UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Framework para simplificar la construcción del modelado y simulación de cadenas de suministros

AUTORES: Matias Pankow y Esteban Sanchez

DIRECTOR: Dr. Alejandro Fernández

CODIRECTOR: Dr. Jorge Hernández (Universidad de Liverpool)

ASESOR PROFESIONAL:

CARRERA: Licenciatura en informática

Resumen

En esta tesina se construyó un framework que facilita la construcción y simulación de modelos de cadenas de suministros. Gracias a este framework, se obtienen mejoras frente a estrategias existentes, en términos de flexibilidad, implementación, ejecución, despliegue e interacción entre los modelos generados. Además, cuenta con una herramienta gráfica que permite la creación, simulación y generación de reportes de una cadena de suministros. Obteniendo estas características a partir de la utilización de diversos estándares y nuevas tecnologías, aplicadas sobre un ejemplo concreto del mundo real.

Palabras Clave

Framework. Modelo. Simulación. Cadena de suministros. FIPA. Serverless.

Conclusiones

Gracias al desarrollo de este framework se logró simplificar el trabajo de creación de modelos y simulaciónes de cadenas de suministros, haciendo posible de esta forma que las simulaciones desarrolladas a partir del mismo requieran menor costo y alcancen una mejor calidad, agregando una capa de abstracción sobre temas primordiales como el almacenamiento, la comunicación entre agentes y el control sobre las simulaciones realizadas, logrando importantes mejoras sobre estrategias existentes, aportando una herramienta gráfica para la gestión de los modelos y demostrando la utilidad de todo el trabajo realizado a través de un ejemplo real.

Trabajos Realizados

Estudio del estado del arte.
Definición de casos de uso.
Análisis de herramientas de desarrollo y ejecución.
Desarrollo de un framework para la generación de sistemas de modelado y simulación de cadenas de suministros.
Instalación y configuración del software implementado.
Evaluación de los aportes en un caso concreto.

Trabajos Futuros

Extender el framework para que realice un manejo de usuarios, permitiendo el registro de usuarios nuevos en el sistema pudiendo establecer los accesos que posee. Desarrollar una herramienta de depuración para facilitar la detección de errores en la nube. Aprovechar las métricas que se pueden obtener utilizando recursos de la nube. Ampliar el frontend del sistema, agregando geolocalización, visualización de mapas 3D, ejecutar multiples simulaciones en paralelo, entre otros. Utilizar el framework para representar ejemplos reales más completos y complejos.

Fecha de la presentación: Julio 2019

Índice general

Índice de ilustraciones

Índice de tablas

Capítulo 1

Introducción

Cadena de suministros

El término “Gestión de Cadena de Suministros” también conocido como “Cadena de Abasto” ,del inglés Supply Chain Management (SCM), entró al dominio público cuando Keith Oliver, un consultor en Booz Allen Hamilton (firma global en consultoría, análisis, soluciones digitales, ingeniería y ciberespacio[1]), lo usó en una entrevista para el Financial Times (empresa británica de noticias[2]) en 1982. Tomó tiempo para afianzarse y quedarse en el léxico de negocios, pero a mediados de los 1990’s empezaron a aparecer una gran cantidad de publicaciones sobre el tema y se convirtió en un término regular en los nombres de los puestos de algunos funcionarios [Jacoby - 2009[3]].

Una cadena de suministros es un conjunto de empresas que pasan materiales. Normalmente, varias empresas independientes participan en la fabricación y la colocación de un producto en manos del usuario final. Productores de materias primas y componentes, ensambladores, mayoristas, comerciantes minoristas y compañías de transporte son todos miembros de una cadena de suministros [La Londe y Masters - 1994[4]].

Otra definición señala que una cadena de suministros es la red de organizaciones involucradas, a través de enlaces ascendentes y descendentes, en los diferentes procesos y actividades que producen valor en forma de productos y servicios entregados al consumidor final [Christopher - 1992[5]].

[1] Booz Allen Hamilton - https://www.boozallen.com/about.html

[2] Financial Times - https://aboutus.ft.com/en-gb/

[3] David Jacoby, “The Economist Guide To Supply Chain Management”. Economist Books, 2009

[4] La Londe, Bernard J. and James M. Masters. “Emerging Logistics Strategies: Blueprints for the Next Century”. International Journal of Physical Distribution and Logistics Management, 1994, Vol. 24, No. 7, pp. 35-47.

[5] Christopher, Martin L. “Logistics and Supply Chain Management”. London: Pitman Publishing, 1992.

Ilustración 1. Imagen de los componentes que integran la gestión de cadenas de suministros

En otras palabras, una cadena de suministros consta de múltiples empresas, tanto en sentido ascendente (es decir, suministros) como en sentido descendente (es decir, distribución). Hay que tener en cuenta que estos conceptos incluyen al consumidor final como parte de la cadena de suministros.

En el contexto de este trabajo, nos interesa entender que una cadena de suministros está compuesta por "nodos" o “agentes”, que pueden actuar como proveedores y/o consumidores de productos y/o servicios. La relación principal entre estos nodos (desde nuestra perspectiva) está dada por el proceso de negociación, es decir, por los pedidos y ofertas que ocurren entre ellos y las condiciones bajo las cuales son aceptados o rechazados. Un factor que esa relación intenta mejorar es el beneficio (por ejemplo económico) que los nodos obtienen.

Modelado y simulación de una cadena de suministros

Los modelos, en general, apuntan a reducir el riesgo, ayudando a comprender mejor un problema complejo y sus posibles soluciones antes de emprender el gasto y esfuerzo de una implementación completa [Selic - 2003[6]].

Estos modelos se construyen como una simplificación de la realidad, los cuales permiten comprender sistemas complejos en su totalidad. Dado que se reduce el problema que se está estudiando, centrándose en un solo aspecto a la vez.

El término “simulación” viene del latín, de la unión de “similis” (parecido) y el sufijo “-ion” (acción y efecto). Puede definirse a la simulación como la experimentación con un modelo que imita ciertos aspectos de la realidad. Esto permite trabajar en condiciones similares a las reales, pero con variables controladas y en un entorno que se asemeja al real pero que está creado o acondicionado artificialmente.

[6] Selic, B. “The pragmatics of model-driven development”. IEEE Softw. 20(5), September 2003, 19–25.

Gracias a las simulaciones es posible corregir fallos antes de que estos se concreten en la realidad, ya que el objetivo de una simulación es comprobar el comportamiento de una persona, de un objeto o de un sistema en contexto que, si bien no son idénticos a lo real, ofrecen el mayor parecido posible [Pérez Porto y Merino - 2011[7]].

Al simular modelos de cadenas de suministros, las organizaciones pueden analizar y experimentar el proceso de cadena de suministros, en un entorno virtual libre de riesgos, reduciendo el tiempo y los costos de las pruebas físicas, de esta forma, se pueden evaluar posibles escenarios de negociaciones, para intentar alcanzar el más favorable para la cadena de suministros.

En esta tesina es de interés modelar el aspecto de pedido, oferta y acuerdo entre los distintos nodos de la cadena de suministros y poder observar qué es lo que sucede entre ellos en las distintas iteraciones, viendo los distintos escenarios que se van generando, en busca de aquél que sea más beneficioso para todos los integrantes de dicha cadena de negociaciones.

El modelado y simulación de este proceso de pedido, oferta y acuerdo facilitará la tarea de buscar la mejor combinación de negocios para todos los nodos de la cadena, ya que se realizará en un entorno controlado y monitoreado de pruebas, donde los nodos tendrán la posibilidad de configurarse para utilizar mecanismos de toma de decisiones diversos.

Antecedentes

Como antecedente a esta tesina de grado, se cuenta con la tesis doctoral "*Propuesta de una arquitectura para el soporte de la planificación de la producción colaborativa en cadenas de suministros de tipo árbol*" desarrollada por el Dr. Jorge Hernández Hormazábal.

Dicha tesis trata sobre el análisis del proceso de planificación de cadenas de suministros, cómo puede abordarse este proceso utilizando sistemas multiagentes y finalmente el desarrollo de un sistema informático de modelado y simulación de cadenas de suministros.

El sistema desarrollado y presentado por el Dr. Hernández se trata de un sistema multiagente que utiliza tecnologías de la época que actualmente pueden ser reemplazadas, con el fin de mejorar en performance, acceso distribuido, almacenamiento y mantenibilidad.

Dentro de estas tecnologías se encuentran, por ejemplo, el uso de *JADE*, un framework JAVA para el desarrollo de sistemas multiagentes. Este framework brinda la posibilidad de crear simulaciones de sistemas distribuidos, en los que los agentes ejecutan comportamiento distribuido, pero con la restricción de estar dentro de un único contenedor. Además, es necesario contar con un servidor configurado para el soporte de esta tecnología, lo que implica la puesta a punto y trabajo de mantenimiento futuro para asegurar el buen funcionamiento del sistema multiagente [JADE[8]].

[7] Pérez Porto J. y Merino M. (2011). Definición de simulación. Retrieved from https://definicion.de/simulacion

[8] Sitio web oficial JADE - http://jade.tilab.com/

Otra de las tecnologías utilizadas es *MySQL*[9], lenguaje y motor de base de datos, para el pasaje de información entre los distintos agentes de la cadena. Esta implementación conlleva un déficit en performance y almacenamiento, creando un retardo significativo en el proceso de comunicación entre agentes y un aumento en el uso de espacio de almacenamiento, desventajas derivadas de la estrategia de comunicación de procesos por medio de un espacio de memoria compartido.

[9] Sitio web oficial MySQL - https://www.mysql.com/

Capítulo 2

Comparación con el estado del arte

Esta tesina sienta sus bases sobre la tesis antes mencionada, del Dr Hernandez Hormazabal, pero no es el único trabajo realizado sobre el tema. A lo largo de los años, la gestión de cadenas de suministros ha sido ampliamente estudiada y desarrollada con el fin de minimizar riesgos, optimizar procesos, predecir resultados, entre otros, logrando así, contar con múltiples opciones en el mercado de sistemas de modelado y simulación de cadenas de suministros.

Estos sistemas, por nombrar algunos, AnyLogic[10], FlexSim[11], Simio[12], entre otros, proveen una gran variedad de opciones a la hora de modelar y simular cadenas de suministros, contando con vistosos entornos de trabajo, como mapas satelitales o 3D, modelado 3D de líneas de ensamblaje y distribución, trazado de rutas entre los diferentes nodos de la cadena de suministros, inteligencia artificial, realidad virtual, etc.

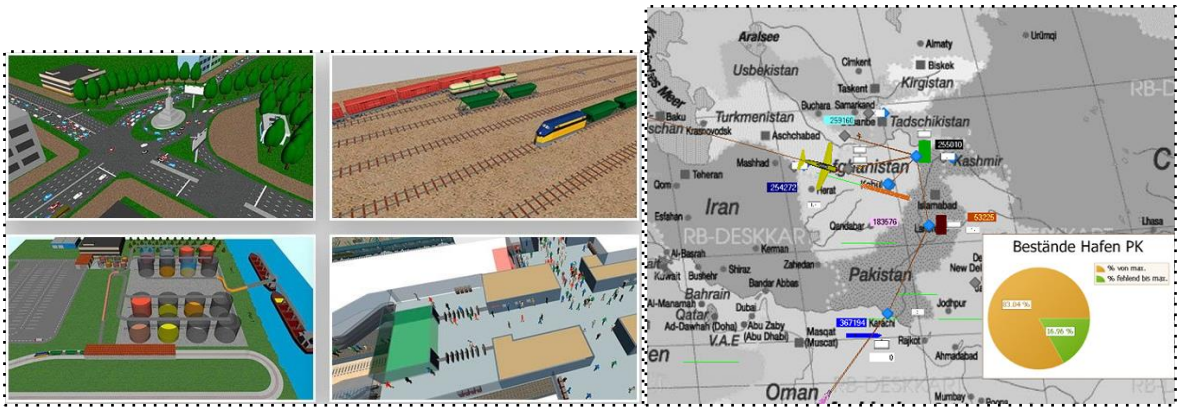

Ilustración 2. Simulaciones 3D de AnyLogic y modelado de cadena de suministros con Simio.

Ilustración 3. Simulaciones FlexSim con realidad virtual.

Es por esto que el objetivo principal de esta tesina no se basa en el desarrollo de un sistema más, de modelado y simulación de cadenas de suministros, sino que se centra en una beta no explotada del problema en cuestión. Esta beta se refiere al desarrollo de un

[10] Sitio oficial AnyLogic - https://www.anylogic.com/

[11] Sitio oficial FlexSim - https://www.flexsim.com/

[12] Sitio oficial Simio - https://www.simio.com/index.php

framework (Supply Chain Simulator o SCS) para la creación de sistemas de modelado y simulación de cadenas de suministros. En otras palabras, este trabajo no tiene el fin de brindar un sistema para modelar y simular, sino que se focaliza en el desarrollo de un sistema que brinda la posibilidad al usuario de crear su propia aplicación de modelado y simulación de cadenas de suministros, pudiendo así dotarla de la lógica de decisiones y características que más se ajusten a su negocio, ahorrándole el trabajo de tener que investigar y programar los pilares básicos de un sistema de esta índole, tales como la creación de los nodos de la cadena, los mecanismos de comunicación entre dichos nodos, la persistencia de la información lograda a partir del modelado y simulación, etc y además, cuenta con el agregado opcional de una herramienta gráfica, capaz de trabajar en conjunto con el sistema desarrollado por el usuario, la cual también es de código abierto, por ende, es enteramente modificable por él para ajustarse a las necesidades que este disponga, característica no presente en los actuales sistemas de modelado y simulación de cadenas de suministros, los cuales son de código cerrado o privado, donde el usuario final del producto no tiene ningún acceso para realizar cualquier tipo de modificación deseada, teniendo así que ajustarse a las posibilidades limitadas que le brinda cada sistema comercial.

Además, gracias a SCS, el usuario será capaz de desarrollar, de manera simple y open *source*, sistemas distribuidos, capaces de interactuar con otros sistemas externos de manera estandarizada, sin importar cómo estén desarrollados dichos sistemas y obteniendo resultados con gran performance, amigable para el usuario y con grandes características en escalabilidad y mantenibilidad, ventajas que posee sobre sus competencias actuales en el mercado, ya que los mismos son sistemas que corren sobre un servidor centralizado, es decir, no cuentan con las ventajas de un sistema distribuido, no poseen interacción con sistemas externos, más allá de simples conexiones con bases de datos externas, lo que los lleva a ser sistemas más limitados en cuanto a accesos e interacciones externas.

Capítulo 3

Objetivos generales

Los objetivos generales de esta tesina son proponer una solución que simplifique la construcción, el despliegue y la ejecución de simulaciones de cadenas de suministros, donde los nodos de la cadena se pueden encontrar distribuidos. Soportará también el monitoreo y reporte de las negociaciones entre los nodos y la posibilidad de integración con sistemas externos.

Para diseñar la solución se armara un framework. La palabra framework hace referencia a un sistema que se puede personalizar, especializar o ampliar para proporcionar capacidades más específicas, más apropiadas o ligeramente diferentes. Este tipo de sistemas respaldan la reutilización de arquitecturas e implementaciones de software probadas, y su uso reduce el costo del software y mejora su calidad.

Este framework estará compuesto por frozen spots y hot spots. Se llama frozen spots a aquellas piezas de software codificadas que se reutilizan, es decir a los procesos del sistema que tienen el mismo comportamiento sin importar el nodo que lo esté ejecutando, y hot spots a los elementos flexibles que permiten al usuario ajustar el framework a las necesidades de la aplicación concreta [Parsons, Rashid, Telea y Speck - 2006[13]].

Además, los frameworks pueden clasificarse en frameworks de *"caja blanca"* o frameworks de *"caja negra"*. Que un framework sea de *"caja blanca"* indica que el usuario del framework deberá conocer de antemano cómo está diseñado el framework y cómo es su funcionamiento interno, en cambio, los de *"caja negra"* indican que este conocimiento anterior no es necesario, el usuario del framework lo puede utilizar sin ningún inconveniente, sin la necesidad de saber el funcionamiento y diseño interno del mismo, sino que solamente conoce cuáles son los hot spots que debe implementar y una descripción general de lo que hace el mismo.

[13] Parsons D., Rashid A., Telea A.,Speck A. "An architectural pattern for designing component- based application frameworks". Published online 22 November 2005 in Wiley InterScience.

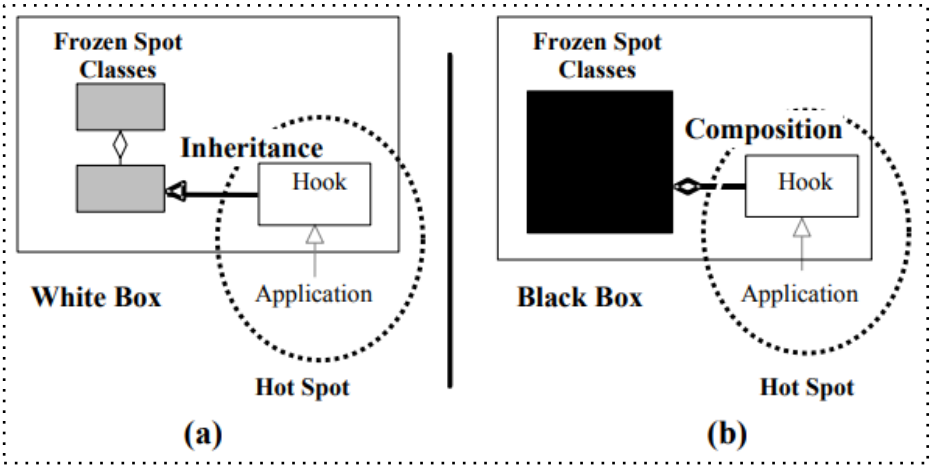

Ilustración 4. Clasificación de los frameworks, (a) de caja blanca y (b) de caja negra.

El proyecto de esta tesina es un framework de "*caja negra"*, debido a que los usuarios del mismo no necesitan saber cómo es el funcionamiento interno de, por ejemplo, el medio de comunicación de los agentes o cómo el modelo de la cadena de suministros es persistido para no perder toda su información, sino que lo pueden utilizar tan solo implementando los métodos necesarios, como por ejemplo el método de toma de decisiones de cada agente.

Objetivos específicos

Los objetivos específicos de esta tesina son, partir del antecedente antes mencionado, modernizarlo, utilizando nuevas tecnologías y protocolos estándares, para así lograr:

- Obtener un framework que simplifique el armado de modelos de cadenas de suministros.
- Ofrecer mejoras, en relación a estrategias ya existentes, en términos de flexibilidad implementación, ejecución, despliegue e interacción de los modelos.
- Ofrecer herramientas de monitoreo de ejecución de los modelos.
- Evaluar los aportes en un caso concreto.

Capítulo 4

Enfoque general

Para lograr los objetivos anteriormente listados se va a armar un framework, Supply Chain Simulator (SCS), donde los hot spots serán personalizables para cada nodo de la cadena de suministros, definiendo así, el mecanismo de toma de decisiones de cada uno, y los frozen spots contendrán la lógica del armado de cadenas de suministros, es decir, la creación de los nodos, el medio de comunicación entre los mismos y la forma de almacenar y obtener información de una base de datos distribuida.

Este framework será desarrollado dentro del marco de nuevas tecnologías, tales como el uso de serverless, base de datos *NoSQL*, y estándares como el protocolo de comunicación *FIPA Communicative Act Library* (CAL) y el protocolo de comunicación *HTTP*.

El uso de tecnologías serverless consiste en utilizar un servidor en la nube, completamente manejado por un proveedor de servicios, eliminando la necesidad de contar con un servidor físico para el sistema, con las tareas de configuración y mantenimiento que esto conlleva, permitiendo así la posibilidad de centrar la atención en el desarrollo del sistema, dejando de lado las preocupaciones sobre los recursos necesarios para la infraestructura del mismo y utilizando de forma más eficiente los recursos brindados por los proveedores en la nube [Savage - 2018[14]].

Durante décadas, predominó el uso de bases de datos relacionales para el desarrollo de aplicaciones, tales como *Oracle*, *SQL Server*, *MySQL*, entre otras. Pero esto comenzó a cambiar a partir de la década del 2000 con el aumento en la utilización de modelos de datos no relacional, es decir "*NoSQL*" [Amazon[15]]. Este tipo de bases de datos cuentan con características tales como, flexibilidad, escalabilidad, alto rendimiento y alta funcionalidad.

FIPA (Foundation For Intelligent Physical Agents) es una organización de estándares de la *Sociedad de Computación IEEE* que promueve la tecnología basada en agentes y la interoperabilidad de sus estándares con otras tecnologías. El estándar de comunicación *CAL* es un protocolo que define todos los aspectos presentes en un acto de comunicación genérico, tales como los mensajes que van de un emisor a un receptor y la forma correcta de cómo contestar a dichos mensajes. Este estándar es aplicable a cualquier tipo de comunicación, ya sea verbal, no verbal, digital, etc. [Foundation For Intelligent Physical Agents - 2002[16]].

HTTP, Hypertext Transfer Protocol o Protocolo de transferencia de hipertexto es un protocolo de comunicación de nivel de aplicación para sistemas de información distribuidos, colaborativos. Es un protocolo genérico, sin estado, es decir, no guarda ninguna información sobre conexiones anteriores, se puede utilizar para muchas tareas, más allá de su uso para

[14] Neil Savage (2018), "Going serverless".

[15] Amazon - NoSQL - https://aws.amazon.com/es/nosql/

[16] Foundation For Intelligent Physical Agents (2002), "FIPA Communicative Act Library Specification".

hipertexto, como servidores de nombres y sistemas de administración de objetos distribuidos, mediante la extensión de sus métodos de solicitud, códigos de error y encabezados. Desde 1990 es utilizado por la iniciativa de información global *World-Wide Web* (WWW) [World Wide Web Consortium & Internet Engineering Task Force - 1999[17]].

HTTP también se utiliza como un protocolo genérico para la comunicación entre agentes de usuario y proxies / puertas de enlace a otros sistemas de Internet, incluidos los compatibles con protocolos *SMTP*, *NNTP*, *FTP*, *Gopher* y *WAIS*. De esta forma, *HTTP* permite el acceso básico a recursos disponibles desde diversas aplicaciones.

SCS cuenta con la virtud de ejecutar el modelado y simulación de cadenas de suministros en 2 posibles ambientes. Uno de ellos es ejecutar el framework en *Amazon Web Service* (AWS), mientras que la otra posibilidad es ejecutarlo en un ambiente local.

AWS es la opción brindada por Amazon para computación en la nube. Gracias a la computación en la nube, es posible reemplazar los gastos de infraestructura de capital por adelantado, por costos variables bajos que escalan con el negocio. En otras palabras, es posible obtener instantáneamente cientos o miles de servidores en minutos y entregar resultados más rápido, sin la necesidad de planificar ni adquirirlos con semanas o meses de antelación [Amazon[18]].

