sus eventos de envío. En paralelo, se muestran los tiempos de vida de todos los nodos en el minimapa (flecha verde).

Anexo J. Código de *adoptprocessor*

A continuación se presenta el código con la lógica característica de este procesador para colectores de OpenTelemetry. Se deja fuera todo código adicional que sea común en idea o forma a la de otros procesadores.

Estructura de datos del procesador:

```
type adoptProcessor struct {
    config Config
    parent ptrace.Span // Parent Adopter
}
```

Inicialización del procesador:

```
// newAdoptProcessor returns the adopt processor.
func newAdoptProcessor(config Config) (*adoptProcessor, error) {
    ap := &adoptProcessor{
        config: config,
    }

    span := ptrace.NewSpan()

    ap.parent = span
    return ap, nil
}
```

Función de procesamiento de trazas del procesador:

```
// Main processing function that adopts the incoming traces under the adoptant
// parent trace.
func (ap *adoptProcessor) processTraces(_ context.Context, td ptrace.Traces)
(ptrace.Traces, error) {

    rss := td.ResourceSpans()
    for i := 0; i < rss.Len(); i++ {
        rs := rss.At(i)
        ilss := rs.ScopeSpans()
        for j := 0; j < ilss.Len(); j++ {
            ils := ilss.At(j)
            for k := 0; k < ils.Spans().Len(); k++ {
                span := ils.Spans().At(k)
                _, selfexists := span.Attributes().Get("self")
                _, evtexists := span.Attributes().Get("evt")

                if selfexists || evtexists {
                    if ap.parent.TraceID().IsEmpty() {
                        ap.parent.SetTraceID(span.TraceID())
                    } else {
                        span.SetTraceID(ap.parent.TraceID())
                    }
                    simbotip, exists := span.Attributes().Get("src_addr")
                    if exists {
                        rs.Resource().Attributes().
                            UpsertString(
                               conventions.AttributeServiceName,
                               "simbot_"+simbotip.AsString(),
                            )
                    }
                }
            }
        }
    }
    return td, nil
}
```

Anexo K. Código de *vtimelineprocessor*

Aquí se muestra el código que implementa el procesador para la traducción de tiempo real a tiempo de simulación en las trazas. Al igual que en el Anexo J, sólo se incluyen las cuestiones relevantes al funcionamiento del mismo.

Estructura de datos del procesador, constantes y variables:

```
var (
    numbers  *regexp.Regexp = regexp.MustCompile("[0-9]+")
    vdt_base int64          = time.Now().UTC().UnixNano()
)
const vdt_magn = 1000000000 // Nanoseconds in a Second

type vtimelineProcessor struct {
    config Config
}
```

Inicialización del procesador:

```
// newVtimelineProcessor returns the vtimeline processor.
func newVtimelineProcessor(config Config) (*vtimelineProcessor, error) {
    ap := &vtimelineProcessor{
        config: config,
    }
    return ap, nil
}
```

Función de procesamiento de trazas del procesador:

```
// Main processing function that translates real timestamps from incoming
// traces to their Simulation timestamps.
func (vp *vtimelineProcessor) processTraces(_ context.Context, td
ptrace.Traces) (ptrace.Traces, error) {
    rss := td.ResourceSpans()
    re := regexp.MustCompile("SimulatedClock.[0-9]*.")

    for i := 0; i < rss.Len(); i++ {
        rs := rss.At(i)
        ilss := rs.ScopeSpans()
        for j := 0; j < ilss.Len(); j++ {
            ils := ilss.At(j)
            for k := 0; k < ils.Spans().Len(); k++ {

                span := ils.Spans().At(k)
                selfctx, exists := span.Attributes().Get("self")

                if exists {
                    realts := span.StartTimestamp().String()
                    span.Attributes().InsertString(
                        "real_start_timestamp", realts,
                    )
                    simclock := re.FindAllString(selfctx.AsString(), -1)[0]
                    ts, err := strconv.ParseInt(
                        numbers.FindString(simclock), 10, 0,
                    )
                    if err != nil {
                        ts = 0
                    }
                    span.SetStartTimestamp(
                        pcommon.Timestamp(ts*vdt_magn + vdt_base),
                    )

                    fc, found := span.Attributes().Get("final_cycle")
                    var fcs string
                    if !found {
                        fcs = "2"
                    } else {
                        fcs = fc.AsString()
                    }

                    realte := span.EndTimestamp().String()
                    span.Attributes().InsertString(
                        "real_end_timestamp", realte,
                    )
                    te, err := strconv.ParseInt(
```

```
                        numbers.FindString(fcs), 10, 0,
                    )
                    if err != nil {
                        te = 0
                    }
                    span.SetEndTimestamp(
                        pcommon.Timestamp(te*vdt_magn + vdt_base),
                    )
                }

                selfctx, exists = span.Attributes().Get("evt")

                if exists {
                    realts := span.StartTimestamp().String()
                    span.Attributes().InsertString(
                        "real_start_timestamp", realts,
                    )
                    realte := span.EndTimestamp().String()
                    span.Attributes().InsertString(
                        "real_end_timestamp", realte,
                    )

                    re := regexp.MustCompile("SimulatedClock.[0-9]*.")
                    simclock := re.FindAllString(selfctx.AsString(), -1)[0]
                    ts, err := strconv.ParseInt(
                        numbers.FindString(simclock), 10, 0,
                    )
                    if err != nil {
                        ts = 0
                    }
                    span.SetStartTimestamp(
                        pcommon.Timestamp(ts*vdt_magn + vdt_base),
                    )
                    span.SetEndTimestamp(
                        pcommon.Timestamp((ts+1)*vdt_magn + vdt_base),
                    )
                }
            }
        }
    }
    return td, nil
}
```