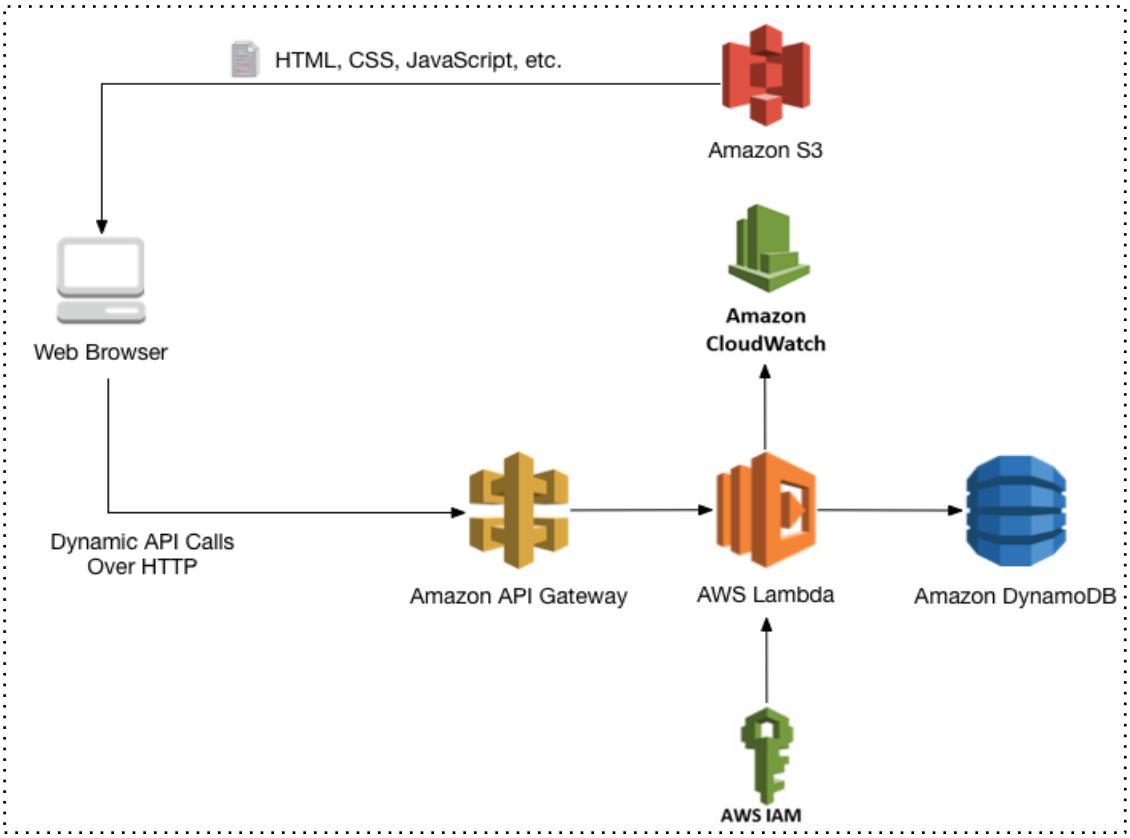

Ilustración 5. Los servicios ofrecidos por AWS utilizados en esta tesina son: Lambda, CloudWatch, IAM, S3, API Gateway,y DynamoDB

[17] World Wide Web Consortium & Internet Engineering Task Force (1999), RFC 2616.

[18] Amazon Web Services - https://d1.awsstatic.com/whitepapers/aws-overview.pdf

El ambiente local emula al entorno de AWS en un servidor local. Presentando las mismas posibilidades de trabajo, pero en el marco de un hardware limitado. Es decir, que se obtendrán los mismo resultados finales, pero con distinta performance de ejecución.

Capítulo 5

Arquitectura

El framework de esta tesina desarrolla sistemas con una arquitectura de nodos descentralizados. Dichos sistemas pueden ser ejecutados en 2 posibles ambientes, uno en la nube y otro en forma local. Ambos ambientes de ejecución respetan el concepto de *serverless*, *REST* y utilizan protocolos como *HTTP*, *FIPA*, entre otros.

Serverless

Serverless permite el desarrollo de aplicaciones y servicios sin la necesidad de preocuparse por los servidores. Tareas de infraestructura tales como mantenimiento del sistema operativo, aprovisionamiento de servidores o clusters, entre otras, no son necesarias al utilizar arquitecturas serverless. De esta forma, la atención se centra principalmente en el desarrollo de aplicaciones o servicios, con mayor agilidad y a menor costo [Amazon[19]].

Las 4 propiedades esenciales de serverless son:

1. **Administración cero:** El código es implementado sin aprovisionar ningún tipo de recurso antes o después, ya que no se utilizan conceptos como flota, instancias o sistemas operativos.
2. **Escalado automático:** el escalado necesario, producto del aumento o disminución del tráfico, es manejado por el proveedor del servicio serverless.
3. **Pagar para usar:** Con un proveedor de hosting tradicional, se debe pagar dependiendo del aprovisionamiento de recursos previos solicitados, en cambio, con serverless el pago se hace en base al uso dado sobre estos recursos. En otras palabras, si los servicios se encuentran inactivos, entonces no se pagará nada por ellos.
4. **Velocidad incrementada:** Es más rápido el poner en producción una idea, debido a que no es necesario aprovisionar ningún tipo de recurso previamente, ni administrarlo posteriormente a la implementación.

Además, *Serverless* también es un framework de código abierto utilizado para desarrollar aplicaciones con arquitectura *serverless* [Serverless[20]].

Este framework cuenta con las siguientes características:

1. **Agnóstico en la nube:** Esto hace referencia a que *Serverless* funciona independientemente del proveedor de servicios en la nube que se utilice,

[19] Amazon - https://aws.amazon.com/serverless/

[20] Serverless - https://serverless.com/learn/overview/

configurándose de forma automática a partir del proveedor y lenguaje de programación que se use.
2. **Componentes:** Cuenta con numerosos componentes de código abierto ya implementados y listos para que los desarrolladores los utilicen. Algunos de ellos pueden utilizarse para administrar tablas en *DynamoDB*, habilitar la autenticación con *Auth0*, entre otros.
3. **Código para su infraestructura:** Esto hace referencia a la posibilidad de automatización con la que cuenta *Serverless Framework*, tales como recrear una aplicación por completo a partir de la ejecución de un comando.
4. **Centrado en la experiencia de desarrollo:** Se centra en el beneficio de serverless de hacer que los desarrolladores sean más productivos.

REST

Transferencia de estado representacional, del inglés Representational State Transfer (REST), es un estilo de arquitectura de software que define un conjunto de restricciones para la creación de servicios web. Estos servicios web, que respetan estas normas son conocidos como servicios web *RESTful* (RESTful Web Services, RWS) [Fielding - 2000[21]].

La definición de "*recurso web*" abarca a todas las cosas o entidades que pueden identificarse, nombrarse, tratarse o manejarse, de cualquier manera en la web. En un RWS, las solicitudes realizadas a una URI obtendrán una respuesta codificada en algún formato. Los formatos más utilizados para estas respuestas son: *HTML*, *XML* o *JSON*. El protocolo más utilizado para estas solicitudes es *HTTP*, el cual cuenta con la siguiente lista de métodos [RFC 7231[22]]:

- **GET:** Transfiere una representación actual del recurso objetivo.
- **HEAD:** Igual que GET, pero solo transfiere la línea de estado y la sección del encabezado.
- **POST:** Realiza el procesamiento específico de recursos en la carga útil de la solicitud.
- **PUT:** Reemplaza todas las representaciones actuales del recurso destino con la carga útil de la solicitud.
- **DELETE:** Elimina todas las representaciones actuales del recurso destino.
- **CONNECT:** Establece un túnel al servidor identificado por el recurso destino.
- **OPTIONS:** Describe las opciones de comunicación para el recurso objetivo.
- **TRACE:** Realiza una prueba de bucle de retorno de mensaje a lo largo de la ruta al recurso destino.

El conjunto de restricciones definido por REST es el siguiente:

- **Cliente-Servidor:** Respeta el estilo arquitectónico cliente-servidor, este estilo es el más frecuente en aplicaciones basadas en red, y básicamente, está compuesto por

[21] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". University of California, Irvine 2000.
[22] RFC 7231 (2014, June). Retrieved from https://tools.ietf.org/html/rfc7231.

un servidor que ofrece un conjunto de servicios y escucha las solicitudes de dichos servicios. Además del servidor, también se compone de uno o varios clientes, los cuales desean que se realice un servicio, envían la solicitud al servidor a través de un conector, el servidor rechaza o realiza la solicitud y envía una respuesta al cliente. De esta manera se logra separar lo que es la interfaz de usuario (del lado del cliente) de la capa lógica (presente en el servidor). Gracias a esta separación es posible lograr una mejora en la portabilidad de la interfaz de usuario en múltiples plataformas y una mejora en la escalabilidad al simplificar los componentes del servidor, pero por sobre todas las cosas, permite que los componentes evolucionen independientemente, lo que respalda el requisito a escala de internet de múltiples dominios organizacionales.

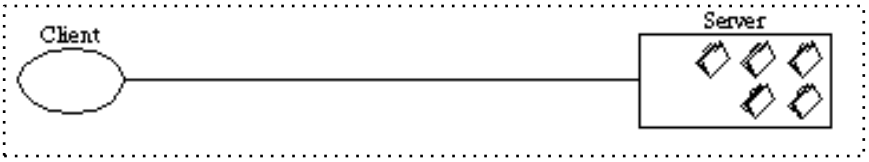

Ilustración 6. Imagen de la arquitectura cliente-servidor con un cliente y un servidor con sus servicios.

- **Sin estado:** La comunicación entre el cliente y el servidor debe ser de naturaleza sin estado. Esto quiere decir que el servidor no almacenará ningún estado de la sesión, por ende, cada solicitud del cliente al servidor debe contener toda la información necesaria para comprender la solicitud y no puede aprovechar ningún contexto almacenado en el servidor. El estado de la sesión se mantiene enteramente en el cliente. Esto hace que la propiedad de visibilidad de la arquitectura mejore, debido a que un sistema de monitoreo no tiene que mirar más allá de un solo dato de solicitud para determinar la naturaleza completa de la solicitud. También, mejora la confiabilidad porque facilita la tarea de recuperación de fallas parciales. Y además, mejora la escalabilidad porque no tener que almacenar el estado entre solicitudes permite que el componente del servidor libere recursos rápidamente y simplifica aún más la implementación. La desventaja que conlleva no almacenar un estado en el servidor, es que puede disminuir el rendimiento de la red debido al aumento de datos repetitivos (sobrecarga por interacción) enviados en una serie de solicitudes, ya que los mismos no pueden dejarse como contexto compartido en el servidor. Además, tener el estado de la aplicación del lado del cliente conlleva la reducción del control por parte del servidor sobre el comportamiento consistente de la aplicación, ya que la aplicación se vuelve dependiente de la implementación correcta de la semántica en múltiples versiones de clientes.
- **Caché:** Se agrega una caché para mejorar la eficiencia de la red. Esta caché actúa como un mediador entre el cliente y el servidor, en el que las respuestas a solicitudes anteriores pueden, si se consideran almacenables en caché, reutilizarse en respuesta a solicitudes posteriores que son equivalentes y pueden dar como resultado una respuesta idéntica a la de la caché si la solicitud fue enviada al servidor. Agregar una caché trae consigo la ventaja de que tienen el potencial de eliminar parcial o completamente algunas interacciones, mejorando la eficiencia y el rendimiento percibido por el usuario. La desventaja de las memorias caché es que la propiedad de confiabilidad disminuye si los datos quedan obsoletos dentro de la misma y estos difieren significativamente de los datos que se hubieran obtenido si la solicitud hubiera sido atendida directamente por el servidor.

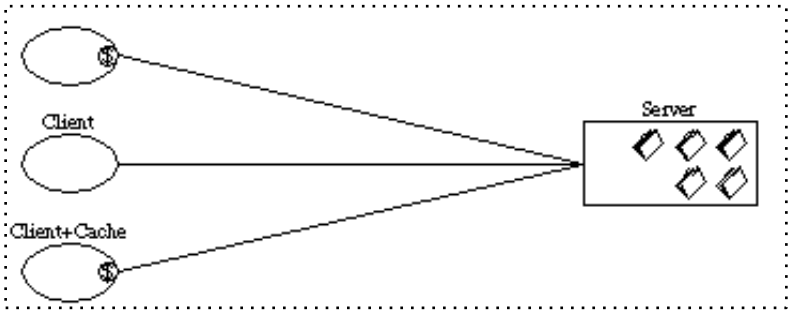

Ilustración 7. Imagen de la arquitectura cliente-servidor con caché.

- **Interfaz uniforme:** Poseer una interfaz uniforme entre los componentes es la principal característica que hace diferenciar a REST de otras arquitecturas basadas en red. Al darle generalidad a la interfaz del componente se logra simplificar la arquitectura general del sistema y se mejora la visibilidad de las interacciones. El desacoplamiento entre las implementaciones y los servicios que proporcionan promueve la capacidad de evolución independiente. El inconveniente que esto genera es que una interfaz uniforme degrada la eficiencia, ya que la información se transfiere en una forma estandarizada, en lugar de una forma específica a las necesidades de una aplicación, agregando sobrecarga y latencia al procesamiento de datos, lo que reduce el rendimiento percibido por el usuario. Este problema puede ser compensado con el uso de caché, gracias a los beneficios que esta conlleva explicados en el punto anterior.

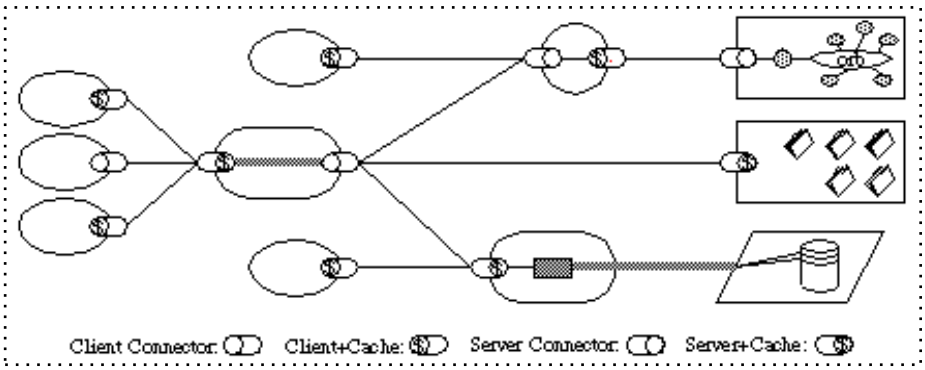

Ilustración 8. Imagen que muestra las interfaces uniformes usadas para conectar clientes y servidores.

- **Código bajo demanda:** Esta es una restricción opcional en REST. Con código bajo demanda, el cliente tiene acceso a un conjunto de recursos, pero no al “know-how” sobre cómo procesarlos, es decir, envía una solicitud a un servidor remoto para el código que representa ese “know-how”, recibe ese código y lo ejecuta en forma local. Las ventajas de esta restricción incluyen la capacidad de agregar funciones a un cliente ya implementado, lo que permite tener la capacidad de ampliación y configuración mejoradas, y un mejor rendimiento y eficiencia percibidos por el usuario cuando el código puede adaptar sus acciones al entorno del cliente e interactuar con el usuario localmente en lugar de a través de interacciones remotas. También, se obtiene una mejora en la escalabilidad del servidor, ya que puede descargar el trabajo al cliente y así, de esta forma, el servidor no consume sus recursos. Por otro lado, código bajo demanda trae consigo el inconveniente de la reducción de la simplicidad, debido a la necesidad de administrar el entorno de

evaluación. Esto puede compensarse, en algunos casos, como resultado de simplificar la funcionalidad estática del cliente. La principal limitación es la falta de visibilidad debido a que el servidor envía código en lugar de simples datos, lo que conduce a problemas de implementación si el cliente no confía en los servidores.

En resumen, REST proporciona un conjunto de restricciones arquitectónicas que, al aplicarse, acentúan la escalabilidad de las interacciones entre componentes, la generalidad de las interfaces y la implementación independiente de los componentes para reducir la latencia de la interacción.

FIPA

Fundación para Agentes Físicos Inteligentes, del inglés Foundation for Intelligent Physical Agents (FIPA), es una organización de estándares de la Sociedad de Computación IEEE que fomenta la tecnología basada en agentes y la interoperabilidad de sus estándares con otras tecnologías [FIPA[23]].

FIPA se formó en Suiza, en 1996, como una organización para la producción de especificaciones de estándares de software para agentes heterogéneos e interactivos y sistemas basados en agentes. Esta organización ha incidido fuertemente en el desarrollo de estándares de agentes y ha promovido una serie de iniciativas y eventos que aportaron al desarrollo y la adopción de las tecnologías de agentes. Además, muchas de las ideas que se originaron y desarrollaron en FIPA ahora se están enfocando en las nuevas generaciones de tecnologías web/internet y especificaciones relacionadas.

En Marzo de 2005, todos los miembros de FIPA votaron de forma unánime para unirse a la Sociedad de Computación IEEE. De esta forma, el 8 Junio de ese año y de forma oficial, esta organización de estándares para agentes y sistemas multiagentes fue aceptada oficialmente por la IEEE como su undécimo comité de estándares.

A partir de este hecho, FIPA comenzó a trasladar los estándares para agentes y sistemas basados en agentes al contexto más amplio del desarrollo de software.

Esta organización, actualmente se encuentra inactiva, pero su colección de estándares está abierta a todos y se pueden encontrar en el Repositorio de FIPA[24]. Este repositorio es un archivo explorable de todas las especificaciones de FIPA disponibles en cualquier etapa de su ciclo de vida. Cada especificación está representada por un número de documento y tiene un identificador de especificación formal e informal.

Un identificador informal está diseñado para ser utilizado internamente en esta organización para hacer referencia a una instancia determinada de una especificación en un punto particular de su ciclo de vida. Los identificadores informales de especificación se componen

[23] Sitio oficial FIPA - http://www.fipa.org/

[24] Repositorio de FIPA - http://www.fipa.org/specs/index.html

del estado del ciclo de vida de la especificación y su número de documento. Por ejemplo, “XC00023” se lee como “Número de especificación del componente experimental 23”.

Un identificador formal está diseñado para hacer referencia a las especificaciones de FIPA en otras especificaciones de FIPA y en documentos externos, como documentos de conferencias, diarios, etc. Un identificador de especificación formal se compone de la palabra clave “FIPA” y el número de documento, por ejemplo, “FIPA00023”. Este tipo de identificador, a diferencia del informal, no hace referencia a un estadío del ciclo de vida de la especificación, sino que hace referencia a la etapa que porta la última versión de esta especificación. Así, por ejemplo, si hubiera “PC00023” y “XC00023” disponibles para una especificación dada, la última versión para “FIPA00023” sería “XC00023”, ya que es una etapa más avanzada dentro del ciclo de vida de la especificación.

El ciclo de vida de las especificaciones detalla qué etapas puede alcanzar una especificación mientras forma parte del proceso de los estándares FIPA. Las especificaciones FIPA se clasifican según su posición en el ciclo de vida de las especificaciones. La intención del ciclo de vida de las especificaciones es trazar el progreso de una especificación dada desde su inicio hasta su resolución final.

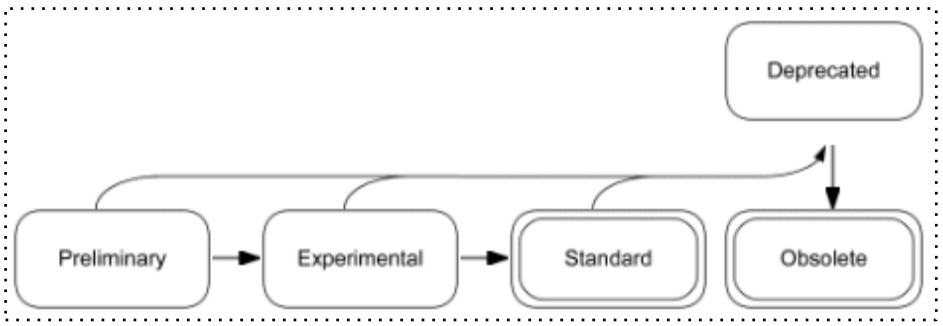

Ilustración 9. Imagen del ciclo de vida de las especificaciones FIPA.

Las 5 etapas del ciclo de vida de las especificaciones FIPA son:

1. **Preliminar:** La concepción inicial es conocida como etapa Preliminar y es generada por los Comités Técnicos. Esta etapa es considerada un borrador en construcción, inestable y no apta para implementación, ya que es susceptible a sufrir muchas revisiones o cambios. Todas las especificaciones Preliminares tienen un identificador de especificación informal que comienza con la letra “P”.
2. **Experimental:** Cuando una especificación Preliminar está lista, la misma avanza a la etapa Experimental, cambiando el identificador que comenzaba con “P”, ahora pasará a comenzar con “X”. Estas especificaciones son consideradas estables por un período de dos años o hasta que asciende al estado Estándar. Durante este período de tiempo solo se permite realizar cambios menores a la especificación a través de la aprobación de la Junta de Arquitectura FIPA. Una vez que se alcanza el estado Experimental, estas especificaciones son válidas para la implementación dentro de las plataformas de agentes compatibles con FIPA.
3. **Estándar:** Las implementaciones suficientes de una especificación en etapa Experimental y la retroalimentación de esas implementaciones determinan cuándo se puede presentar una especificación Estándar. Cuando la especificación alcanza este estadío obtiene un nuevo identificador de documento informal, que comienza

con “S”. Una especificación en etapa Estándar es considerada estable y fue aprobada y avalada por el organismo de normas FIPA. Para alcanzar el estado de Estándar, una especificación debe implementarse con éxito en más de una plataforma de agentes compatible con FIPA y son adecuadas para el lanzamiento e implementación general dentro de las plataformas de agentes compatibles con FIPA. Las especificaciones mantienen esta etapa hasta que se vuelven innecesarias y entran en Desuso.

4. **Desuso:** Una especificación en Desuso es una especificación que se ha identificado como potencialmente innecesaria para el estándar FIPA. Dicha especificación puede haberse convertido en innecesaria debido a cambios de tecnología o cambios en otras normas o especificaciones FIPA. Cuando una especificación entra en Desuso, su identificador de documento informal pasa a comenzar con la letra “D”. Una especificación puede pasar a Desuso desde cualquiera de las etapas anteriormente explicadas y luego quedar finalmente Obsoleta. Para que esto ocurra debe pasar un periodo de gracia para garantizar que la especificación sea definitivamente innecesaria para el estándar FIPA. Por el contrario, si durante el periodo de gracia se resuelve que la especificación no debe quedar obsoleta, entonces pasa del estado en Desuso al estado original que poseía antes de quedar en Desuso, asignándole el identificador de documento informal apropiado.
5. **Obsoleta:** Todas las especificaciones Obsoletas tienen un identificador informal con “O”. Por lo explicado en la etapa anterior, una especificación se vuelve Obsoleta luego de estar en Desuso y haber expirado su periodo de gracia. Es posible revertir el estado de Obsoleta, si se considera necesario, convirtiéndose así en una nueva especificación Preliminar con un nuevo identificador de documento.

Para el desarrollo de esta tesina de grado se utilizaron las especificaciones FIPA “SC00061G FIPA ACL Message Structure Specification”, “SC00037J FIPA Communicative Act Library Specification” y “SC00029H FIPA Contract Net Interaction Protocol Specification”, tres estándares reconocidos por esta organización, como se puede observar por sus identificadores informales, todos comienzan con la letra “S”, los cuales serán explicados en detalle a continuación.

ACL Message Structure

Este estándar contiene especificaciones para los parámetros del mensaje ACL (Agent-Communication-Language) de FIPA y tiene la finalidad de ayudar a garantizar la interoperabilidad al proporcionar un conjunto estándar de estructura de mensajes ACL, y proporcionar un proceso bien definido para mantener este conjunto [FIPA[25]].

Un mensaje FIPA ACL contiene un conjunto de uno o más parámetros de mensaje.Estos parámetros variarán dependiendo de la situación en que se encuentran los agentes intervinientes de la conversación. El único parámetro obligatorio en todos los mensajes ACL es el parámetro “performative”, aunque es deseable que los mensajes también contengan los parámetros “sender”, “receiver” y “content”.

[25] Foundation For Intelligent Physical Agents. (2002, December 3). ”Especificación FIPA ACL Message Structure” Retrieved from http://www.fipa.org/specs/fipa00061/SC00061G.html

El usuario del estándar es libre de implementar sus propios parámetros, además de los definidos por FIPA. También, a través del contexto de la conversación, algunos parámetros podrían omitirse. Sin embargo, FIPA no posee ningún mecanismo para manejar tales condiciones, por lo tanto, si estas omisiones se utilizan, no es posible garantizar que estas implementaciones interactúen correctamente entre sí.

Los posibles parámetros para un mensaje ACL, establecidos por FIPA, son los siguientes:

Parámetro	**Categoría de parámetro**
performative	Tipo de acto comunicativo
sender	Participante en la comunicación
receiver	Participante en la comunicación
reply-to	Participante en la comunicación
content	Contenido del mensaje
language	Descripción del contenido
encoding	Descripción del contenido
ontology	Descripción del contenido
protocol	Control de conversación
conversation-id	Control de conversación
reply-with	Control de conversación
in-reply-to	Control de conversación
reply-by	Control de conversación

Tabla 1. Parametros para un mensaje ACL

- **performative:** Este parámetro indica el tipo de acto comunicativo del mensaje ACL y sus posibles valores reservados serán explicados más adelante en el estándar “SC00037J FIPA Communicative Act Library Specification”. Este parámetro es el único obligatorio en los mensajes ACL de FIPA.
- **sender:** Este parámetro contiene la identidad del remitente o emisor del mensaje.
- **receiver:** Al contrario de **sender**, **receiver** indica la identidad de los destinatarios del mensaje enviado. Este puede contener un nombre o un conjunto de nombres de agentes, en caso de ser un mensaje multidifusión.
- **reply-to:** Como lo indica su traducción (“responder-a”), expresa que los mensajes subsiguientes en el hilo de conversación deben dirigirse al agente nombrado en el parámetro de respuesta, en lugar de al agente nombrado en el parámetro **sender**.
- **content:** El contenido del mensaje.

- **language:** Indica el idioma en el que se expresa el parámetro **content**. Sus posibles valores pueden verse en el documento FIPA00007[26], actualmente en Desuso.
- **encoding:** Este parámetro indica la codificación específica del lenguaje del contenido. El contenido puede estar codificado de varias maneras, las cuales pueden verse también en el documento FIPA00007.
- **ontology:** Denota la/s ontología/s utilizada/s para dar significado a los símbolos del contenido. Este parámetro se utiliza en conjunto con **language** para apoyar la interpretación del contenido por parte del agente receptor.
- **protocol:** Dicho parámetro indica el protocolo de interacción que el agente emisor está empleando con el mensaje ACL. Si este parámetro se utiliza, se considera que pertenece a una conversación, por ende, el mensaje ACL debe respetar las siguientes reglas:
    - El iniciador del protocolo debe asignar un valor al parámetro **conversation-id**.
    - Todas las respuestas al mensaje, dentro del alcance del mismo protocolo de interacción, deben contener el mismo valor para el parámetro **conversation-id**.
    - El valor de tiempo de espera (timeout) en el parámetro **reply-by** debe indicar la última hora en que el agente emisor desearía haber recibido el siguiente mensaje en el flujo del protocolo (no confundir con la última hora en que el protocolo de interacción debe terminar).
- **conversation-id:** Identificador de conversación que sirve para identificar la secuencia en curso de actos comunicativos que forman una conversación.
- **reply-with:** Expresión que utilizará el agente que responde para identificar este mensaje. Como indica su traducción (“responder-con”), este parámetro está diseñado para seguir un hilo de conversación en una situación en la que se producen varios diálogos simultáneamente. Por ejemplo, el agente *i* envía un mensaje con el parámetro **reply-with: <expr>**, entonces el agente *j* contesta con un mensaje con el parámetro **in-reply-to: <expr>**.
- **in-reply-to:** Expresa que este mensaje es una respuesta de otro mensaje anterior. Debe contener la misma expresión que el parámetro **reply-with** al que responde.
- **reply-by:** Hora y/o fecha límite que el agente emisor desea recibir una respuesta.

Communicative Act Library

Biblioteca del Acto Comunicativo, del inglés Communicative Act Library (CAL), es la especificación del estándar FIPA que define la estructuración y semántica de cada acto comunicativo FIPA [FIPA[27]].

Este estándar está desarrollado para cumplir los siguientes objetivos:

[26] FIPA00007 - http://www.fipa.org/specs/fipa00007/DC00007C.html

[27] Foundation For Intelligent Physical Agents. (2002, December 3). “FIPA Communicative Act Library Specification”. Retrieved from http://www.fipa.org/specs/fipa00037/SC00037J.html.

- Contribuir en la ayuda para garantizar la interoperabilidad al brindar un conjunto estándar de actos comunicativos compuestos, derivados de los actos comunicativos primitivos de FIPA.
- Facilitar la reutilización de actos comunicativos compuestos.
- Proporcionar un proceso bien definido para mantener un conjunto de actos comunicativos y etiquetas de actos para su uso en el ACL de FIPA.

CAL es una recopilación pública y estandarizada de definiciones de actos comunicativos, lo que facilita el uso de estos actos por parte de los agentes desarrollados en diferentes contextos.

FIPA es responsable de mantener una lista consistente de nombres de actos comunicativos aprobados y propuestos y de hacer que esta lista esté disponible públicamente. La misma, se deriva del estándar CAL de FIPA.

El nombre asignado a un acto comunicativo debe identificar unívocamente qué acto comunicativo se usa dentro de un mensaje ACL de FIPA, no debe causar conflicto con ningún nombre actualmente en la biblioteca y debe ser una palabra o abreviatura en inglés que sugiera la semántica.

Además, de la caracterización semántica y la información descriptiva que se requiere, cada acto comunicativo en este estándar puede especificar información adicional, como información de estabilidad, de contacto, control de versiones, niveles de soporte, etc.

Actualmente, CAL cuenta con 22 especificaciones de actos comunicativos. A continuación se explicarán aquellos que son utilizados en el framework de esta tesina de grado.

Call for Proposal

Abreviado como *cfp*, este acto comunicativo es una acción de propósito general para iniciar un proceso de negociación al hacer una convocatoria de propuestas para realizar una acción dada. El protocolo real bajo el cual se establece el proceso de negociación se conoce por acuerdo previo o se establece explícitamente en el parámetro de protocolo del mensaje.

Al enviar un mensaje de tipo *cfp*, se envía una tupla que contiene una expresión de acción que denota la acción a realizar, y una expresión referencial que define una proposición de un solo parámetro que da las condiciones previas de la acción.

El agente que responde a un *cfp* debe responder con una proposición que indique el valor del parámetro en la expresión de condición previa original. Por ejemplo, el *cfp* podría buscar propuestas para un viaje de Frankfurt a Munich, con la condición de que el modo de viaje sea en tren. Una propuesta compatible en respuesta sería para el tren expreso 10.45. Una propuesta incompatible sería viajar en avión.

Notar que *cfp* también se puede usar para verificar simplemente la disponibilidad de un agente para realizar alguna acción. Además, esta formalización de *cfp* está restringida al caso común de propuestas caracterizadas por un solo parámetro en la expresión de la

propuesta. Otros escenarios pueden involucrar múltiples parámetros de propuesta, curvas de demanda, respuestas de forma libre, etc.

Propose

Este es un acto de propósito general para hacer una propuesta o responder a una propuesta existente durante un proceso de negociación, al proponer que se realice una acción determinada sujeta a ciertas condiciones que son verdaderas. El protocolo real bajo el cual se lleva a cabo el proceso de negociación se conoce por acuerdo previo o se indica explícitamente en el parámetro de protocolo del mensaje.

Al enviar un mensaje del tipo *propose,* se envía una tupla que contiene una descripción de la acción, que representa la acción que el emisor propone realizar, y una proposición que representa las condiciones previas en el desempeño de la acción. De esta forma, el emisor informa al receptor que el emisor adoptará la intención de realizar la acción una vez que se cumpla la condición previa, y el receptor notifique al emisor la intención del receptor de que el emisor realice la acción.

Un uso típico de la condición adjunta a la propuesta es especificar el precio de una oferta en un protocolo de subasta o negociación.

Accept Proposal

El acto comunicativo *accept-proposal* es una aceptación de propósito general de una propuesta que se presentó anteriormente (generalmente a través de un acto *propose*). El agente que envía la aceptación informa al receptor que tiene la intención de que (en algún momento del futuro) el agente receptor realizará la acción, una vez que la condición sea cierta.

Este tipo de acto comunicativo envía una tupla que consiste en una expresión de acción que denota la acción a realizar, y una proposición que da las condiciones del acuerdo. Esta proposición dada como parte de la aceptación indica las condiciones previas que el agente está adjuntando a la aceptación. Un uso típico de esto es finalizar los detalles de un acuerdo en algún protocolo. Por ejemplo, una oferta anterior para “celebrar una reunión en cualquier momento del martes” podría aceptarse con una condición adicional de que la hora de la reunión es a las 11.00 hs.

Notar que un agente puede intentar que una acción se realice sin tener necesariamente la condición previa. Por ejemplo, durante la negociación sobre una tarea determinada, las partes negociadoras pueden no intentar de manera inequívoca sus ofertas de apertura: el agente *A* puede ofrecer un precio *P1* como condición previa, pero debe estar preparado para aceptar el precio *P2.*

Reject Proposal

Este acto comunicativo es un rechazo de propósito general a una propuesta presentada anteriormente. El agente emisor del mensaje informa al agente receptor que no tiene intención de que el destinatario realice la acción dada en las condiciones previas dadas.

El contenido de este tipo de mensaje es una tupla que consta de una descripción de acción y una proposición que formó la propuesta original que se rechazó, y una proposición adicional que denota la razón del rechazo.

Refuse

Este acto representa la acción de negarse a realizar una acción determinada y explicar el motivo de la negativa.

Como contenido del mensaje se envía una tupla, que consiste en una expresión de acción y una proposición que indica la razón de la negativa.

El acto *refuse* se realiza cuando el agente no puede cumplir todas las condiciones previas para que se lleve a cabo la acción, tanto implícita como explícita. Por ejemplo, es posible que el agente no sepa algo que se le está solicitando, u otro agente solicitó una acción para la cual no tiene suficientes privilegios.

Cuando un agente recibe un mensaje del tipo *refuse* el mismo puede sacar como conclusión que la acción no se ha realizado, que la acción no es factible (desde el punto de vista del remitente de la denegación), y la razón de la negativa está representada por la proposición del segundo elemento de la tupla del contenido del mensaje.

Failure

El acto *failure* representa a la acción de decirle a otro agente que se intentó una acción pero el intento falló.

El mismo consta de una tupla que contiene una expresión de acción y una proposición que da la razón del fracaso.

El agente receptor de un mensaje *failure* entiende que la acción no se ha rechazado, y que la acción es o, al menos en el momento en que el agente intentó realizar la acción, fue factible.

Inform Ref

Los mensajes de tipo *inform-ref,* no son actos comunicativos como los anteriormente nombrados, estos son actos macro. Los actos comunicativos pueden ser realizados directamente, pueden ser planeados por un agente y pueden ser solicitados a un agente por otro. En cambio, los actos macro pueden ser planeados y solicitados, pero no realizados directamente.

Este acto permite al emisor informar al receptor sobre algún objeto que el emisor cree que corresponde a un descriptor, como un nombre u otra descripción de identificación. Por ejemplo, un agente puede planificar un *inform.ref* de la hora actual para el agente *j,* y luego realizar el acto representado por “informar a *j* que la hora es 10:45 am”.

El agente emisor de este tipo de acto comunicativo no puede asumir que el receptor del mismo ya sepa qué objeto o conjunto de objetos corresponden a la expresión enviada como referencia. Es por esto que el agente receptor puede optar por responder con un mensaje *refuse* si no puede establecer las condiciones del acto.

Contract Net Interaction Protocol

El Protocolo de Interacción de Red Contractual de FIPA, o FIPA Contract Net Interaction Protocolo (IP), es una especificación sobre el flujo de interacción entre los diferentes actos comunicativos. El mismo indica qué tipo de mensaje se debe utilizar para responder ante la recepción de cada acto [FIPA[28]].

En este estándar, un agente (el *Iniciador*) asume el rol de administrador que desea que uno o más agentes (los *Participantes*) realicen alguna tarea y desea optimizar una función que caracteriza la tarea. Esta característica durante la negociación, se puede expresar de diferentes maneras, como por ejemplo, el precio, el momento más pronto para completar la tarea, la distribución justa de tareas, entre otras. Para una tarea determinada, cualquier número de *Participantes* puede responder con una propuesta, el resto debe negarse. Luego continúan las negociaciones con los *Participantes* que propusieron.

[28] Foundation For Intelligent Physical Agents. (2002, December 3). “Especificación FIPA Contract Net Interaction Protocol”. Retrieved from http://www.fipa.org/specs/fipa00029/SC00029H.html

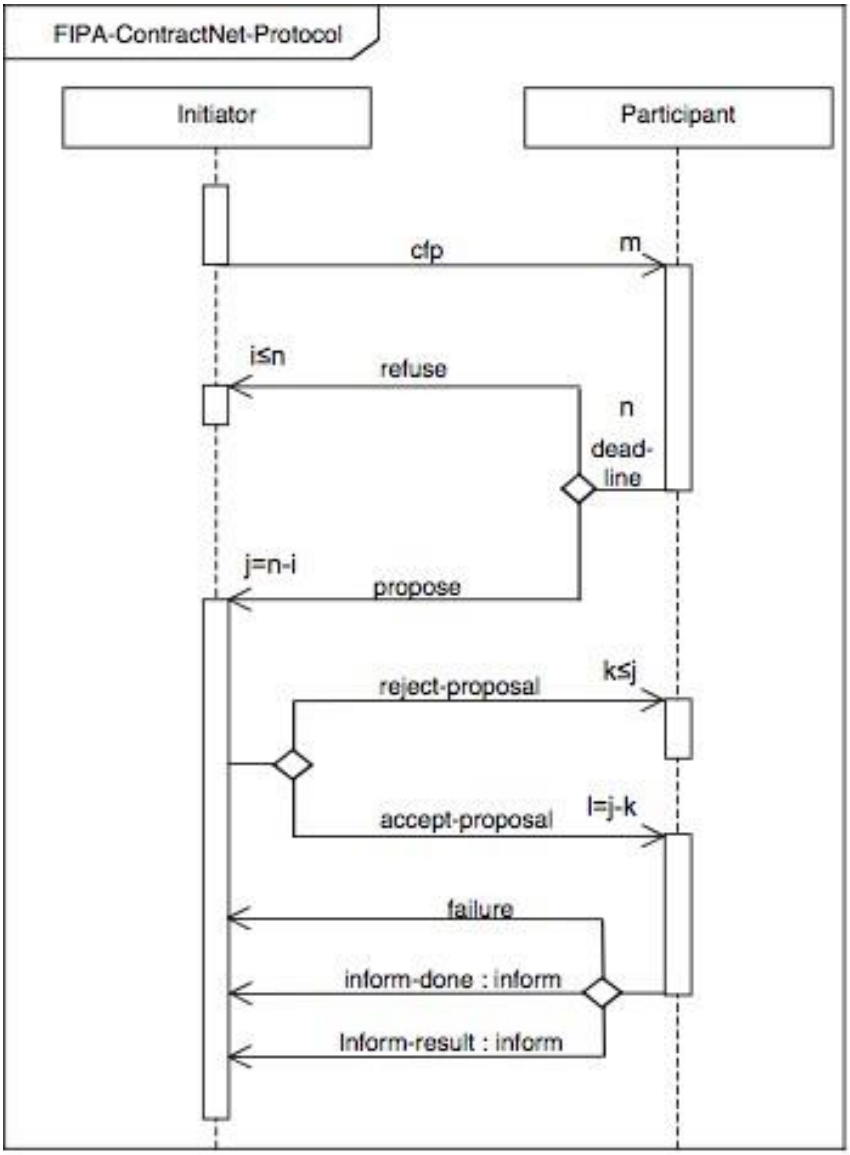

Ilustración 10. Imagen del flujo posible de interacción entre los distintos actos comunicativos

El flujo comienza cuando el *Iniciador* solicita *m* propuestas de otros agentes al emitir actos del tipo *call-for-proposal* (*cfp*), en los cuales se especifica una tarea, así como cualquier condición que el *Iniciador* esté poniendo sobre la ejecución de la tarea. Los *Participantes* que reciben el *cfp* pueden generar *n* respuestas, las cuales *j* son propuestas para realizar la tarea (acto comunicativo *propose*) y las demás son rechazos (acto comunicativo *refuse*).

Una vez que el plazo de espera del *Iniciador* finaliza, este agente evalúa las *j* propuestas recibidas y elige qué agente/s realizará/n la tarea. Uno, varios o ninguno pueden ser elegidos. Si un agente es seleccionado para realizar la tarea, entonces el *Iniciador* le enviará un mensaje del tipo *accept-proposal,* caso contrario, le enviará un mensaje del tipo *reject-proposal.* Las propuestas son vinculantes para el *Participante,* de modo que una vez que el *Iniciador* acepta la propuesta, el *Participante* adquiere un compromiso para realizar la tarea. Cuando el *Participante* ha completado la tarea, envía un mensaje al *Iniciador* con formato de *inform-done* (informe de realizado) o su versión más explicativa *inform-result* (informe de resultado). Sin embargo, si el *Participante* no logra completar la tarea, se envía un mensaje del tipo *failure.* Notar que los mensajes *inform-done* e *inform-result* son

derivados del acto comunicativo *inform-ref,* con el fin de brindarle al protocolo mayor claridad, además, tener en cuenta que el acto *inform-result* implica al acto *inform-done.*

Un requerimiento a cumplir para IP es que el *Iniciador* debe saber cuándo ha recibido todas las respuestas. En caso de que algún *Participante* no responda, el *Iniciador* podría esperar indefinidamente. Para prevenir esto, los mensajes del tipo *cfp* contienen el parámetro *reply-by,* que indicará la fecha límite en la cual el *Iniciador* debe recibir las respuestas. Una vez pasado este plazo, las propuestas recibidas posteriormente serán rechazadas automáticamente.

Todas las interacciones entre agentes a través de este protocolo debe ser identificada mediante un parámetro de identificación de conversación (*conversation-id*) no nulo, único, asignado por el *Iniciador.* Los agentes pertenecientes a una interacción deben etiquetar a todos sus mensajes con este identificador de conversación. De esta manera, cada agente es capaz de administrar sus estrategias y actividades de comunicación, por ejemplo, le permite a un agente identificar conversaciones individuales y razonar a través de registros históricos de conversaciones. En los casos de interacción *1:n,* es decir, un *Iniciador* y *n Participantes,* el agente administrador es el encargado de decidir si se debe usar el mismo *conversation-id* para los mensajes enviados a todos los *Participantes,* o si se genera uno nuevo para cada agente.

Cancel Meta Protocol

El Contract Net Interaction Protocol cuenta con un meta protocolo, el Cancel Meta Protocol. Este meta protocolo puede ejecutarse en cualquier punto del flujo de la conversación y especifica que el agente *Iniciador* puede cancelar el protocolo de interacción. Los mensajes intervinientes en este meta protocolo mantienen el *conversation-id* de la conversación en curso que se quiere cancelar. La semántica de cancelación debe interpretarse aproximadamente como que el *Iniciador* ya no está interesado en continuar la interacción y que debe terminarse de manera aceptable para el *Iniciador* y el *Participante.* El *Participante* informa al *Iniciador* que la interacción es finalizada correctamente, a través de un *inform-done,* o que surgió algún fallo durante la cancelación, enviando un *failure.*

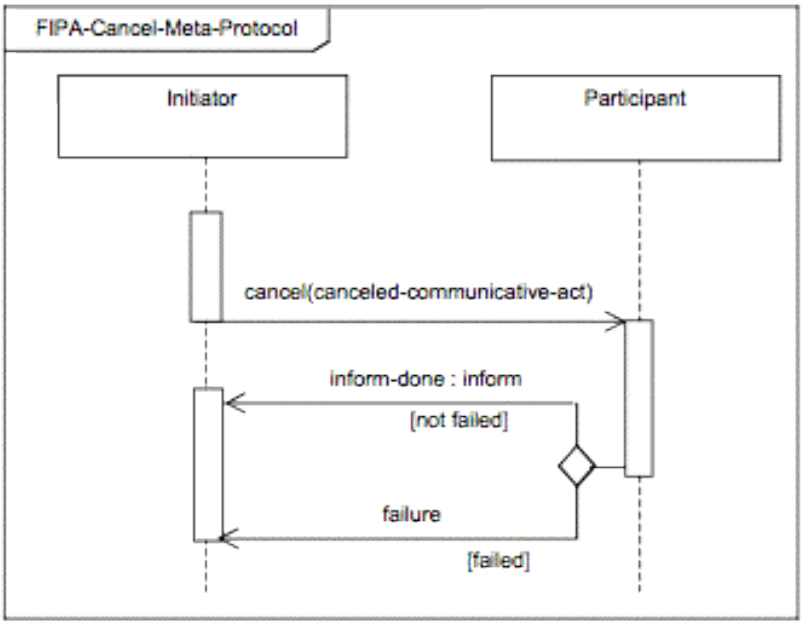

Ilustración 11. Imagen del flujo del meta protocolo de cancelación.

NoSQL

Como arquitectura para el almacenamiento de datos de esta tesina de grado se utilizó NoSQL, pero antes de definir y explicar lo que es, se debe hacer una breve introducción sobre lo que es SQL y las bases de datos relacionales.

Bases de datos

Una base de datos, no es más que un conjunto de datos relacionados. Una guía telefónica es un claro ejemplo de base de datos de nombres, números de teléfonos y direcciones de todas las personas que viven en una región en particular, pero cuenta con los siguientes problemas [Beaulieu - 2009[29]]:

- Encontrar un número de teléfono de una persona puede llevar mucho tiempo, especialmente si la guía contiene una gran cantidad de registros.
- Una guía telefónica está indexada sólo por apellido / nombre, por lo que encontrar los nombres de las personas que viven en una dirección particular, aunque es posible en teoría, no es un uso práctico para esta base de datos..
- Desde el momento en que se imprime esta guía, la información se vuelve cada vez menos precisa a medida que la gente ingresa o sale de una región, cambian sus número de teléfono o se trasladan a otra ubicación dentro de la misma región.

Estos mismos problemas también pueden ser aplicados a cualquier sistema de almacenamiento de información manual. Es por esto que algunas de las primeras aplicaciones informáticas fueron sistemas de bases de datos, es decir, almacenamiento de

[29] Alan Beaulie. “Learning SQL”. O’Reilly Media, 2009.

datos computarizados y mecanismos de recuperación de datos. Ya que estos sistemas de bases de datos almacenan información electrónicamente, en lugar de en papel, el mismo puede recuperar esos datos más rápidamente, indexándolos de múltiples maneras y ofreciendo información actualizada a sus usuarios.

Bases de datos relacionales y SQL

En 1970, el Dr. E. F. Codd, del laboratorio de investigación de IBM, publicó un documento titulado “Un modelo relacional de datos para grandes bancos de datos compartidos”, que proponía que los datos se representen como conjuntos de tablas, donde los datos redundantes se utilicen para vincular registros en diferentes tablas.

Junto con la definición del modelo relacional de Codd, se propuso un lenguaje llamado *DSL/Alpha* para manipular los datos en tablas relacionales. Posteriormente, IBM encargó la construcción de un prototipo basado en esta idea. Como resultado de esto se surgió una versión simplificada de *DSL/Alpha*, llamado *SQUARE*. Este lenguaje, a lo largo del tiempo, se fue refinando, y fue renombrado como *SEQUEL* y finalmente *SQL*.

SQL ha sufrido muchos cambios desde su creación. A mediados de la década de 1980, el *American National Standards Institute* (ANSI[30]) comenzó a trabajar en el primer estándar para el lenguaje *SQL*, que se publicó en 1986. Los refinamientos posteriores llevaron a nuevas versiones del estándar *SQL*, y entre las nuevas características agregadas podemos encontrar funciones orientadas a objetos, integración con *XML* (eXtensible Markup Language o lenguaje de marcado extensible), definición de lenguaje *XQuery* (lenguaje que se utiliza para consultar datos en documentos *XML*), entre otras.

En otras palabras, el *lenguaje de consulta estructurado* (Structured Query Languaje o *SQL*), es uno de los mecanismos más aceptados universalmente para acceder y manipular los datos almacenados en un sistema de administración de bases de datos relacionales, también conocidos como *RDMS* por su sigla en inglés, *Relational Database Management System*. El mismo, es un lenguaje basado en texto, que le permite al usuario describir completamente la estructura jerárquica de una base de datos relacional en una consulta, lo que hace posible crear poderosas y complejas consultas de manera arbitraria y directa [Jamison 2003[31]].

Bases de datos NoSQL

En la última década, la popularidad de las *bases de datos no relacionales* (o NoSQL) han aumentado drásticamente. Con el aumento de la accesibilidad de internet y la disponibilidad de almacenamiento barato, se capturan y almacenan enormes cantidades de datos estructurados, semiestructurados y no estructurados para una variedad de aplicaciones. Dichos datos se conocen comúnmente como *Big Data*. El procesamiento de tal cantidad de

[30] ANSI - https://www.ansi.org/

[31] Jamison, D. C. “Structured Query Language (SQL) Fundamentals”. Current Protocols in Bioinformatics, 00(1), 9.2.1–9.2.29, 2003

datos requiere velocidad, esquemas flexibles y bases de datos distribuidas. Las bases de datos NoSQL son las preferidas para operar *Big Data*. Esto también llevó a un aumento en el número de ofertas de bases de datos NoSQL. Existen varias implementaciones comerciales y de código abierto de las bases de datos NoSQL, como *BigTable*[32] y *HBase*[33].[Li y Manoharan 2013[34]].

Se cree que *BigTable* de Google fue la primera base de datos NoSQL y junto a *DynamoDB*[35] de Amazon recaudaron gran éxito, lo cual impulsó el desarrollo de nuevas bases de datos no relacionales de código abierto y cerrado. Este éxito se logró gracias a su facilidad de acceso, velocidad y escalabilidad.

Las bases de datos NoSQL cuentan con esquemas flexibles y están diseñadas para modelos de datos específicos. Para estos modelos de datos se cuenta con una gran variedad, que incluyen documentos, grafos, clave-valor, entre otros, que están optimizados específicamente para aplicaciones que requieren grandes volúmenes de datos, baja latencia y modelos de datos flexibles, características otorgadas por la flexibilización en la coherencia de datos, propiedad restringida en las bases de datos relacionales [Amazon[36]].

Propiedades de las bases de datos NoSQL

Las bases de datos no relacionales cuentan con las propiedades de tener esquemas dinámicos, la posibilidad de sharding automático, replicación automática y una capa de caché integrada. A continuación se desarrollarán estas propiedades más en detalle [MongoDB[37]].

Esquemas dinámicos

Las bases de datos relacionales cuentan con esquemas estáticos, ya que si se desea, por ejemplo, guardar datos de clientes, tales como el nombre, email, teléfono, etc, en SQL es necesario conocer el tipo de dato de antemano que se va a almacenar. Esto causa que, cada característica nueva que se quiera agregar, modifique el esquema de la base de datos, y luego de cada modificación de esquema se debe migrar la base de datos entera al nuevo esquema, por ende, esto afecta a la agilidad en el desarrollo.

En cambio, en las bases de datos no relacionales se pueden insertar datos sin contar con un esquema predefinido. Como resultante se pueden realizar modificaciones importantes en tiempo real y sin interrupciones del servicio. Gracias a esto, el desarrollo es más rápido, la integración del código es más fiable y los administradores de bases de datos tienen un trabajo menor.

---

[32] Sitio oficial de BigTable - https://cloud.google.com/bigtable/

[33] Sitio oficial de HBase - https://hbase.apache.org/

[34] Li, Y., & Manoharan, S. “A performance comparison of SQL and NoSQL databases”. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), 2013

[35] Amazon DynamoDB - https://aws.amazon.com/dynamodb/

[36] Amazon - “What is NoSQL?” - https://aws.amazon.com/nosql/

[37] MongoDB - https://www.mongodb.com/nosql-explained

Además, al utilizar bases de datos relacionales, se debe contar con validaciones del lado de las aplicaciones para realizar controles de calidad de los datos, tales como, la obligatoriedad de campos, tipos de datos o valores específicos. Esto es algo que con NoSQL se puede aplicar del lado de la base de datos, a través de reglas de validación, aprovechando la agilidad que ofrece un esquema dinámico.

Sharding automático

Por su carácter de esquema estructurado, las bases de datos relacionales suelen escalar de forma vertical, es decir, un solo servidor hospeda a toda la base de datos para garantizar un rendimiento aceptable de las operaciones *JOIN* y transacciones entre tablas. Esto causa un aumento en los costos, limita la escalabilidad y crea un número de puntos de fallo relativamente pequeño para la infraestructura de la base de datos.

Para solucionar estas cuestiones, se utiliza la escalabilidad horizontal. En este tipo de escalabilidad no se agrega más capacidad a un único servidor, sino que se agregan más servidores.

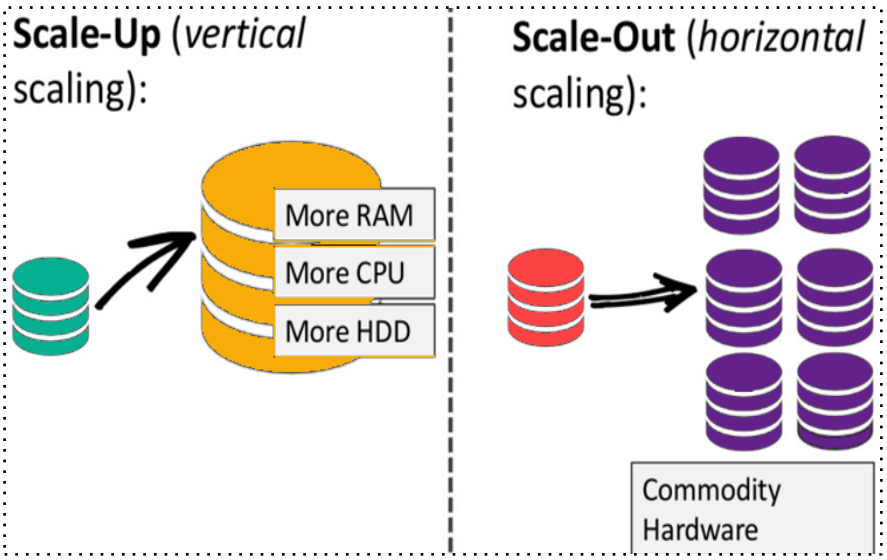

Ilustración 12. Descripción gráfica de escalado vertical y horizontal.

Sharding es la posibilidad de distribuir los datos entre los servidores de bases de datos. Aún contando con una base de datos relacional se puede lograr *sharding* entre múltiples instancias de servidor, pero para esto se necesitan sistemas SAN (Storage Area Network o red de área de almacenamiento, es una arquitectura completa de red de almacenamiento integral) y otras configuraciones complejas para que el hardware actúe como un único servidor. Además, SQL al no contar de forma nativa con esta característica, se deben implementar varias bases de datos en diferentes servidores, las cuales cuentan con datos autónomos. Esto causa que sea necesario el desarrollo de la aplicación de forma que distribuya los datos, las consultas y agregue los resultados de los datos en todas las instancias de base de datos. Adicionalmente, es necesario desarrollar mecanismos para el manejo de fallos de recursos, para realizar operaciones JOIN entre diferentes bases de

datos, para aplicar reequilibrio de datos y para aplicar replicación. Además, al utilizar *sharding manual* se pierden ventajas de las bases relacionales, como por ejemplo la integridad transaccional.

Las bases de datos NoSQL, en cambio, pueden realizar *auto-sharding,* es decir, pueden distribuir los datos de forma nativa y automática (sin la necesidad de agregar arquitecturas, configuraciones o mecanismos extras) entre un número arbitrario de servidores, sin que la aplicación deba tener constancia de la composición del grupo de servidores. Además, los datos y la carga de consultas se equilibran, de forma automática, entre los servidores, y cuando uno falla, se puede sustituir de forma rápida y transparente sin que la aplicación deje de funcionar.

Replicación

Para garantizar la disponibilidad en caso de que se produzcan interrupciones de servicio o paradas de mantenimiento planificadas, la mayoría de las bases de datos no relacionales cuentan con la posibilidad de la replicación automática de base de datos. Las bases más sofisticadas son autorreparables y ofrecen la función de conmutación por error[38] y recuperación. También, para soportar fallos regionales, cuentan con la posibilidad de distribuir las bases de datos entre varias regiones geográficas y permitir la localización de los datos. Todas estas características son posibles sin la necesidad de utilizar otras aplicaciones o complementos, algo que para las bases de datos relacionales no es posible.

Caché integrada

Los sistemas de caché se utilizan para mejorar el tiempo de lectura de la base de datos de forma importante. A las bases de datos SQL se les puede agregar un sistema de manejo de caché. Esta opción sólo es viable cuando la aplicación que utilizará la base de datos cuenta mayoritariamente con operaciones de lectura, por el contrario, si la mayoría de operaciones realizadas sobre la base de datos son de escritura u otras, la utilización de sistemas de caché, no solo no mostrará un incremento en la performance, sino que también agregará una mayor complejidad operativa a las implementaciones, a la hora de gestionar la invalidación de caché.

En contraposición a las bases de datos relacionales, muchas de las bases NoSQL incorporan excelentes funcionalidades de caché, sin la necesidad de una capa extra para esto. También, para los casos en que se tiene una carga de trabajo que requiere el máximo rendimiento y una latencia mínima, algunas de las bases NoSQL también ofrecen una capa de gestión de la base de datos integrada en su memoria.

[38] Conmutación por error o failover, es un modo de funcionamiento de respaldo en el que las funciones de un componente del sistema (tal como un procesador, servidor, base de datos, etc) son asumidos por componentes del sistema secundario, cuando el componente principal no está disponible.

Desventajas del NoSQL

Gracias a las propiedades antes mencionadas, las bases de datos NoSQL cuentan con múltiples ventajas frente a las bases de datos relacionales. Pero también poseen algunas desventajas detalladas a continuación:

- La mayoría de las bases de datos NoSQL no admiten funciones de fiabilidad, característica que si se encuentra en las bases de datos SQL. Como fiabilidad podemos entender a las propiedades de *Atomicidad*[39], *Consistencia*[40], *Aislamiento*[41] y *Durabilidad*[42] (ACID, del inglés Atomicity, Consistency, Isolation and Durability).
- Para subsanar la desventaja anterior, el desarrollador debe encargarse de implementar mecanismos para lograr la fiabilidad y coherencia, lo que representa una mayor complejidad para el sistema.
- Los dos puntos anteriores conllevan a la reducción del número de aplicaciones confiables para realizar transacción seguras con datos sensibles, tales como sistemas bancarios.
- Además, la mayoría de las bases de datos no relacionales sufren de incompatibilidad con consultas SQL, requiriendo así la utilización de otros sistemas que logren la traducción e integración entre los mismos, causando el aumento en la complejidad y la disminución en la velocidad de los procesos.

Tipos de bases de datos NoSQL

Existen numerosos tipos de bases de datos no relacionales, algunos de ellos son:

- **Clave-Valor:** son las bases de datos NoSQL más simples y a la vez, las que mayor escalado horizontal brindan en comparación con otras NoSQL. Cada elemento de la base de datos se almacena como un nombre de atributo (o *clave*), junto con su valor. Su utilización puede verse en aplicaciones de juegos, tecnología publicitaria e *IoT* (Internet of Things, o Internet de las Cosas, concepto que refiere a la interconexión digital de objetos cotidianos con internet). Algunos ejemplos son *Amazon DynamoDB*[43], *Riak*[44], *Berkeley*[45], entre otros.
- **Documentos:** en los sistemas informáticos, los datos se representan a menudo como un objeto o documento de tipo JSON. En este tipo de bases de datos, el almacenamiento de datos utiliza el mismo formato de modelo de documento que el código de la aplicación. Cada clave se empareja con esta estructura de datos compleja llamada *documento.* Los mismos pueden contener muchos pares de clave-

[39] Atomicidad indica que las transacciones son completas, es decir, cuando una operación consiste en una serie de pasos, bien todos ellos se ejecutan o bien ninguno.

[40] Consistencia asegura que sólo se empieza aquello que se puede acabar, es decir, que se ejecutan aquellas operaciones que no van a romper las reglas y directrices de integridad de la base de datos.

[41] Aislamiento, propiedad que indica que una operación no puede afectar a otras. Esto asegura que dos transacciones simultáneas sobre la misma información sean independientes y no generen ningún tipo de error.

[42] Durabilidad, propiedad que indica que una vez realizada una operación, la misma persiste y no se podrá deshacer aunque falle el sistema.

[43] Sitio oficial DynamoDB - https://aws.amazon.com/dynamodb

[44] Sitio oficial Riak - https://riak.com/

[45] Sitio oficial Oracle Berkeley - https://www.oracle.com/database/technologies/related/berkeleydb.html

valor distintos, pares de clave-matriz o incluso documentos anidados. La naturaleza flexible, semi estructurada y jerárquica de los documentos y las bases de datos de documentos permite que evolucionen según las necesidades de las aplicaciones. Este tipo de bases de datos no relacionales suelen ser utilizadas con sistemas de catálogos, perfiles de usuarios y sistemas de administración de contenido en los que cada documento es único y evoluciona con el tiempo. Algunos ejemplos de bases de datos de documentos son *MongoDB*[46], *Amazon DocumentDB*[47], *Elastic*[48], *Azure CosmosDB*[49], entre otros.

- **Orientadas a columnas:** ejemplos de estas bases de datos pueden ser *Cassandra*[50], *HBase*[51], *Scylla*[52], etc. Con ellas se pueden realizar consultas en grandes conjuntos de datos y almacenan los datos en columnas, en lugar de filas.
- **Grafos:** este tipo de base de datos se utiliza principalmente en aplicaciones que funcionan con conjuntos de datos altamente conectados, tales como redes sociales, motores de recomendaciones, detección de fraude o gráficos de conocimiento. Algunos ejemplos son *Neo4J*[53], *Giraph*[54], *HyperGraphDB*[55], etc.

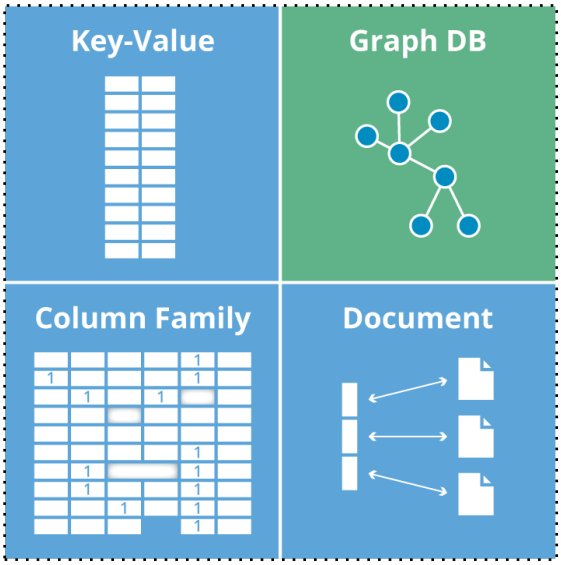

Ilustración 13. Descripción gráfica de cómo almacenan los datos algunas BD NoSQL.

Además de las antes explicadas, también se pueden encontrar bases de datos multimodelo (*ArangoDB, OrientDB, FaunaDB*, etc), bases de datos de objetos (*Versant, Objectivity, Perst, JADE*, etc), bases de datos XML (*EMC Documentum xDB, eXist, Sedna, BaseX, Berkeley DB XML*, etc) y muchas otras más [NoSQL[56]].

[46] Sitio oficial MongoDB - https://www.mongodb.com
[47] Sitio oficial DocumentDB - https://aws.amazon.com/documentdb
[48] Sitio oficial Elastic - https://www.elastic.co/
[49] Sitio oficial Azure CosmosDB - https://azure.microsoft.com/es-es/services/cosmos-db/
[50] Sitio oficial Apache Cassandra - https://cassandra.apache.org/
[51] Sitio oficial Apache HBase - https://hbase.apache.org/
[52] Sitio oficial Scylla - https://www.scylladb.com/
[53] Sitio oficial Neo4J - https://neo4j.com/
[54] Sitio oficial Giraph - https://giraph.apache.org/
[55] Sitio oficial HyperGraphDB - http://www.hypergraphdb.org/
[56] Sitio oficial NoSQL - http://nosql-database.org/

Comparación de terminologías SQL y NoSQL

La siguiente tabla compara la terminología utilizada por las bases de datos NoSQL seleccionadas, con la terminología utilizada por las bases de datos SQL

SQL	**MongoDB**	**DynamoDB**	**Cassandra**	**Couchbase**
Tabla	Colección	Tabla	Tabla	Cubo de datos
Fila	Documento	Elemento	Fila	Documento
Columna	Campo	Atributo	Columna	Campo
Clave principal	ID de objeto	Clave principal	Clave principal	ID de documento
Índice	Índice	Índice secundario	Índice	Índice
Vista	Vista	Índice secundario global	Vista materializada	Vista
Tabla u objeto anidado	Documento embebido	Mapa	Mapa	Mapa
Matriz	Matriz	Lista	Lista	Lista

Tabla 2. Terminologías SQL y NoSQL

Capítulo 6

Alcance del framework

Este framework permite el desarrollo de sistemas para la creación y simulación de cadenas de suministros. Esto se logra con la creación de clases de agentes, el despliegue de las mismas, el armado de agentes que utilicen las clases previamente creadas y por último, la realización de la simulación. Permitiendo, además, la posibilidad de integración con agentes o cadenas externas. Con esta flexibilidad, se pueden generar cadenas de agentes que interactúen con otras cadenas ya establecidas, pero con la restricción de que las mismas deben de utilizar el estándar de comunicación FIPA CAL.

Por otro lado, se tiene la herramienta gráfica que permite el armado y la simulación de cadenas de suministros. Con esta herramienta, el usuario del framework puede crear todas las cadenas que crea necesario, teniendo a su alcance la posibilidad de establecer cadenas de suministros como *Template* para luego poder reutilizar las mismas. Una vez establecidas las cadenas, el usuario deberá de crear en cada una de estas los agentes que crea necesario para la simulación.

Frozen spots

Son aquellos elementos inmutables del framework, que independientemente de los distintos sistemas que se puedan desarrollar con él, estos nunca van a cambiar, siempre van a hacer lo mismo. Si se tiene que clasificar a los frozen spots de este framework, sería en tres grandes grupos: fronzen spots en el almacenamiento de los datos, en la comunicación entre agentes y por último, en la ejecución de la simulación.

Los frozen spots responsables del almacenamiento de los datos, se encargan de guardar todos los datos que refieren al modelo del agente, incluyendo la memoria propia de cada agente modificable por el usuario. Es decir, estos proveen una estructura de almacenamiento de los datos del agente, permitiendo que el usuario se desentienda de esta tarea.

Los frozen sports que refieren a la comunicación entre agentes, hacen referencia tanto al envío como a la recepción de mensajes entre los mismos.

Y en cuanto a la ejecución de la simulación, el framework es el encargado de controlar que la simulación se lleve adelante, coordinando el trabajo entre la comunicación y la persistencia de los datos.

Hot spots

Los elementos que pueden ser modificados de un framework son denominados hot spots. Estos permiten implementar el comportamiento deseado para el software que se está desarrollando.

Hot spots programables

Esto es posible creando nuevas clases de agentes, las cuales se pueden generar utilizando uno de los comandos que provee el framework. El uso de este comando crea una clase que hereda su comportamiento de la clase principal y los posibles métodos a implementar, los hot spots programables.

Una vez creada la nueva clase de agente, se puede definir el comportamiento de la misma a través de la implementación de los distintos métodos que en ella figuran. Estos métodos están, en su mayoría, relacionados con lo que va hacer el agente ante los distintos mensajes que puede recibir. Es posible optar por definir este comportamiento o dejarlo en blanco, haciendo que el agente no reaccione ante un determinado tipo de mensaje.

Los hot spots relacionados a los mensajes que puede recibir el agente son *processCallForProposal, processPropose, processAcceptProposal, processRejectProposal, processRefuse, processFailure, processInformDone y processInformResult.*

```
'use strict';
var Agent = require('../agent');
let Message = require('../message');

class Example extends Agent {

  constructor(dataAgent={}){
    dataAgent.agentType = 'Example';
    super(dataAgent);
  }
  // Subclasses should redefine this method
  defaultStore(){
  }
  // Subclasses should redefine this method
  processCallForProposal(message){
  }
  // Subclasses should redefine this method
  processPropose(message){
  }
  // Subclasses should redefine this method
  processAcceptProposal(message){
  }
  // Subclasses should redefine this method
  processRejectProposal(message){
  }
  // Subclasses should redefine this method
  processRefuse(message){
  }
  // Subclasses should redefine this method
  processFailure(message){
  }
  // Subclasses should redefine this method
  processInformDone(message){
  }
  // Subclasses should redefine this method
  processInformResult(message){
  }
}

module.exports = Example;
```

Ilustración 14. Clase Example.

Además de los métodos relacionados a la recepción de mensajes, se debe definir el modelo por defecto de la clase. Lo que se defina en el método *defaultStore()*, se va a utilizar como una estructura de la memoria interna de cada agente. Esta puede ser utilizada para guardar stock, indicar los productos que el agente puede satisfacer y todo lo que el programador de la clase crea necesario almacenar. Este modelo es utilizado por la herramienta gráfica como un modelo de memoria interna a la hora de dar de alta agentes.

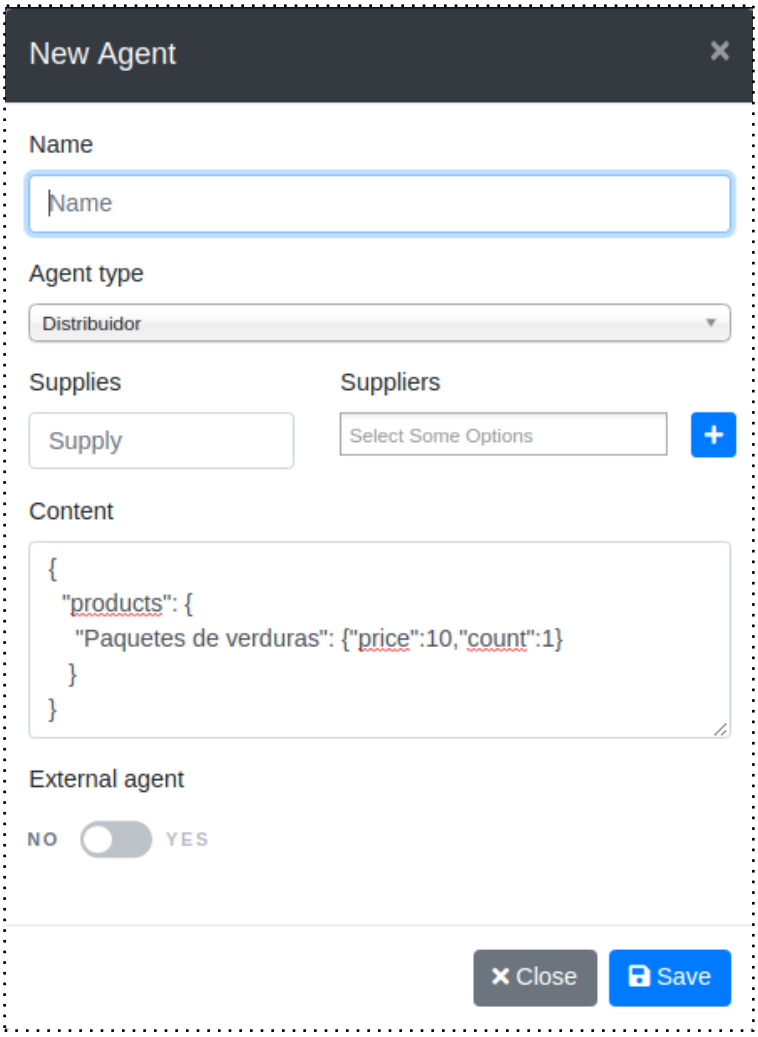

Ilustración 15. Ejemplo de uso del defaultStore.

Hot spots de desarrollo

Los hot spots de desarrollo, son utilizados para indicar al framework el *enviroment* (o entorno) que debe de levantar, pudiendo este manejar un *environment* local y otro en la nube. Esto se puede realizar a través del archivo *config.json* del framework. En este podemos observar la clave *ENVIRONMENT*, la cual se puede establecer con valores *aws* o *local*.

Entorno Local

Cuando se establece que el entorno de trabajo es *local*, hace que el framework ejecute los servicios de *Lambda, Api Gateway* y *DynamoDB* de forma local, para poder realizar la simulación. De esta forma, toda la simulación es llevada adelante en el equipo de forma local, sin la necesidad de invocar algún servicio remoto. Esto se dá gracias a que se levanta un servidor local en el puerto 3000 que permite invocar las funciones a través del dominio que se le haya definido.

Entorno AWS

Por otra parte, cuando el valor de la clave *ENVIRONMENT* es *aws*, el framework utiliza los servicios de la nube de Amazon Web Services. Además de *Lambda* y *DynamoDB*, en AWS se utiliza los servicios de *Api Gateway* el cual provee un punto de ingreso a las funciones *Lambda* y *CloudWatch* para registrar la salida durante la ejecución de estas funciones.

Es fundamental que la opción *ENVIRONMENT* esté establecida correctamente cuando se realice el deploy de las funciones a Amazon, así se levantan las configuraciones necesarias para la utilización de los servicios en AWS, sino el framework no podrá tener acceso a los servicios, lo que imposibilita llevar a cabo la simulación.

Hot spots configurables por herramienta gráfica

Estos hot spots tienen la tarea de indicar a la herramienta gráfica cuál es la fuente de datos para los distintos entornos, *aws* o *local*. Para esto, es necesario que en el archivo *index.js* estén establecidos los valores de las variables *API_URL_LOCAL* y *API_URL_AWS*. Por defecto, *API_URL_LOCAL* toma el valor de *http://localhost:3000*, que es la URL por defecto donde se levanta el plugin *serverless offline*. En cuanto a la variable *API_URL_AWS* por defecto está sin valor asignado y representa la URL base dada por AWS para las funciones desplegadas en *Lambda*. Por lo tanto, si desea utilizar la herramienta gráfica, de forma completa, es indispensable configurar estas dos variables.

Hook methods

Métodos gancho (o hook methods) son los elementos públicos del framework, con los cuales el usuario puede construir métodos que, una vez definidos, conformarán la distinción entre varias simulaciones.

En este framework se pueden encontrar los siguientes hook methods:

- **getStore():** Permite obtener la memoria interna de cada agente.
- **setStore(aStore):** Esta función se utiliza para establecer la memoria interna de cada agente.
- **queue(aMessage):** Encolar un mensaje, para que luego el framework se encargue de enviarlo.
- **getConversation(conversationId):** Dado un id de conversación, esta función permite obtener la conversación completa a la que pertenece dicho identificador.
- **haveSuppliers():** Retorna un valor *booleano* que permite saber si el agente tiene o no proveedores.

.

Editor de cadenas

Con el editor de cadenas, se permite que un usuario, no necesariamente el mismo que programó las distintas clases de agentes posibles, pueda crear cadenas y realizar tantas simulaciones sobre la misma como lo crea necesario.

Para esto, esta herramienta gráfica y simple de utilizar permite que un usuario, sin conocimientos de programación, establezca sus agentes, los configure como crea adecuado y luego los utilice en distintas simulaciones. Esto se lleva adelante creando las distintas cadenas que se necesite. Una vez creadas estas, el usuario deberá de establecer los agentes participantes de las simulaciones. Por lo tanto, para cada agente se deberá de establecer un nombre, además se podrá indicar con qué otros agentes se relaciona, el contenido que se almacenará en la memoria del agente y la clase a la que pertenece. Cabe destacar que cada clase que el usuario programador realizó, van a aparecer como posibles opciones para el agente que se está creando.

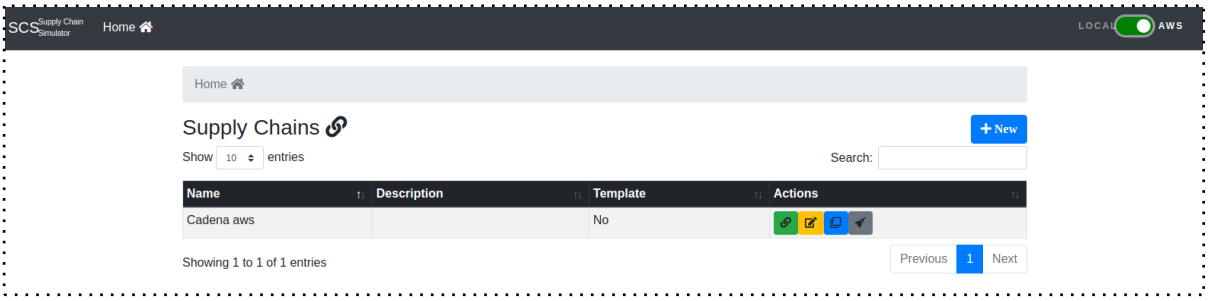

Ilustración 16. Herramienta gráfica.

A estas alturas ya se tiene al menos una cadena con un conjunto de agentes que pueden interactuar entre sí. Para que esto suceda, desde la misma herramienta se puede tomar un agente en particular y enviarle un *CallForProposal* para ver cómo reacciona el agente ante este mensaje.

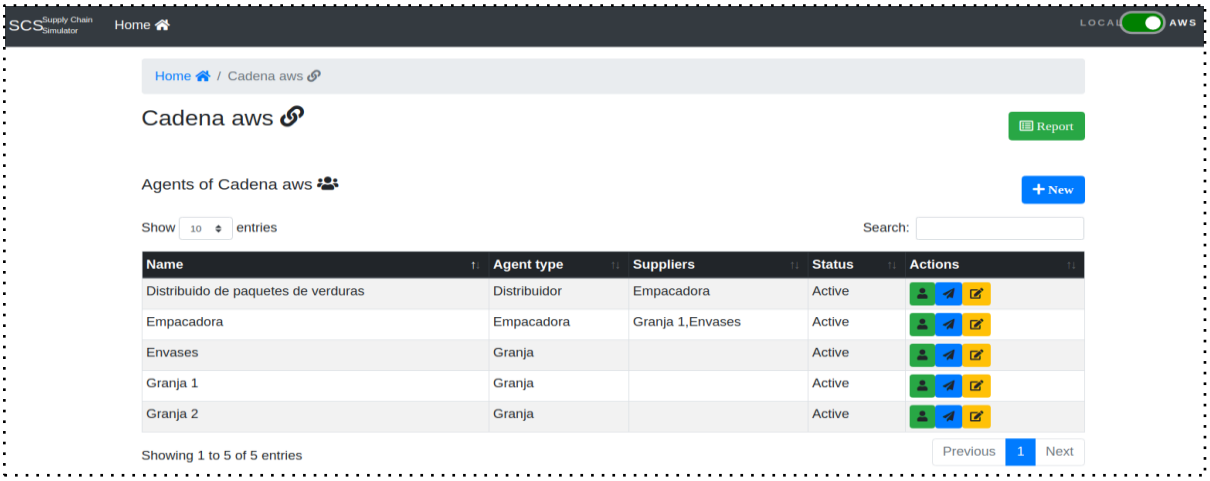

Ilustración 17. Agentes de una cadena.

Una vez finalizada la simulación con el agente elegido, viendo el mismo, se pueden observar los distintos mensajes que se intercambiaron en la simulación realizada. A su vez,

si este agente, en dicha simulación, interactúa con algún otro agente de la cadena, accediendo a este último se pueden observar, de la misma manera, los mensajes que implican para este.

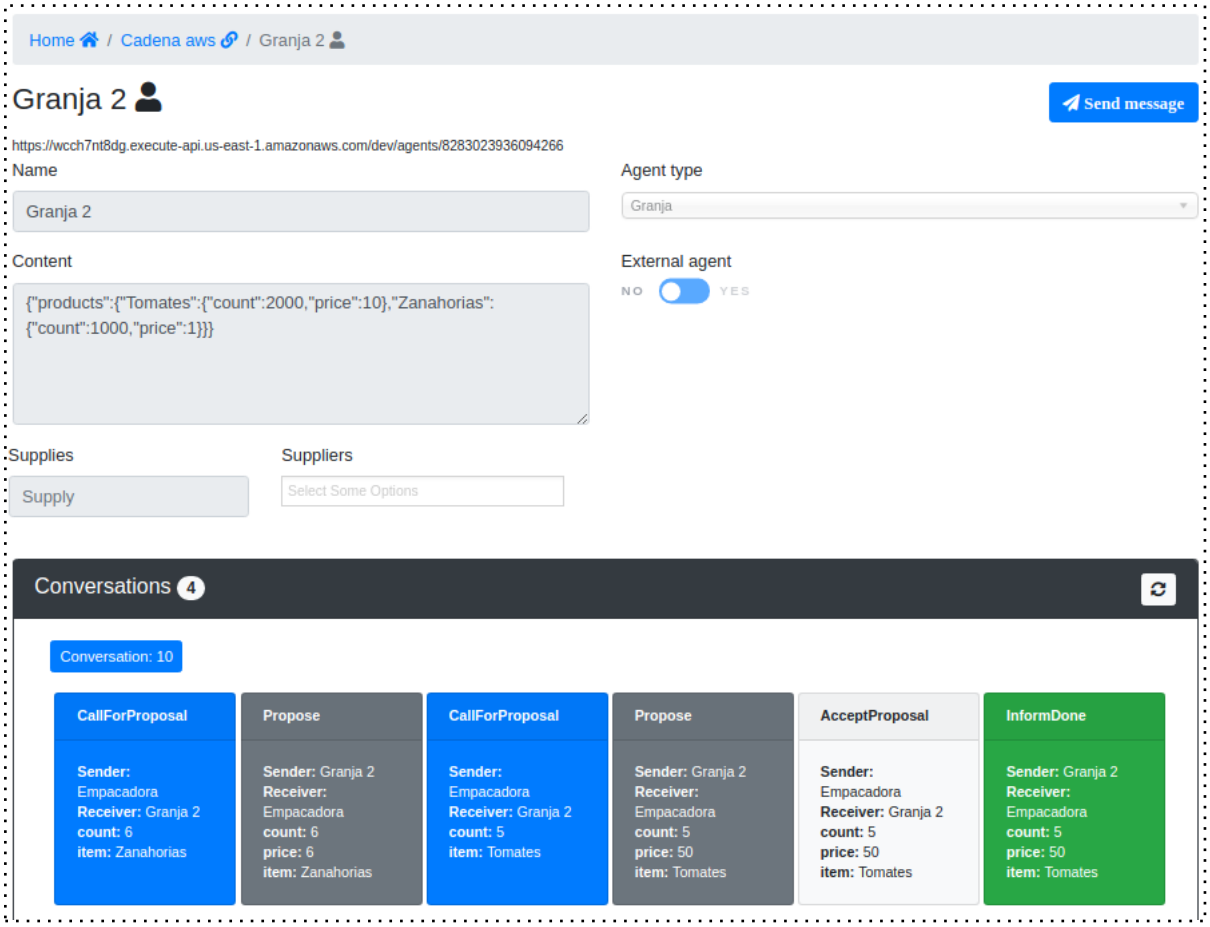

Ilustración 18. Detalle del agente Granja.

Similar al framework, que puede ejecutarse de forma *local* o en AWS, la herramienta gráfica permite al usuario elegir con qué entorno trabajar, si con uno que se encuentra de forma remota es decir en la nube de AWS o si con un servidor local. Esto se logra con el switch que se puede observar en la esquina superior derecha. Con ese simple switch se puede cambiar entre los distintos ambientes posibles de backend.

Detalles de implementación

El framework SCS está desarrollado en *JavaScript* utilizando el entorno de ejecución *Node.js* en su versión 8.10.

Por otra parte, utiliza como base el framework *Serverless*, el cual provee una estructura de desarrollo fundamental para poder llevar adelante este framework. Entre sus principales características se encuentra la posibilidad de elegir entre los distintos proveedores de

serverless en la nube, como Amazon Web Services[57], Microsoft Azure[58], Google Cloud[59], entre otros, permitiendo deployar las funciones con solo unos simples cambios en las distintas plataforma existentes y soportadas por el framework *Serverless*.

Uno de esos pocos cambios tiene que ver con la configuración de credenciales, permitiendo establecer las credenciales de los distintos proveedores de serverless en la nube, condición necesaria para poder realizar el despliegue de las funciones en el proveedor.

Además, *Serverless* permite establecer los eventos, es decir, los motivos por los cuales se desencadenan las ejecuciones de las funciones. Permitiendo establecer distintos eventos para una misma función, entre estos se encuentran eventos *HTTP*, almacenamiento de datos, notificaciones, etc.

En esta tesina, el principal evento es de tipo *HTTP*, es decir, se invoca una función a través de un recurso *HTTP* que ingresa a través del servicio establecido en base al proveedor elegido, por ejemplo, *API Gateway* en el caso de AWS.

A su vez, permite por cada función, establecer los permisos y recursos accesibles por la misma. Esto quiere decir, que se puede indicar si la función tiene acceso a algún recurso en particular (*Api Gateway, S3, DynamoDB, CloudWatch*, etc en AWS por ejemplo) y las acciones que puede realizar con dicho recurso.

Por otra parte, está el *deploy*, el cual permite desplegar las funciones en la nube con los recursos y permisos establecidos en la configuración.

En cuanto al desarrollo e implementación, *Serverless* provee toda una estructura necesaria para la generación y ejecución de funciones, pero no del todo completa, si se decide montar un ambiente de AWS local. Para que esta asi lo sea, es necesario sumar dos plugins npm, *serverless-offline* y *serverless-dynamodb-local*.

El plugin *serverless-offline* permite simular los servicios de *Lamba* y *Api Gateway*. Es decir, que se encarga de levantar un servidor *HTTP* que expone, en este caso, la *API REST* con el acceso a las funciones.

Y con respecto al plugin *serverless-dynamodb-local* se encarga de simular de forma local el servicio *DynamoDB*, servicio clave en las simulaciones para almacenar toda la información de las mismas.

Con estos dos plugins, más el framework, se tiene todo lo necesario para desarrollar y probar de forma local las funciones realizadas.

[57] Sitio oficial AWS - https://aws.amazon.com/

[58] Sitio oficial Microsoft Azure - https://azure.microsoft.com/

[59] Sitio oficial Google Cloud - https://cloud.google.com/

Servicios utilizados

En esta tesina, se utiliza como proveedor de infraestructura en la nube a Amazon Web Services. Dentro de esta estructura se utilizan los servicios de *Lambda, API Gateway, DynamoDB, CloudWatch, IAM y S3*. En este apartado se van a desarrollar cada uno de ellos.

Lambda

Lambda es el servicios de funciones serverless de AWS. Solamente requiere que se cargue el código fuente de las funciones y este servicio se encarga de todo lo necesario para ejecutarlas y escalar el código con alta disponibilidad, sin la necesidad de tener un servidor ni un backend, evitando de esta forma la necesidad de administración de los mismos [Amazon[60]].

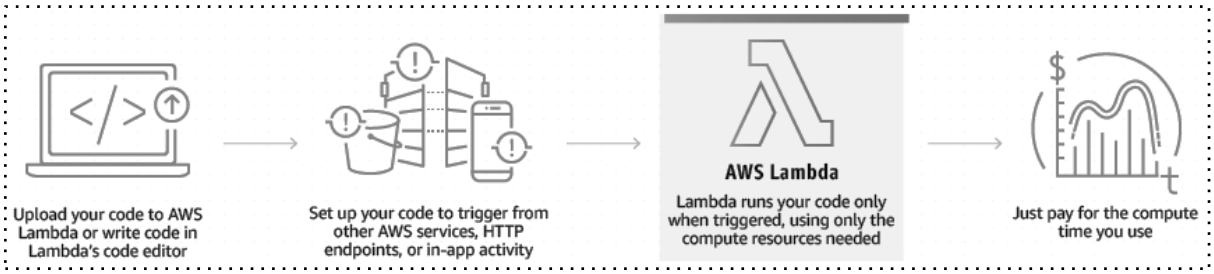

Ilustración 19. Funcionamiento de Amazon Lambda.

Además, brinda características como escalado continuo (el servicio escala automáticamente en respuesta a cada disparador), el costo del mismo depende del tiempo de ejecución de la función, es decir, si la función no se ejecuta, no se deberá abonar nada, y la omisión de un servidor en el cual ejecutar el código.

Si bien, estas características/beneficios que obtiene el usuario, tiene su beneficio en AWS, por ejemplo, el omitir un servidor donde ejecutar el código, es un beneficio interesante de abordar, ya que desliga al usuario del mantenimiento del servidor, siendo AWS el encargado de esto, evitando de esta forma que el usuario tenga servidores con fallos de seguridad que puedan comprometer al resto de la plataforma.

Algo similar sucede con los costos en la ejecución. AWS prefiere cobrar por el tiempo de uso que pueda generar una función, en lugar de cobrar por tener un servidor reservado al cual no se lo está utilizando y se puede estar aprovechando de mejor manera.

Por otro lado, las funciones *Lambda* pueden ejecutarse como el desencadenante de un evento realizado en otro de los servicios de Amazon Web Services, por ejemplo, un evento en *Api Gateway* puede desencadenar la ejecución de una función, modificaciones realizadas a objetos en *buckets* (cubetas) de *S3*, actualización de tablas en *DynamoDB*, entre otros.

[60] Sitio oficial Amazon Lambda - https://aws.amazon.com/lambda/

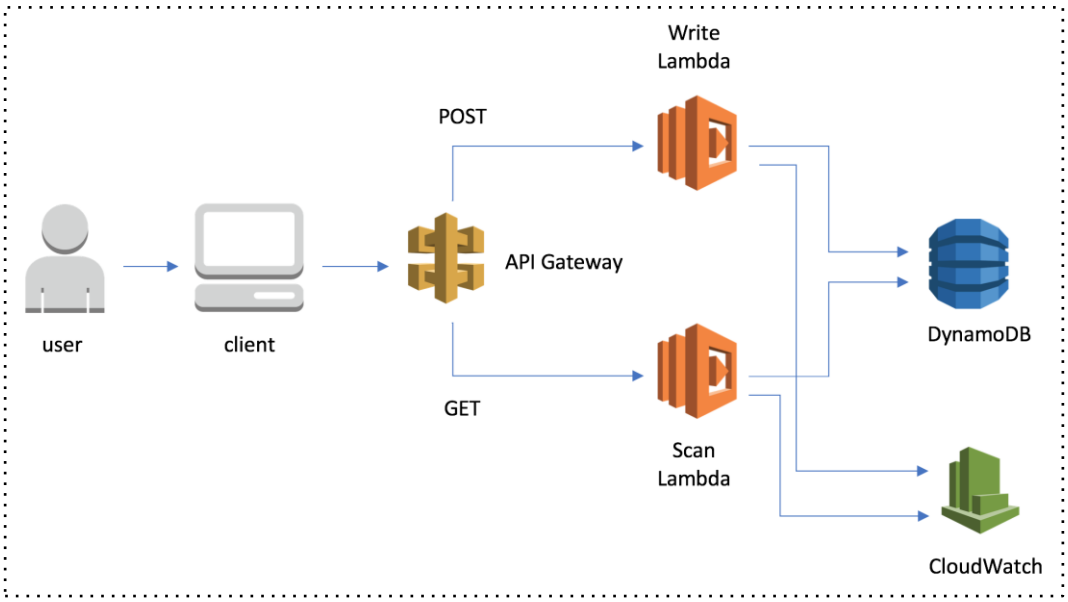

Ilustración 20. Ejemplo de cómo un evento de Api Gateway puede desencadenar la lectura o escritura sobre una base de datos de DynamoDB. Adicionalmente, cada ejecución de Lambda guarda logs en Amazon CloudWatch.

Las funciones de *Lambda "no tienen estado"* y no tienen ninguna afinidad con la infraestructura subyacente, por lo que puede lanzar rápidamente tantas copias de la función como resulten necesarias para ajustar la escala al índice de eventos entrantes.

Estas funciones pueden utilizarse como una ampliación de otro producto, es decir, se tiene un producto que se encarga de subir videos a *S3*, con *Lambda* se le puede agregar un valor extra, haciendo que cada video que se suba a *S3* pase por una función *Lambda* que se encargue de comprimir el mismo. De esta forma se hace un uso responsable de la capacidad de almacenamiento.

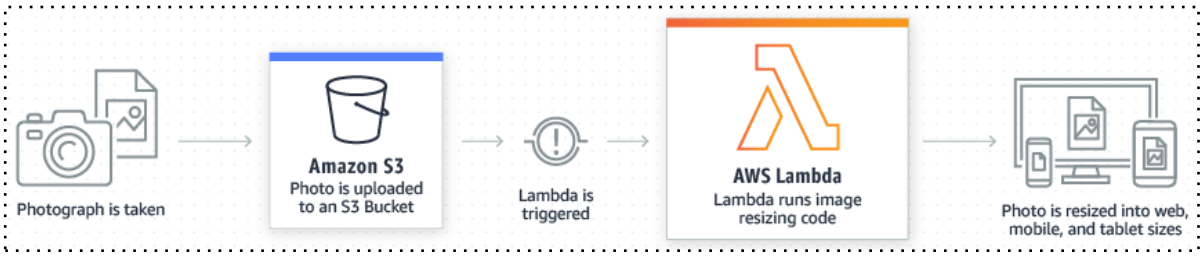

Ilustración 21. Otro ejemplo también puede ser utilizar Lambda para modificar el tamaño de fotos subidas a S3.

Otro punto clave tiene que ver con la tolerancia a errores. AWS mantiene la capacidad de cómputo en varias zonas de disponibilidad, en cada región, para ayudar a proteger el código fuente frente a fallos en equipos individuales o fallos en las instalaciones del centro de datos.

Además, se pueden coordinar varias funciones de *Lambda* para tareas complejas o largas, mediante la creación de flujos de trabajo con *Amazon Step Functions*[61]. *Step Functions* permite definir flujos de trabajo, capaces de activar diferentes funciones de *Lambda*, mediante el uso de pasos secuenciales, paralelos, bifurcados o con control de errores. Con

[61] Sitio oficial Amazon Step Functions - https://aws.amazon.com/step-functions/

Step Functions y *Lambda*, se pueden crear procesos complejos y de larga duración para aplicaciones y backends.

En cuanto a seguridad, a cada función *Lambda* se le pueden establecer permisos de acceso a los distintos recursos que pueda necesitar. Esto se logra con el servicio *IAM*, estableciendo roles de ejecución. Además, cada función *Lambda* ejecuta el código fuente dentro de una *VPC* (Virtual Private Cloud) de manera predeterminada, donde solo puede ver los recursos a los cuales se le dio acceso.

Por último, el usuario puede establecer la memoria que desea asignar a las funciones. Cuando hace esto, también está proporcionando una capacidad de CPU, ancho de banda de red y E/S de disco.

API Gateway

Este es el servicio de AWS que permite a los desarrolladores la creación, publicación, mantenimiento, monitoreo y protección de API a cualquier escala. Con tan solo unos pocos pasos, se pueden crear *API REST* y *API WebSocket* que actúen como "puerta delantera" para que las aplicaciones obtengan acceso a datos, lógica de negocio o funcionalidades desde sus servicios backend. El cual puede estar en alguno de sus otros servicios, como lo son *Amazon Elastic Compute Cloud*[62] *(Amazon EC2), AWS Lambda*, cualquier aplicación web o aplicaciones de comunicación en tiempo real [Amazon[63]].

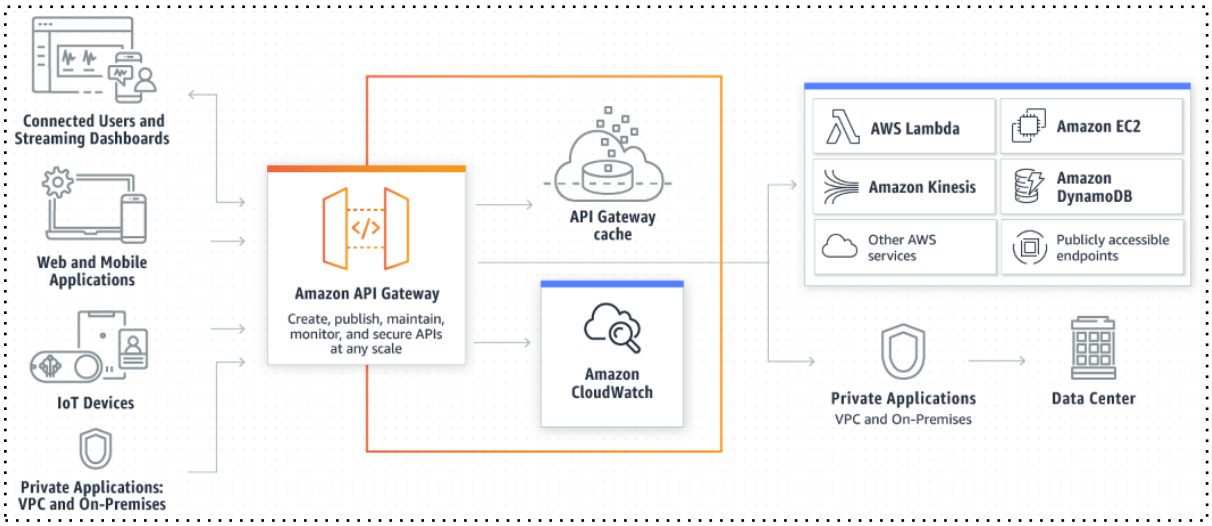

Ilustración 22. Flujo de trabajo de API Gateway.

Este servicio gestiona todas las tareas implicadas en la aceptación y el procesamiento de hasta cientos de miles de llamadas a API simultáneas, entre ellas, la administración del tráfico, el control de autorizaciones y acceso, el monitoreo y la administración de versiones de API.

[62] Sitio oficial Amazon EC2 - https://aws.amazon.com/ec2/

[63] Sitio oficial Amazon Api Gateway - https://aws.amazon.com/api-gateway/

En cuanto a los costos de su uso está directamente relacionado con la cantidad de solicitudes y el tamaño de respuesta que deba de procesar. Similar al servicio *Lambda*, si este no se utiliza no se deberá abonar nada.

Otra característica de *API Gateway* es que permite que se ejecuten varias versiones de la misma API de forma simultánea, lo que facilita la iteración, puesta a prueba y permitiendo la publicación de nuevas versiones con rapidez.

Por otro lado, el servicio permite agregar seguridad a la API a través de otros servicios como *Amazon Identity and Access Management (IAM)* y *Amazon Cognito*[64].

En caso de ya tener un servicio de autorización, *API Gateway* puede contribuir a verificar las solicitudes entrantes mediante la ejecución de un autorizador de *Lambda*.

Además, *API Gateway* permite generar un *SDK* de cliente para varias plataformas, que se pueden utilizar para poner a prueba las API nuevas, desde las aplicaciones, con rapidez y distribuir los *SDK* entre desarrolladores de terceros. Los SDK generados pueden administrar las claves de las API y las solicitudes de inicio de sesión mediante las credenciales de AWS. Los *SDK* están disponibles para Java, JavaScript, Java para Android, Objective-C o Swift para iOS y Ruby. Para generar el *SDK*, utilizando la *CLI (Command Line Interface* o interfaz de línea de comando*)* de AWS es necesario ejecutar el comando *get-sdk*.

Y por último, mencionar dos características del servicio, que tienen que ver con el monitoreo y el rendimiento. En cuanto al rendimiento, el servicio permite proporcionar bajas latencias para las solicitudes y respuestas de API, esto es logrado gracias a la red global que AWS tienen montada. Sumado que permite cachear la salida de una llamada, permitiendo que ante una misma consulta se retornen los datos de esta caché sin tener que ir al servicio backend. Y lo que refiere al monitoreo, este permite obtener métricas del rendimiento del servicio, pudiendo observar latencia de datos y tasas de errores, permitiendo detectar donde es necesario ajustar para obtener el rendimiento deseado.

DynamoDB

Es el servicio de base de datos no relacional de AWS. *DynamoDB* es una base de datos de claves-valor y documentos que ofrece rendimiento en milésimas de segundos a cualquier escala. Se trata de una base de datos multiregión, es decir, cuenta con réplicas de la misma en las distintas regiones que tiene Amazon Web Services, lo cual se logra gracias a que estas son multimaestro, lo que significa que cualquier integrante perteneciente al grupo de réplica, puede actualizar el resto de bases distribuidas en las distintas zonas de AWS [Amazon[65]].

[64] Sitio oficial Amazon Cognito - https://aws.amazon.com/cognito/

[65] Sitio oficial Amazon DynamoDB - https://aws.amazon.com/dynamodb/

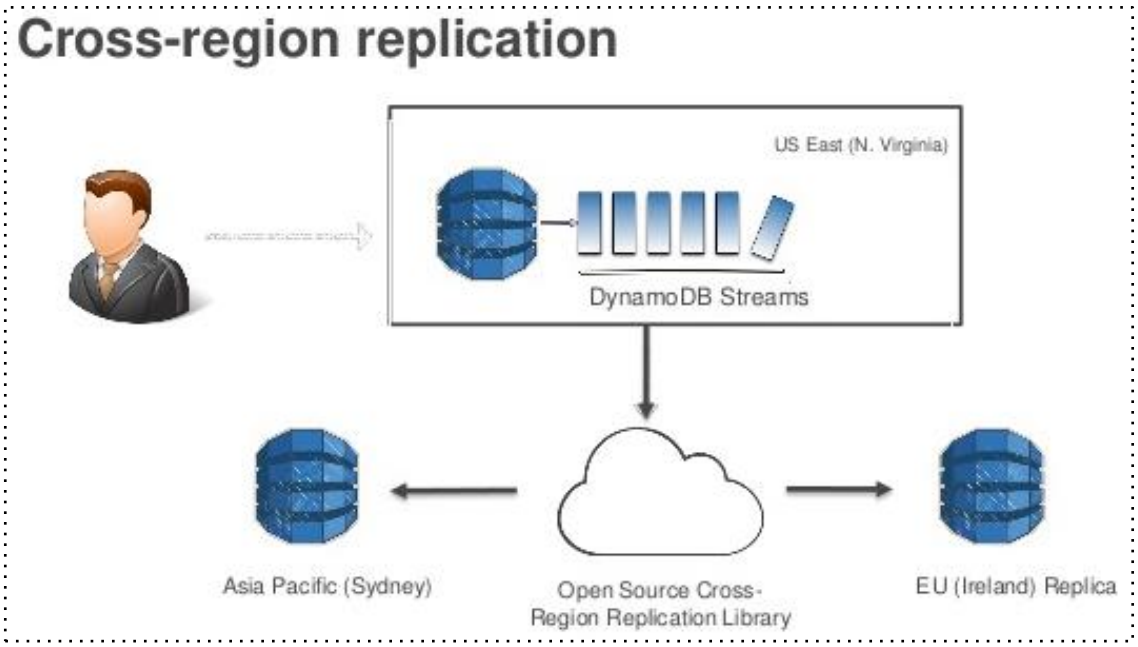

Ilustración 23. Replicación de datos multiregión de Amazon DynamoDB.

Además, es completamente administrada, esto quiere decir que AWS administra el software de la base de datos y el aprovisionamiento del hardware necesario para ejecutarlo. Esto permite que *DynamoDB* aumente su capacidad de forma automática, en base a la demanda de carga de trabajo, y crear particiones y subparticiones de los datos, a medida que aumenta el tamaño de la tabla. *DynamoDB* realiza una réplica sincronizada de los datos entre centros de múltiples regiones de AWS, esto proporciona un alto nivel de disponibilidad y durabilidad de los datos.

Por otro lado provee la generación de copia de seguridad y restauraciones de los mismos, y almacenamiento de datos en memoria caché para agilizar la respuesta de datos que son consultados frecuentemente.

En lo que refiere a seguridad, es necesario tener credenciales para poder utilizar el servicio. Las mismas pueden establecer a qué tablas se puede ingresar y con qué accesos lo pueden hacer.

Como gran parte de sus servicios, no hay servidores que aprovisionar, parchear o administrar, y no hay software que instalar, mantener o utilizar.

CloudWatch

Amazon CloudWatch es el servicio de monitoreo y administración creado para desarrolladores, operadores de sistemas, ingenieros de fiabilidad de sitio y gerentes de TI. *CloudWatch* ofrece datos e información procesable para monitorear las aplicaciones, comprender cambios de rendimiento que afectan a todo el sistema y tomar acciones, optimizar el uso de recursos y lograr una vista unificada del estado de las operaciones [Amazon[66]].

[66] Sitio oficial Amazon CloudWatch - https://aws.amazon.com/cloudwatch/

Este servicio recopila datos de monitoreo y operaciones en formato de registros, métricas y eventos, lo que ofrece una vista unificada de los recursos, aplicaciones y servicios de AWS que se ejecutan en servidores locales y en la nube de Amazon.

Se puede utilizar *CloudWatch* para definir alarmas de alta resolución, ver registros y métricas lado a lado, tomar acciones automatizadas, resolver errores y descubrir información para optimizar las aplicaciones y asegurar que se estén ejecutando sin problemas.

Como el resto de los servicios no hay tarifas mínimas, y sólo se paga por lo que se usa.

Ilustración 24. Funcionamiento de Amazon CloudWatch.

CloudWatch es un potente servicio que permite saber todo lo que sucedió entre los distintos micro servicios que se puede llegar a tener. Esto permite superar el desafío de monitorear desde aplicaciones y sistemas individuales aislados (servidor, red, base de datos, etc) hasta una pila completa (aplicaciones, infraestructura y servicios). En base a este monitoreo es posible establecer alarmas, registros y datos de eventos para tomar acciones automatizadas.

En cuanto a la recopilación de datos, este servicio permite recopilar y almacenar registros de los distintos recursos, aplicaciones y servicios casi en tiempo real.

Una vez recabados estos datos, se pueden obtener métricas integradas y métricas personalizables. Las integradas son métricas predeterminadas que ofrece AWS de sus distintos servicios. Las personalizadas, son aquellas que el usuario puede establecer, permitiendo obtener métricas de estudios que el usuario crea necesario.

Por otro lado, el monitoreo permite establecer paneles y alarmas. Los paneles permiten crear gráficos reutilizables y ver las aplicaciones y los recursos en una vista unificada. De esta forma, se puede visualizar en un gráfico los datos de registros y métricas lado a lado en un único panel. Un caso común para esto es un gráfico de utilización de CPU y memoria del servicio. En cuanto a las alarmas, permiten a través de un umbral de métricas definir acciones a realizar. Por ejemplo, activar una alarma cuando un servicio supera un uso preestablecido como asi tambien, notificar si se tiene instancias de servicios sin utilizar, o dar un paso más y cerrar las instancias sin uso.

La etapa de acción, es qué medida tomar ante una alarma en relación con una métrica clave. Una opción es automatizar el *auto scaling*, ante una métrica que se cumple, la alarma puede desencadenar en ejecutar una acción de *auto scaling*. Por ejemplo, se puede configurar un flujo de trabajo de *auto scaling* para añadir o eliminar instancias *EC2* en función de métricas de uso de CPU y optimizar los costos de los recursos.

Luego, cuenta con una etapa de análisis, esta etapa tiene varios servicios pertenecientes a *CloudWatch* que permiten realizar operaciones con las métricas. El servicio permite monitorear tendencias y estacionalidad con 15 meses de datos de métricas (almacenamiento y retención). Estos datos permiten realizar un análisis histórico permitiendo ajustar el uso de los recursos. Además, puede generar operaciones personalizadas sobre las métricas obtenidas con el servicio *Metric Math* de *Amazon CloudWatch*.

Amazon Cloudwatch Logs Insights permite aplicar inteligencia procesable a los registros para abordar problemas operativos, sin necesidad de aprovisionar los servidores o administrar el software.

Y por último, la etapa de conformidad y seguridad, *CloudWatch* se integra con el servicio *IAM* de manera que puede controlar qué usuarios y recursos tienen permiso para obtener acceso a sus datos y de qué manera lo hacen.

IAM

Amazon Identity and Access Management (IAM) permite administrar el acceso a los servicios y recursos de AWS de manera segura. Con *IAM*, se puede crear y administrar usuarios y grupos, así como utilizar permisos para conceder o denegar el acceso de estos a los recursos de AWS [Amazon[67]].

[67] Sitio oficial Amazon IAM - https://aws.amazon.com/iam/

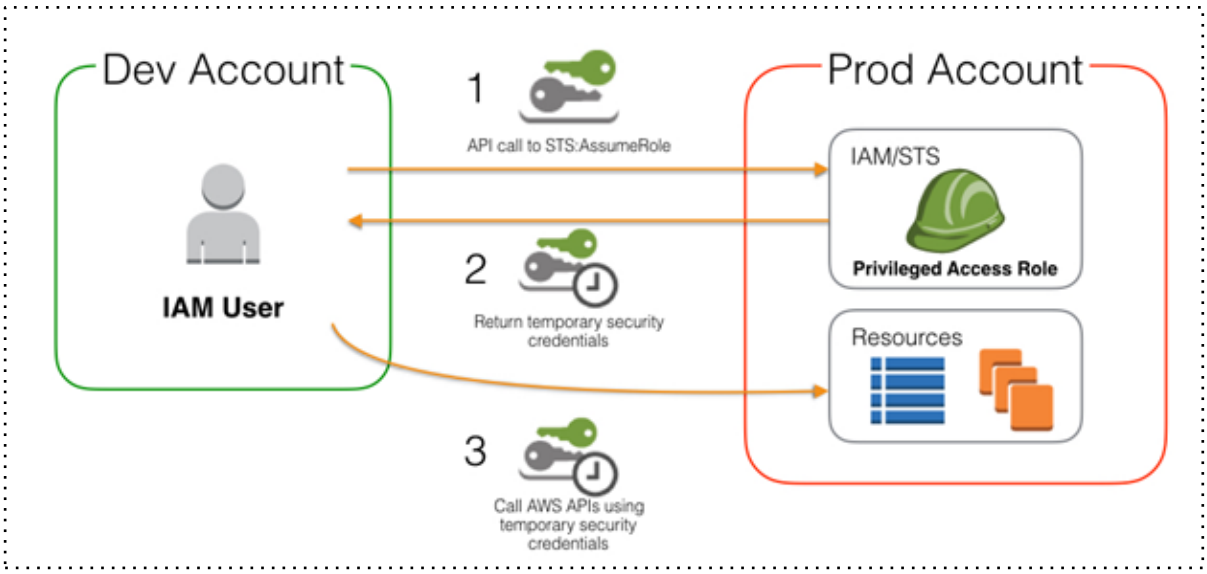

Ilustración 25. Posibles usos de Amazon IAM.

IAM es una característica de la cuenta de AWS que se ofrece sin cargos adicionales. Solo se cobrará por la utilización de los demás servicios de AWS por parte de sus usuarios.

IAM permite crear usuarios, a estos, asignarles sus propias credenciales de seguridad (es decir, claves de acceso, contraseñas y dispositivos de autenticación multifactor) o solicitar credenciales de seguridad temporales para permitirles obtener acceso a los recursos y los servicios de AWS. Además, es posible administrar los permisos para controlar qué operaciones puede realizar cada usuario.

Similar a usuarios, *IAM* permite crear funciones y administrar permisos para controlar qué operaciones pueden llevar a cabo la entidad o el servicio de AWS que asume la función.

Además, permite el manejo de usuarios federados, estos son usuarios que se administran fuera de AWS en un directorio corporativo, pero a los que se les concede acceso a la cuenta de AWS mediante credenciales de seguridad temporales. La diferencia entre los usuarios *IAM* y los federados, es que los usuarios *IAM* se crean dentro de AWS y los federados no.

S3

Amazon Simple Storage Service (Amazon S3) es el servicio de almacenamiento que permite guardar y recuperar cualquier volumen de datos, desde cualquier ubicación. Siendo este, uno de los servicios más reconocidos de AWS [Amazon[68]].

S3 es un servicio de almacenamiento de objetos que ofrece escalabilidad, disponibilidad de datos, seguridad y rendimiento. Esto significa que clientes de todos los tamaños y sectores pueden utilizarlo para almacenar y proteger cualquier cantidad de datos para diversos casos de uso, como sitios web, aplicaciones móviles, procesos de copia de seguridad y restauración, operaciones de archivado, aplicaciones empresariales, dispositivos IoT y

[68] Sitio oficial Amazon S3 - https://aws.amazon.com/s3/

análisis de Big Data. *Amazon S3* proporciona características de administración fáciles de utilizar que permiten organizar los datos y configurar sofisticados controles de acceso.

El servicio posibilita disminuir los costos de almacenamiento utilizando distintas clases. Con esto se busca lograr que los datos que más se frecuentan, se encuentren almacenados en dispositivos de altas prestaciones, mientras que aquellos que son menos frecuentados se almacenen en dispositivos con bajas prestaciones pero más económicos.

Además, el servicio cuenta con una herramienta de análisis de clases de almacenamiento de *S3*, para detectar qué datos se deben mover a una clase de almacenamiento más barata, en función de los patrones de acceso, así como configurar una política de ciclo de vida de *S3* para llevar a cabo la transferencia. Por otra parte, permite incrementar o reducir los recursos de almacenamiento para satisfacer la demanda fluctuante, sin necesidad de inversiones iniciales.

Amazon S3 está diseñado para ofrecer una durabilidad de los datos del 99,999999999%, ya que crea y almacena automáticamente copias de todos los objetos de *S3* en varios sistemas, esto quiere decir que realiza copia de los datos en las distintas zonas de disponibilidad de AWS, por ende, los datos están disponibles cuando se necesitan y protegidos frente a errores y amenazas.

Ilustración 26. Comparación de Amazon S3 con Amazon Glacier, Amazon EBS y Amazon EFS.

Capítulo 7

Partes de la plataforma

Ambientes de ejecución

Este framework cuenta con la posibilidad de trabajar en dos ambientes de ejecución, uno de ellos es en la nube (por ejemplo, Amazon Web Services o sus siglas AWS, Google Cloud, Microsoft Azure, etc) y el otro permite la ejecución local, simulando los servicios de la nube en un servidor local.

Ambiente de ejecución en la nube

En este ambiente toda la ejecución ocurre en la nube, utilizando las distintas ventajas que ésta provee, como costos, disponibilidad y escalabilidad. Como ejemplo, en esta tesina se utilizará Amazon Web Services. En AWS, se utilizan los servicios de *Api Gateway* para poder publicar una API REST, esto nos permite que cada agente sea un recurso particular el cual puede ser invocado realizando un llamado a la URI que lo identifica. Luego, cada llamado que ingresa al *Api Gateway* desencadena una invocación de una función *Lambda* la cual se encarga de procesar el pedido y responder a ese llamado.

El framework hace uso del servicio de *DynamoDB*, una base de datos NoSQL la cual nos permite guardar los datos de los distintos agentes en formato JSON. Y por último, la ejecución de cada función *Lambda* registra en *CloudWatch* (el servicio de monitoreo de AWS) lo que fue sucediendo en dicha ejecución. De este servicio, se puede ampliar el uso actual, utilizando la parte de monitoreo para obtener métricas de las distintas ejecuciones.

Con solo enviar un mensaje a un agente de la cadena se inicia una simulación, la cual puede desencadenar en el envío de un conjunto de mensajes entre los distintos agentes para poder cumplir con el pedido solicitado. Tanto el manejo de agentes y cadenas, como el intercambio de mensajes se hace a través de la ejecución de funciones *Lambda*, las cuales llevan adelante la simulación de las cadenas de suministros.

Ambiente de ejecución local

En este ambiente, toda la ejecución ocurre de forma local en el equipo en el cual se ejecute el servidor. El mismo busca emular al ambiente en AWS, es decir, se puede realizar lo mismo que en la nube, solo que la ejecución y almacenamiento de los datos se realiza de forma local.

Este tipo de ejecución es ideal para la etapa de desarrollo, dado que permite crear nuevas funcionalidades y probarlas sin la necesidad de realizar un despliegue en AWS. Para que esto pueda realizarse, se utilizan dos plugins de npm[69] (npm es uno de los sistemas de gestión de paquetes para JavaScript, en este caso a través de Node.js[70]), *serverless-offline* y *serverless-dynamodb-local*. El primero permite instanciar un servidor para poder realizar la simulación de forma local. Mientras que el plugin *serverless-dynamodb-local* permite levantar *DynamoDB* localmente.

Framework y herramientas para el desarrollo y despliegue de agentes

SCS permite al usuario generar distintas simulaciones de cadenas, implementando distintos comportamientos para sus agentes. Para esto, solo se necesita extender a la clase *Agente*, implementar la lógica deseada y por último asignar esta a los agentes deseados.

Para la implementación del comportamiento del agente se tiene el backend (parte de la aplicación que engloba la lógica de la misma y que no es accesible directamente por el usuario final del sistema), en el cual se puede extender la clase *Agente*.

Además del backend, el framework cuenta con un frontend (parte de la aplicación que complementa al backend, es decir donde el usuario final tiene interacción directa con el sistema) el cual permite manejar la gestión de cadenas y agentes.

Los sistemas creados a partir de *SCS* también cuentan con la posibilidad de clonar cadenas existentes, para facilitar la creación de cadenas similares a estas, en la cual se desea aplicar modificaciones sobre algún agente de la cadena y ver cómo influye esto en la simulación. Por último, existe la facilidad de poder limpiar todas las conversaciones de una cadena, así el usuario no tiene que recorrer todos los agentes uno por uno eliminando todas las conversaciones realizadas por el mismo.

A la hora de crear una nueva clase de agente, es necesario ejecutar el script *createAgentSubclass* el cual sirve de guía para crear una nueva clase. Una vez creada la misma, en la carpeta del proyecto *src/subclass_agent* se encuentra la clase creada. Esta clase contiene los métodos que se pueden implementar. Uno puede optar por no implementar ningún método, dando como resultado un agente sin la posibilidad de interactuar.

En los métodos a implementar se encuentra la definición del producto que maneja el agente y los distintos tipos de mensajes que puede recibir el mismo. Es fundamental definir el modelo del agente, para poder saber si un agente puede o no cumplir con un pedido. En caso de poder cumplirlo, es necesario saber qué materiales puede llegar a necesitar para poder proveer el producto solicitado.

[69] Sitio oficial npm - https://www.npmjs.com/

[70] Node.js es un entorno de ejecución multiplataforma para JavaScript.

En cuanto a los mensajes, se encuentran los distintos métodos para poder responder a un *CallForProposal*, *Propose*, *AcceptProposal*, *RejectProposal*, *InformDone*, *InformResult*, *Failure* y *Refuse*.

Por otra lado, hay una memoria de libre uso destinada al usuario del framework, la cual puede ser accedida a través del método de clase *getStore(),* y en caso de ser necesario modificar la misma, se realiza a través del método *setStore(store)*.

Esta memoria, en formato JSON, permite al usuario poder crear infinitas posibilidades de clases de agentes dado que puede agregar cualquier dato (en formato clave-valor) y recuperarlo en futuras iteraciones en las cuales se vea afectado el agente.

En las clases implementadas para esta tesina se utiliza esta memoria para guardar datos de los productos que el agente es capaz de producir, más precisamente, los materiales que necesita para producir los mismos, guardados bajo la clave *bom* (por sus siglas en inglés Bill Of Materials).

Ilustración 27. Imagen que representa a la memoria por defecto de un agente.

Y por último, las dos formas de ejecutar las simulaciones, local o en la nube.Local, levantar el servidor local (para esto es necesario ejecutar el comando *serverless offline start*), una vez iniciado el servidor ya es posible realizar simulaciones. En caso de querer desplegar el framework en AWS, es necesario ejecutar *serverless deploy*, esperar a que se comprima el código y luego este será subido a *S3*, desde donde será utilizado por *Lambda*. Una vez terminado este proceso ya se pueden realizar simulaciones en AWS.

Agente interactivo

El agente interactivo ISCA[71] o Interactive Supply Chain Agent, es una herramienta desarrollada por el profesor Dr. Alejandro Fernández, la cual permite actuar sobre distintos

[71] ISCA, agente interacitvo - http://casco.lifia.info.unlp.edu.ar/isca

nodos de una cadena, permitiendo al usuario tomar las decisiones. Esto es posible, eligiendo qué mensajes enviar y con qué mensajes responder al resto de los agentes.

Por otro lado, el agente interactivo permite observar la interacción de las cadenas generadas con el framework y una cadena externa. Dicha herramienta es simple de utilizar e intuitiva, con unos pocos pasos ya se puede crear un agente y centrarse en el envío de mensajes.

Lo primero que se debe realizar, luego de acceder a la herramienta, es crear un agente o elegir uno previamente creado. Una vez hecho esto, se puede ingresar a su *inbox*. En este, es posible observar un botón para crear un nuevo *CFP* y, en caso de ser un agente ya utilizado previamente se verá una lista de mensajes, pero si es un agente nuevo su *inbox* estará vacío. Si el agente cuenta con mensajes, al hacer click sobre ellos se puede observar su contenido. Además, al lado de cada mensaje se encuentra un conjunto de acciones disponibles, desde poder eliminar un mensaje, hasta las posibles respuestas a realizar sobre un mensaje recibido.

Ilustración 28. Inbox del agente 1 en el cliente interactivo.

En caso de querer realizar un nuevo pedido, hay que hacer click en *Compose a CFP*, ahí se mostrará una pantalla, en la cual se puede determinar el destinatario del mensaje a través del campo *URI of destination agent*, agregar un número de conversación para luego poder identificarla, un campo *In reply to,* utilizado para ingresar un identificador que refiera a un mensaje previo, un campo *Reply with,* para indicar un identificador con el que se espera que se responda a este *CFP* y el campo *content* del mensaje, el mismo es libre y es posible agregar tantas claves y valores como sean necesarias.

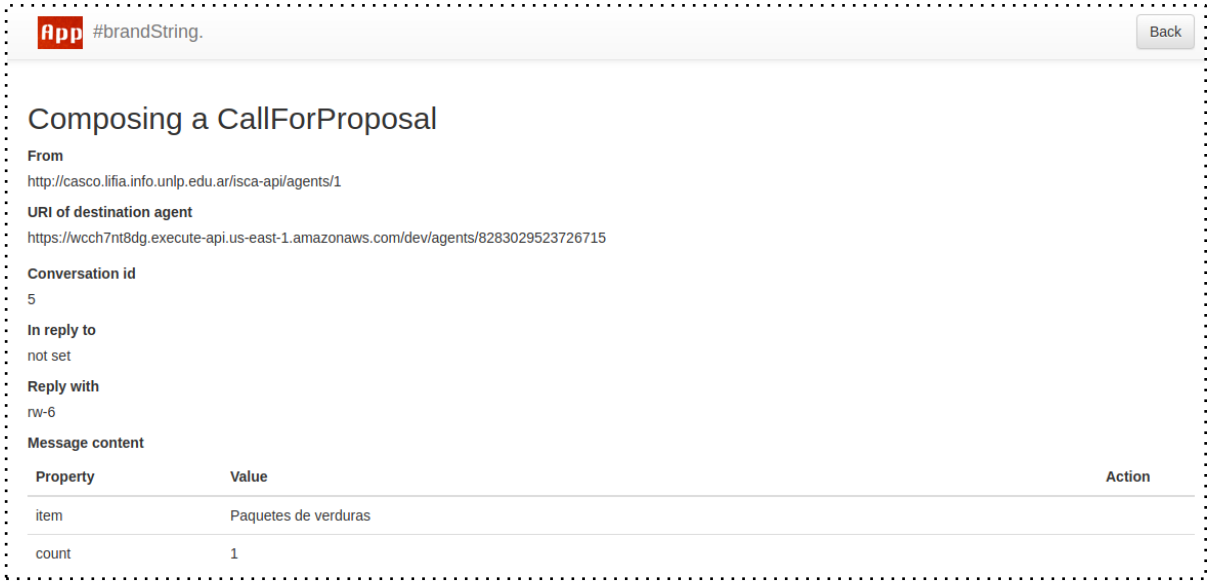

Ilustración 29. Composición de un CFP.

Una vez realizado el envío del CFP, se vuelve al *inbox* y se puede esperar por la respuesta a dicho mensaje. Esta respuesta, puede llegar de forma instantánea o puede tardar, esto depende de varios factores que tienen que ver con el tamaño de la cadena y la cantidad de mensajes que pueden llegar a necesitar intercambiar hasta poder armar una respuesta al pedido realizado.

Una vez que se obtiene la respuesta al *CFP,* es posible decidir qué se va a realizar con ese mensaje recibido, pudiendo tomar distintos caminos y observar qué es lo que va sucediendo en la simulación, observación que se realiza siempre desde el punto de vista de un nodo de la cadena de suministros.

Preparación del ambiente de ejecución

Para poder hacer uso del framework *SCS* ya sea en ambiente local o en la nube, previamente se debe tener instalado *npm* y luego realizar un conjunto de pasos:

1. Clonar el proyecto: esto se puede hacer con el siguiente comando:
    a. *git clone https://bitbucket.org/estebans16/simulation_supply_chain.git*
2. Una vez clonado el proyecto, se debe instalar el paquete serverless ejecutando:
    a. *npm install -g serverless*
3. Luego de esto, proceder a instalar el resto de dependencias del proyecto. Para esto ejecutar:
    a. *npm install*
4. Para finalizar, se debe abrir el proyecto y dentro del directorio *src/config* se encuentra un archivo llamado *config.json*. En este archivo se encuentra una variable llamada *ENVIRONMENT*, la cual puede tomar el valor de "*local*" o "*aws*". Esta variable es utilizada por el framework para saber qué configuración necesita cargar. Dado que la configuración para la gestión de la base de datos difiere en ambos ambientes. Es crucial setear esta de forma correcta dependiendo del ambiente

donde se desea ejecutar la simulación. SI se quiere ejecutar de forma local y la variable de *ENVIROMENT* tiene el valor "*aws*", las simulaciones no se llevarán a cabo, lo mismo pasa si en AWS, la variable *ENVIRONMENT* tiene el valor "*local*".

Ilustración 30. Imagen que representa a la variable de configuración del entorno de ejecución.

Una vez realizado estos pasos comunes para ambos ambientes, hay que seguir los siguientes, que son propios de cada uno. A continuación se describen para cada caso.

Ambiente local

En este ambiente, se va a levantar un servidor y una base de datos *DynamoDB* en forma local. Para esto se necesita tener instalado Java y configurado las credenciales de AWS.

Para verificar la instalación de Java en el equipo se puede ejecutar lo siguiente:

java -version

Esto debería de retornar algo similar a esto:

java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)

En caso de no ser así, se deberá instalar Java antes de poder continuar.

Una vez verificado la existencia de Java en el equipo, es necesario indicarle al framework qué credenciales se van a utilizar. En este caso, al ejecutar todo local, se puede *"engañar"* al framework, generando unas falsas credenciales. En caso de querer colocar credenciales válidas de AWS, ir al siguiente apartado, *Ambiente en la nube,* donde se explica paso a paso como generar las mismas. Luego continuar con el siguiente comando:

serverless config credentials --provider aws --key local --secret local

En este momento ya es posible instalar la base de datos *DynamoDB*, para eso ejecutar lo siguiente:

serverless dynamodb install

Y por último falta levantar el servidor, el cual se iniciará en *http://localhost:3000*. Para eso se debe ejecutar:

serverless offline start

Con esto ya se tiene un servidor funcionando de forma local. Recordar antes de iniciar el servidor, chequear la variable *ENVIRONMENT* del archivo *config.json* que contenga el valor "*local"* para así poder tener el entorno correctamente configurado.

Ahora, si se quiere tener el frontend de forma local, lo que hay que hacer es acceder a la carpeta *frontend* y buscar el archivo *index.html*, abrir este con cualquier navegador web y ya se tiene el frontend levantado en la máquina local.

Con el frontend iniciado localmente se tiene la posibilidad de usar el servidor previamente levantado. Para esto se debe asegurar que en la parte superior derecha figure la opción seleccionada como *LOCAL*.

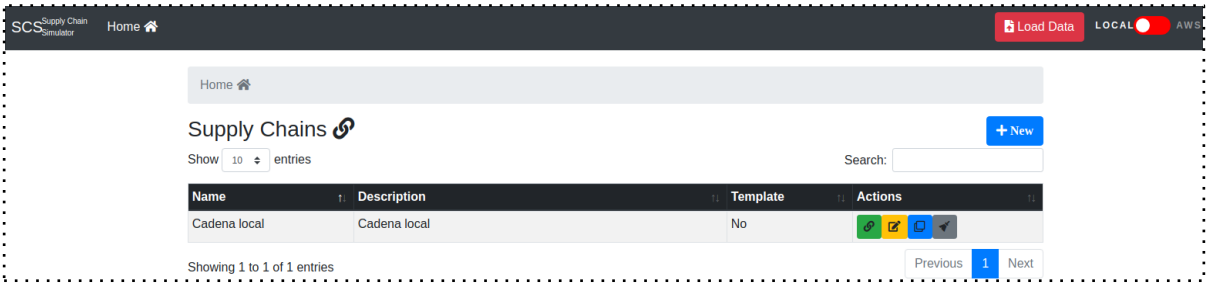

Ilustración 31. Imagen que indica la opción de utilizar el entorno de ejecución local.

Hay que dejar en claro, que cada vez que el servidor se detenga por cualquier motivo, los datos cargados hasta ese momento se perderán, es decir, si se vuelve a ejecutar el servidor, la base de datos *DynamoDB* local estará vacía. Por lo tanto, si se desea manejar un conjunto de datos preexistentes, se puede popular la base de datos realizando los siguientes pasos:

1. En la carpeta *src/seed* hay dos archivos en formato JSON, uno llamado *agents.json* que permite la carga de agentes y el otro *chains.json* que permite la carga de cadenas.
2. Una vez generados los datos de prueba, es necesario invocar a una función para que cargue los mismos. Previo a ejecutar este comando, el servidor local debe de estar corriendo y luego ejecutar:

   serverless invoke local --function loadData

   Esta función lee los archivos *agents.json* y *chains.json* y los carga en la base de datos *DynamoDB* que está en el servidor local, quedando así, los datos listos para ser utilizados por el sistema de modelado y simulación de cadenas de suministros. Una alternativa a este paso es utilizar el botón *Load Data* que se encuentra en la herramienta gráfica. El mismo tiene como objetivo cargar los datos que el usuario género en los *seed*.

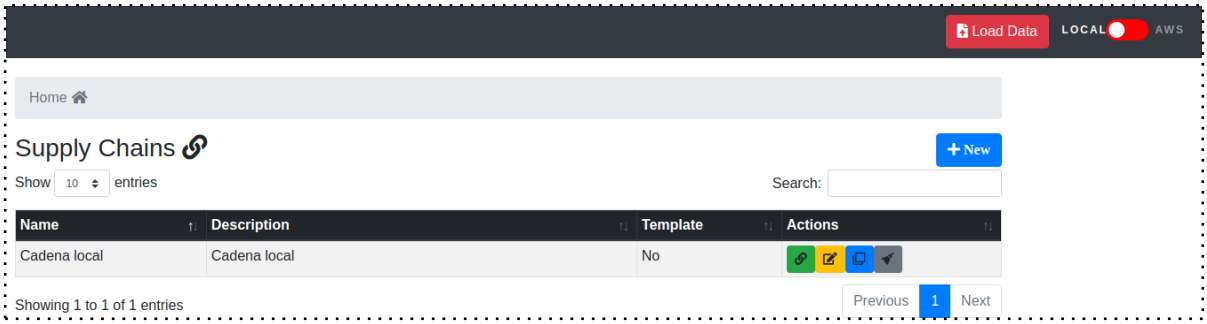

Ilustración 32. Herramienta gráfica con botón Load Data en ambiente local.

Ambiente en la nube

Para poder ejecutar las simulaciones en la nube (en este caso en AWS), es necesario desplegar el framework. Para esto, lo primero que se necesita es configurar las credenciales de AWS:

1. Abrir la consola de *IAM*[72] en AWS.
2. En el panel de navegación de la consola, elegir "*Usuarios*".
3. Seleccionar un nombre de usuario de *IAM.*
4. Elegir la pestaña "*Security credentials*" y a continuación, seleccionar "*Create access key*".
5. Para ver la nueva clave de acceso, elegir "*Show*". Las credenciales tendrán el siguiente aspecto:
    - ID de clave de acceso: *AKIAIOSFODNN7EXAMPLE*
    - Clave de acceso secreta: *wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY*
6. Para descargar el par de claves, elegir "*Download .csv file"*, se recomienda esta opción porque las claves serán necesarias más adelante. Por lo tanto, almacenar las claves en un lugar seguro. Si se pierden las mismas, se tendrán que volver a generar.

Por el momento, se tienen generadas las credenciales de AWS, ahora es necesario configurar el framework para que las utilice. Para esto ejecutar lo siguiente:

serverless config credentials --provider aws --key xx --secret yy

Donde se debe de reemplazar "*xx*" e "*yy*" por las claves generadas previamente. Una vez terminado esto, ya están generadas y configuradas las credenciales para poder ejecutar el sistema. Por último, falta realizar el despliegue del proyecto, para esto ejecutar:

serverless deploy

Al ejecutar ese comando, lo que hace el framework es crear todo lo necesario en AWS para el funcionamiento de las simulaciones. Esto implica crear las entradas a las funciones a

[72] Consola de gestión de Amazon IAM - https://console.aws.amazon.com/iam/home#/home

través del servicio *Api Gateway*, la creación de las funciones en *Lambda*, creación de roles en *IAM,* definiendo las políticas de accesos de las funciones a los distintos servicios de AWS, las tablas en *DynamoDB* y subir las distintas clases que se hayan desarrollado.

Además, se puede hacer lo mismo para el frontend, subiendo esta carpeta al servicio *S3* de Amazon. De esta manera queda corriendo todo en la nube de AWS. Recordar que para un correcto funcionamiento de la herramienta gráfica con el entorno en la nube es necesario tener bien configuradas la variable *ENVIRONMENT:"aws"* del archivo config.json y la variable *API_URL_AWS* con la URL base de las funciones desplegadas en *Lambda.*

Ejemplo de cadena de suministros

Descripción general

Como ejemplo en esta tesina, se va a modelar la cadena de suministros del producto "*Paquete de verduras procesadas para sopa*". Dicha cadena cuenta con 5 agentes intervinientes en las distintas etapas de esta, un agente "*Distribuidor de paquetes de verduras*" (para abreviar, "*Distribuidor*"), un agente "*Empacadora/procesadora de verduras para sopa*" ("*Empacadora*"), dos agentes "*Granjas*" y por último un agente "*Proveedor de envases*" ("*Envases*").

La distribución de estos agentes en la cadena, se da en su relación cliente/proveedor, de esta forma, el agente "*Distribuidor*" tiene como único proveedor al agente "*Empacadora*". El agente "*Empacadora*" tiene 3 agentes que lo proveen de los materiales necesarios para la creación de los paquetes de verduras para sopa, estos son los dos agentes "*Granja*" y el agente "*Envases*".

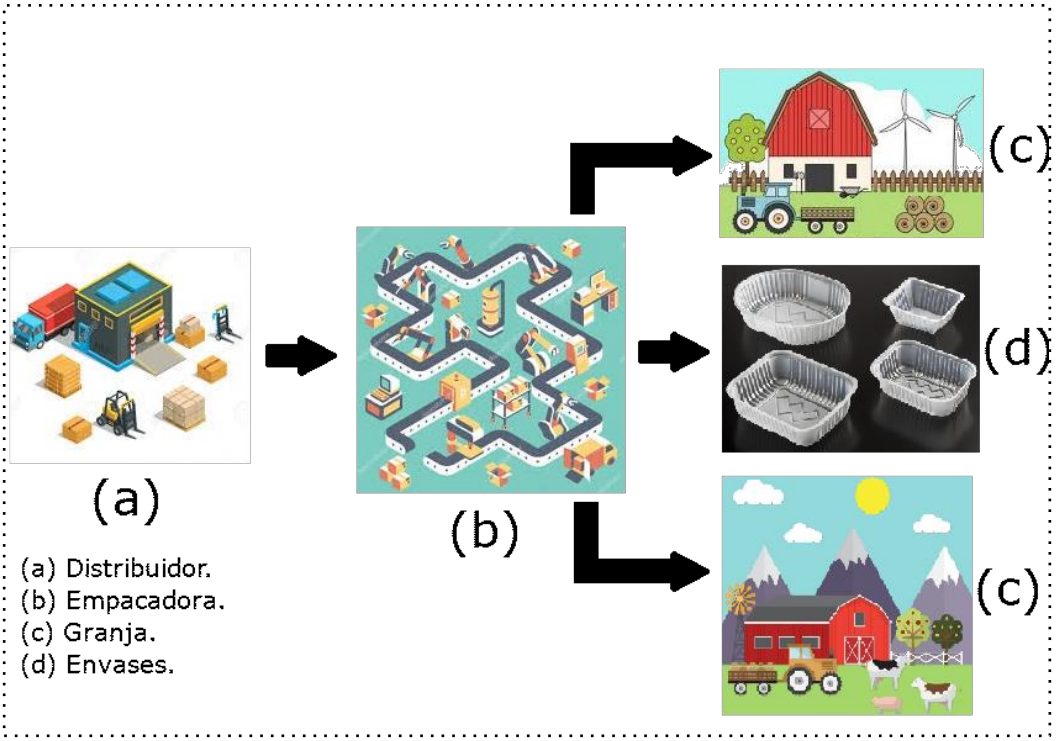

Ilustración 33. Modelo de cadena de suministros de ejemplo.

El flujo de ejecución de esta cadena comienza con el agente "*Distribuidor*", el cual posee un stock limitado de paquetes de verduras procesadas para sopa. Si un usuario le solicita a "*Distribuidor*" una cantidad de dichos paquetes se pueden dar 2 posibles casos:

- Si esta cantidad es menor o igual al stock del cual dispone este agente, entonces el agente le responderá al usuario con una propuesta, donde indicará el precio que deberá de pagar el usuario para obtener esta cantidad del producto.
- Sino, es decir, que la cantidad supera el stock que tiene "*Distribuidor*", entonces este deberá negociar con su proveedor de paquetes de verduras, para que le brinde los paquetes necesarios y así "*Distribuidor*" podrá cumplir con lo pedido por el usuario.

Luego del agente "*Distribuidor*", sigue el agente "*Empacadora*". Este agente es el encargado de procesar las verduras y luego colocarlas en envases descartables. Un paquete de verduras para sopa se compone a partir de zapallos, zanahorias, tomates y un envase descartable.

Para este ejemplo, este agente no cuenta con un stock de productos, entonces cada vez que le llega un pedido de paquetes de verduras procesadas para sopa, de su cliente "*Distribuidor*", "*Empacadora*" debe comenzar a negociar con sus proveedores de materiales y de esta forma obtener todo lo necesario para satisfacer el pedido hecho por su cliente. Así, "*Empacadora*" negocia con los agentes "*Granja*" y con el agente "*Envases*" para obtener las verduras y los envases que necesita.

Este agente cuenta con un mecanismo de toma de decisiones que se basa en aceptar aquella oferta que contenga el menor precio. De esta forma cuando sus dos proveedores "*Granja*" le hagan ofertas por el mismo material, por ejemplo zanahorias, el agente "*Empacadora*" aceptará la oferta más barata. Una vez que se alcanza las unidades

necesarias de cada material, para conformar la cantidad solicitada de paquetes de verduras, “*Empacadora*” le contesta el pedido a “*Distribuidor*” con una oferta propia para este pedido.

Los agentes “*Envases*” y “*Granja*” son similares, todos comparten el mismo comportamiento. Cuentan con stock que nunca decrece y no poseen proveedores, ya que son agentes productores, es decir son generadores de sus propios productos.

Clases de agentes

Clase Granja

La primera clase que se va a crear se llama *Granja*, la cual tiene un comportamiento sencillo. La misma, sabe qué tipo de productos puede satisfacer, como así también, el precio de cada uno de ellos y responde con una propuesta cada vez que tenga stock disponible, en caso de no tener, rechazará un pedido. Se utilizará esta clase para modelar, tanto a los agentes proveedores “*Granja*” como al agente proveedor “*Envases*”.

Para esto, en una terminal de comandos hay que situarse dentro de la carpeta raíz del proyecto y luego ejecutar el script *createAgentSubclass*:

En Linux

./createAgentSubclass.sh

```
➜  simulation_supply_chain git:(master) ✗ ./createAgentSubclass.sh
Enter the name of the subclass of Agent to create (e.g. AgentSubclass):
Granja
Create the file src/subclasses_agent/Granja.js
➜  simulation_supply_chain git:(master) ✗
```

Ilustración 34. Imagen que ilustra la ejecución del comando createAgentSubclass en una terminal Linux.

En Windows

.*createAgentSubclass.bat*

```
PS C:\Users\Matias\Desktop\simulation_supply_chain> .\createAgentSubclass.bat
Enter the name of the subclass of Agent to create (e.g. AgentSubclass): Granja
Create the file src\subclasses_agent\Granja.js
PS C:\Users\Matias\Desktop\simulation_supply_chain>
```

Ilustración 35. Imagen que ilustra la ejecución del comando createAgentSubclass en una terminal Windows.

Al ejecutar el script se pide ingresar el nombre de la clase que será creada, en este caso *Granja*. Una vez realizado esto, ir a un editor de código para poder definir el comportamiento de la clase generada, esta se encuentra dentro de *src/subclasses_agent*

```
'use strict';
var Agent = require('../agent');
let Message = require('../message');

class Granja extends Agent {

  constructor(dataAgent={}){
    dataAgent.agentType = 'Granja';
    super(dataAgent);
  }
  // Subclasses should redefine this method
  defaultStore(){
  }
  // Subclasses should redefine this method
  processCallForProposal(message){
  }
  // Subclasses should redefine this method
  processPropose(message){
  }
  // Subclasses should redefine this method
  processAcceptProposal(message){
  }
  // Subclasses should redefine this method
  processRejectProposal(message){
  }
  // Subclasses should redefine this method
  processRefuse(message){
  }
  // Subclasses should redefine this method
  processFailure(message){
  }
  // Subclasses should redefine this method
  processInformDone(message){
  }
  // Subclasses should redefine this method
  processInformResult(message){
  }
}

module.exports = Granja;
```

Ilustración 36. Imagen representativa de la clase Granja recién creada.

Como se puede observar, se tiene la posibilidad de definir el modelo por defecto de la clase y también cómo debe de actuar el agente ante el recibimiento de los distintos tipos de mensajes.

Lo primero que se va a definir es el modelo por defecto que tienen los agentes *Granjas* y *Envases*.

```
class Granja extends Agent {

  constructor(dataAgent={}){
    dataAgent.agentType = 'Granja';
    super(dataAgent);
  }
  // Subclasses should redefine this method
  defaultStore(){
    return {
      "products": {
        "Tomates": {"price": 10, "count": 200},
        "Zanahorias": {"price": 1, "count": 100},
      }
    }
  }
```

Ilustración 37. Imagen que ejemplifica el modelo por defecto de la memoria del agente.

El modelo por defecto es fundamental a la hora de crear agentes, dado que le da al usuario un ejemplo de cómo debe armar el modelo de la clase para que el mismo sea válido y la clase lo entienda al momento de realizar la simulación.

Con el modelo por defecto armado, cuando se cree un nuevo agente de tipo *Granja*, se puede observar en el *Content* que se muestra esta estructura por defecto para que el usuario la modifique a su gusto.

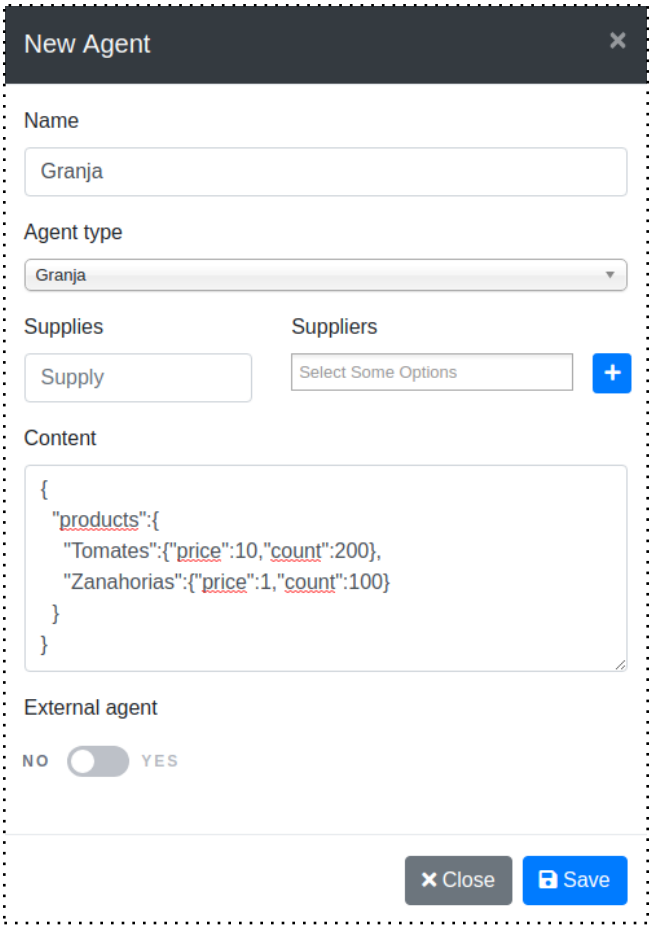

Ilustración 38. Ventana de creación de nuevo agente en el frontend del sistema.

Luego se debe definir cómo va a reaccionar el agente ante los distintos mensajes. Para esta clase es necesario definir cómo reaccionar ante la llegada de mensajes del tipo *CallForPorposal*, *AcceptProposal* y *RejectProposal*.

```
class Granja extends Agent {
  // Subclasses should redefine this method
  processCallForProposal(message){
    if (this.canProvide(message.content.item) && this.canSupplyRequest(message)){
      // armo propuesta
      let price = this.getProduct(message.content.item).price;
      let content = { price: price * parseInt(message.content.count),
                      item: message.content.item, count: message.content.count};
      let propose = message.asProposeTemplate();
      propose.content = content;
      this.queue(propose);
    }else{
      // Refuse
      let refuse = message.asRefuseTemplate();
      refuse.content = message.content;
      this.queue(refuse)
    }
  }
  // Subclasses should redefine this method
  processAcceptProposal(message){
    if (this.canProvide(message.content.item)){
      if (this.canSupplyRequest(message)){
        // inform-done
        let response = message.asInformDoneTemplate();
        response.content = message.content ;
        this.queue(response);
      }
    }else{
      // failure
      let response = message.asFailureTemplate();
      response.content = message.content ;
      this.queue(response);
    }
  }
  // Subclasses should redefine this method
  processRejectProposal(message){
    if (this.canProvide(message.content.item)){
      if (this.canSupplyRequest(message)){
        // armo propuesta
        let price = this.getProduct(message.content.item).price;
        let content = { price: price * parseInt(message.content.count),
                        item: message.content.item, count: message.content.count};
        let propose = message.asProposeTemplate();
        propose.content = content;
        this.queue(propose);
      }
    }
  }
  canSupplyRequest(message){
    let prod = this.getProduct(message.content['item']);
    return (parseInt(message.content["count"]) <= parseInt(prod.count));
  }
  getProduct(productName){
    return this.getStore().products[productName];
  }
}
```

Ilustración 39. Métodos de la clase Granja que definen su comportamiento ante la llegada de los distintos tipos de mensajes.

Tanto para el armado de esta clase, como para las demás, el comportamiento ante los diversos mensajes se desarrolló en base al FIPA Contract Net Interaction Protocol explicado en detalle en el capítulo 4.

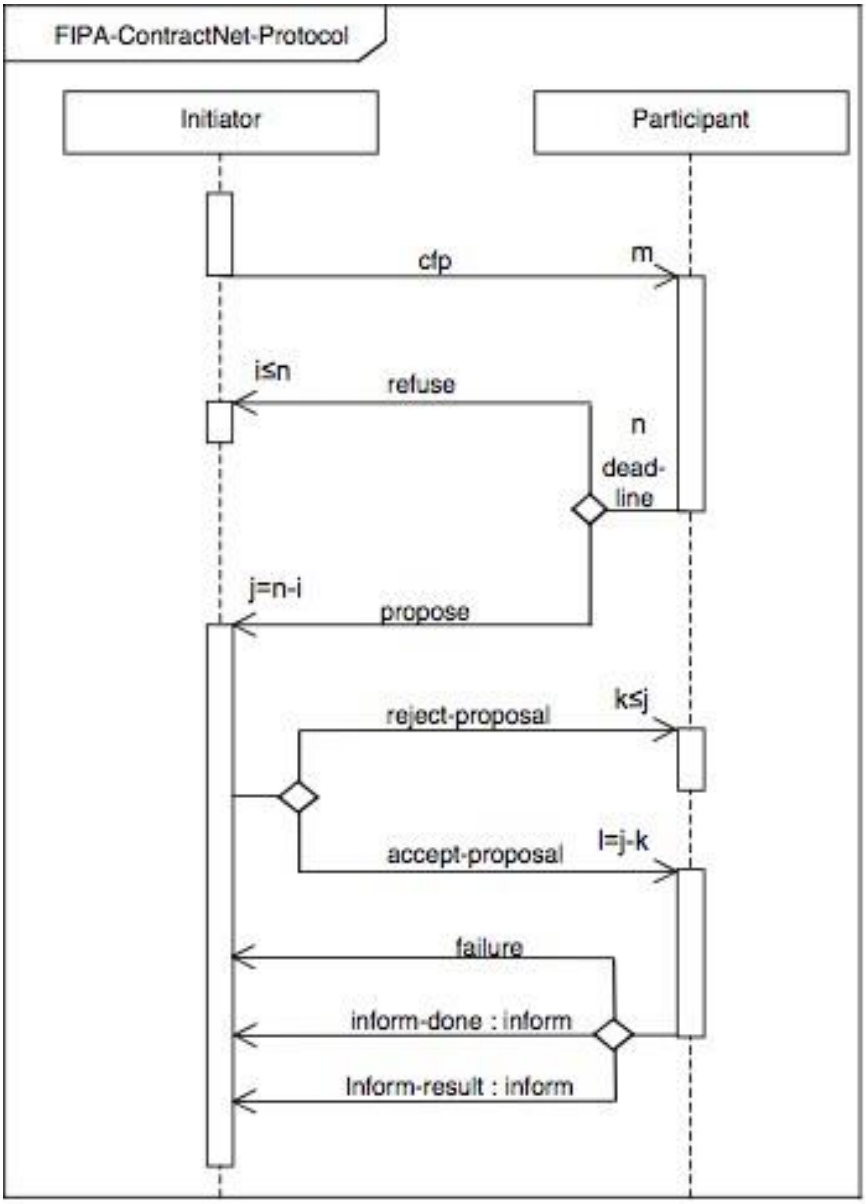

Ilustración 40. Imagen del flujo posible de interacción entre los distintos actos comunicativos

Como agentes tipo *Granja*, cuando se recibe un mensaje de tipo *CallForProposal*, es necesario chequear que se tenga stock disponible. Si es asi, se envía una propuesta (mensaje de tipo *Propose*) a dicho cliente, en caso de no contar con el stock solicitado, se envía un *Refuse*.

Si la propuesta enviada es aceptada, es decir, el cliente contesta con un mensaje de tipo *AcceptProposal*, se vuelve a chequear el stock, si hay disponibilidad, se le envía un *InformDone*, en caso de que no haya la disponibilidad prometida, se le envía un *Failure*.

Si la oferta es rechazada (con un mensaje de tipo *RejectProposal*), nuevamente se chequea stock y se vuelve a generar la misma propuesta al cliente.

Clase Empacadora

En segundo lugar se encuentra la clase *Empacadora*, la cual, entre las tres clases comentadas en esta tesina, es la que tiene un comportamiento más complejo. Esta complejidad viene dada a que en caso de no contar con stock disponible, cuando recibe un pedido, intenta, a través de sus proveedores, conseguir los materiales necesarios para fabricar su producto y luego elaborar una propuesta al agente que le realizó la solicitud. Para esto, en el modelo interno del agente hay que definir el producto que va a brindar y los materiales necesarios para poder fabricar un elemento de este producto. Estos elementos de fabricación serán llamados *Bill Of Materials* (bom) y se ven de la siguiente manera:

```
'use strict';
var Agent = require('../agent');
let Message = require('../message');

class Empacadora extends Agent {

  constructor(dataAgent={}){
    dataAgent.agentType = 'Empacadora';
    super(dataAgent);
  }
  // Subclasses should redefine this method
  defaultStore(){
    return {
      "products": {
        "Paquetes de verduras": {"price": 10, "count": 0},
      },
      "bom": {
        "Tomates": 5,
        "Zanahorias": 6,
        "Zapallos": 7,
        "Envases": 1
      }
    }
  }
}
```

Ilustración 41. Imagen que ejemplifica el modelo por defecto de la memoria del agente.

Una vez definido el modelo por defecto de la clase *Empacadora*, se está en condiciones de poder dar de alta agentes de este tipo. Cuando se da de alta un nuevo agente y se elige la clase *Empacadora*, el *content* se conforma de los productos que va a brindar la clase y el *bom*, es decir lo necesario para poder elaborar más productos.

Por otro lado, es necesario que esta clase conozca quiénes son los proveedores de suministros, así cuando necesita fabricar más productos, puede realizar peticiones a sus proveedores.

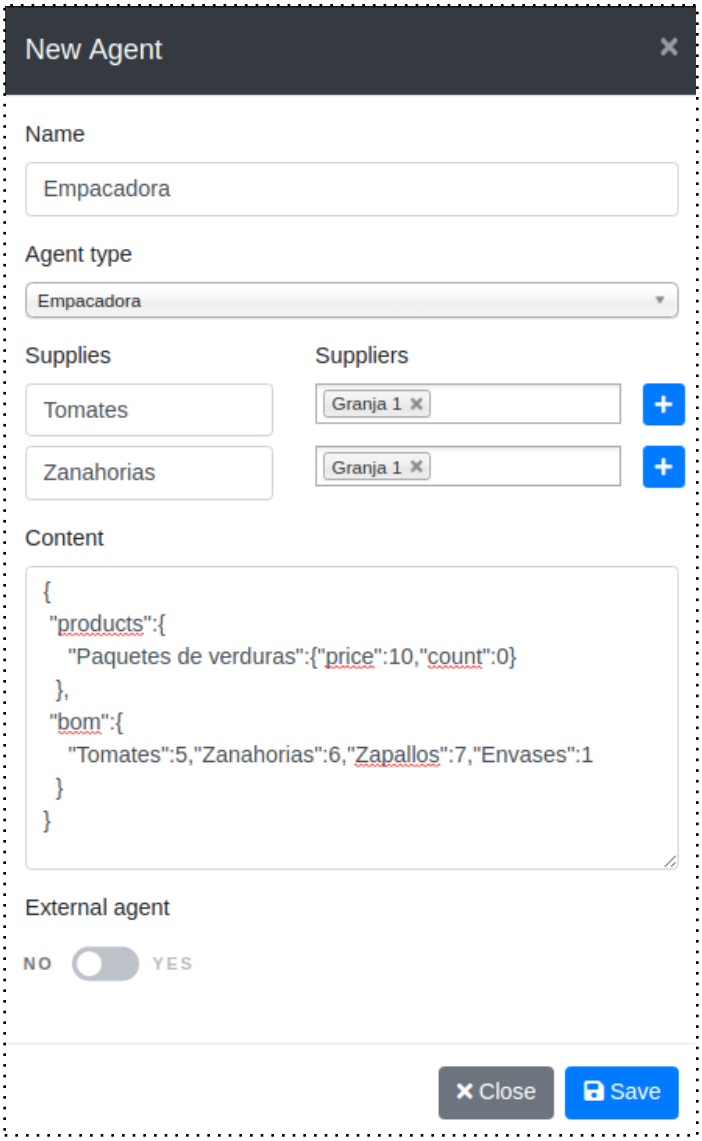

Ilustración 42. Ventana de creación de nuevo agente en el frontend del sistema.

En este momento está faltando definir el comportamiento de clase ante los distintos mensajes que la misma puede recibir por parte de los agentes de la cadena. Por lo tanto hay que definir cómo se debe de comportar el agente cuando reciba un *CallForPorposal*, un *Propose*, un *AcceptProposal,* un *Refuse*, etc.

```
'use strict';
var Agent = require('../agent');
let Message = require('../message');

class Empacadora extends Agent {

  constructor(dataAgent={}){
    dataAgent.agentType = 'Empacadora';
    super(dataAgent);
  }
  // Subclasses should redefine this method
  defaultStore(){
    return {
      "products": {
        "Paquetes de verduras": {"price": 10, "count": 0},
      },
      "bom": {
        "Tomates": 5,
        "Zanahorias": 6,
        "Zapallos": 7,
        "Envases": 1
      }
    }
  }
   // Subclasses should redefine this method
  processCallForProposal(message){
  }
  // Subclasses should redefine this method
  processPropose(message){
  }
  // Subclasses should redefine this method
  processAcceptProposal(message){
  }
  // Subclasses should redefine this method
  processRefuse(message){
  }
  // Subclasses should redefine this method
  processInformDone(message){
  }
}
```

Ilustración 43. Métodos de la clase Empacadora, que definen su comportamiento ante la llegada de los distintos tipos de mensaje.

Se espera que esta clase cuando reciba un *CallForProposal* chequee si tiene stock disponible, en caso de que así sea, prepara una propuesta para el agente que le realizó la solicitud. En caso de no tener stock disponible, verifica de tener al menos un proveedor por cada material necesario para la elaboración del producto, si esto se cumple, envía un *CallForProposal* a cada uno de sus proveedores solicitando la cantidad necesaria de materiales para poder satisfacer el *CallForProposal* que él recibió. En caso de no poder elaborar su producto por la falta de proveedores, el agente rechaza el pedido original generando un *Refuse*.

Recibir un *Propose* para el agente quiere decir que hubo un *CallForProposal* previo que le solicitó *n* productos y él no los puede cumplir con su stock, por lo tanto, se comunicó con sus proveedores para elaborar más productos. El recibir una propuesta genera, que para un mismo material, se acepte la propuesta del proveedor que ofrece el menor precio por la cantidad solicitada. Para esto el agente espera a que todos los proveedores le envíen un *Propose* por el *CallForProposal* que él les envió. Una vez que tiene todas las propuestas, se

acepta la de menor precio, por lo tanto, se elabora un *AcceptProposal* por cada material necesario para elaborar más productos y así satisfacer la demanda original.

Recibir un *InformDone* para el agente quiere decir, que el *AcceptProposal* que él le envió al agente proveedor fue confirmado, por lo tanto el agente proveedor le brinda los materiales al agente *Empacadora* por el precio establecido. El agente *Empacadora* espera recibir un *InformDone* por cada *AcceptProposal* que realizó, una vez que todos los *InformDone* llegan, está en condiciones de poder aumentar su stock y de esta forma poder cumplir con el *CallForProposal* original, por lo tanto, aumenta su stock y le envía una propuesta a su agente cliente.

Cuando llega un *AcceptProposal* quiere decir que el agente cliente acaba de aceptar la propuesta por tantos elementos y a un determinado precio. Por lo tanto, se hace un chequeo de poder cumplir con la petición, si es asi, se envía un *InformDone* al agente cliente, en caso de no poder cumplir con la petición que se envía al agente cliente (esto puede darse porque otro agente cliente solicitó productos y el trato se cerró antes que la negociación actual) se envía un *Failure*, indicando que la negociación falló.

Refuse es un mensaje que se recibe por parte de un proveedor. Cuando esto sucede, quiere decir que algún proveedor no está pudiendo brindar los elementos que son necesarios, por lo tanto, no se va a poder cumplir con el *CallForProposal* recibido anteriormente, y entonces se traslada este *Refuse* al agente que hizo el *CallForProposal* original.

Clase Distribuidor

Para poder completar esta simulación falta definir la clase *Distribuidor*, que su principal tarea es recibir pedidos de sus clientes y armar propuesta a estos, ya sea contando con stock o tratando de sumar más elementos a su stock a través de sus proveedores.

Lo primero que hay que definir es su modelo por defecto. En el mismo se va a establecer qué productos estarán disponibles para la venta, la cantidad de stock actual y el precio por unidad.

```
'use strict';
var Agent = require('../agent');
let Message = require('../message');

class Distribuidor extends Agent {

  constructor(dataAgent={}){
    dataAgent.agentType = 'Distribuidor';
    super(dataAgent);
  }
  // Subclasses should redefine this method
  defaultStore(){
    return {
      "products": {
        "Paquetes de verduras": {"price": 10, "count": 1},
      }
    }
  }
```

Ilustración 44. Imagen que ejemplifica el modelo por defecto de la memoria del agente.

Como en el resto de las clases, ya es posible dar de alta un agente de tipo Distribuidor. Cuando se elige este tipo de agente, se verá un *content* basado en el modelo por defecto establecido. En este caso, en el modelo solo se tiene el producto que se va a ofrecer a la cadena con el stock disponible y el precio del mismo.

Además, se va a establecer al agente *Empacadora* como proveedora de paquetes de verduras. En caso de no contar con stock disponible se le va a pedir a este agente que provea de paquetes de verduras.

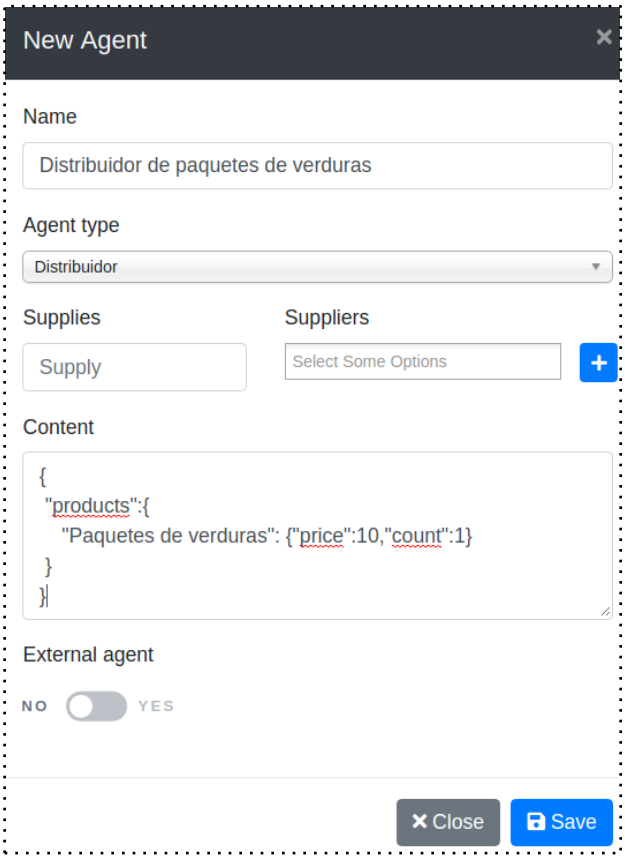

Ilustración 45. Ventana de creación de nuevo agente en el frontend del sistema.

Al igual que el resto de las clases, este es el momento de determinar cómo reacciona el agente antes los distintos mensajes que este puede recibir. En este caso, hay que definir cómo reacciona ante un *CallForProposal*, un *Propose*, un *AcceptProposal*, un *Refuse* y un *InformDone*.

```
'use strict';
var Agent = require('../agent');
let Message = require('../message');

class Distribuidor extends Agent {

  constructor(dataAgent={}){
    dataAgent.agentType = 'Distribuidor';
    super(dataAgent);
  }
  // Subclasses should redefine this method
  defaultStore(){
    return {
      "products": {
        "Paquetes de verduras": {"price": 10, "count": 1},
      }
    }
  }
  // Subclasses should redefine this method
  processCallForProposal(message){
  }
  // Subclasses should redefine this method
  processPropose(message){
  }
  // Subclasses should redefine this method
  processAcceptProposal(message){
  }
  // Subclasses should redefine this method
  processRefuse(message){
  }
  // Subclasses should redefine this method
  processInformDone(message){
  }
}
```

Ilustración 46. Métodos de la clase Distribuidor, que definen su comportamiento ante la llegada de los distintos tipos de mensajes.

Cuando el agente *Distribuidor* recibe un *CallForProposal* verifica si tiene stock disponible para poder satisfacer el pedido, en caso de que así sea, arma una propuesta. Si no tiene el stock disponible, realiza una petición a sus proveedores, en este caso le pide paquetes de verduras al agente *Empacadora*.

El recibir un *Propose*, significa que el agente realizó algún pedido a sus proveedores, en este caso a *Empacadora*. En caso de tener más de un proveedor de paquetes de verduras, el agente analiza todas las propuestas y se queda con la de menor precio, esto implica que le envía un *AcceptProposal* al agente al cual le va aceptar la propuesta.

Cuando llega un *InformDone*, significa que el *AcceptProposal* que se le envió al proveedor fue aceptado, entonces, de esta forma se puede aumentar el stock y enviar un *Propose* al agente que inició esta conversación (es decir, una propuesta al emisor del *CallForProposal* recibido).

Si se recibe un *Refuse*, significa que fue realizado un pedido a un proveedor, por la necesidad de aumentar el stock. Si los proveedores no pueden satisfacer la solicitud implica que el agente *Distribuidor* tampoco puede cumplir con el *CallForProposal* recibido, por lo tanto, se encarga de enviar un *Refuse* al agente que le envió el *CallForPorposal*, de esta forma termina la simulación.

El *AcceptProposal*, significa que la propuesta enviada por el agente *Distribuidor* al agente cliente que inició la simulación fue aceptada, por lo tanto queda verificar si es posible cumplir con esa propuesta previamente enviada. En caso de ser así, se envía un *InformDone*. Si no es posible cumplir esa propuesta, no queda otra opción que enviar un *Failure* al agente.

Como se comentó anteriormente en la clase *Empacadora*, puede darse el caso de enviar una propuesta y luego no poder cumplirla, dado que otra negociación con otro agente se finalizó antes, dejando sin stock disponible al agente para que pueda finalizar este trato, por ejemplo.

El *Failure* enviado indica que la conversación falló y si se desea, se debe de comenzar de nuevo, generando una nueva conversación.

Construcción de la cadena de suministros

Luego de presentar una descripción general de la cadena de suministros de ejemplo y de explicar cada una de las clases que la conforman, se va a demostrar cómo crear esta cadena de suministros utilizando la herramienta desarrollada por el framework SCS.

Este ejemplo se puede realizar tanto en el ambiente de ejecución local como en AWS. Para comenzar, abrir el sistema con el navegador web, el cual mostrará una ventana similar a esta:

Ilustración 47. Imagen del frontend del sistema sin ninguna cadena creada.

Para crear una nueva cadena se debe clickear en el botón *New*. Ingresar los datos solicitados para la creación de la cadena. Estos datos son *Name* (el nombre de la cadena), *Template* (indica si la cadena será un *Template* o no, esto significa que dicha cadena sólo se utilizará como plantilla de otras cadenas y no para realizar simulaciones) y *Description*

(Una descripción breve de la cadena de suministros). Luego de ingresar los datos solicitados, clickear el botón *Save* para que la cadena se guarde.

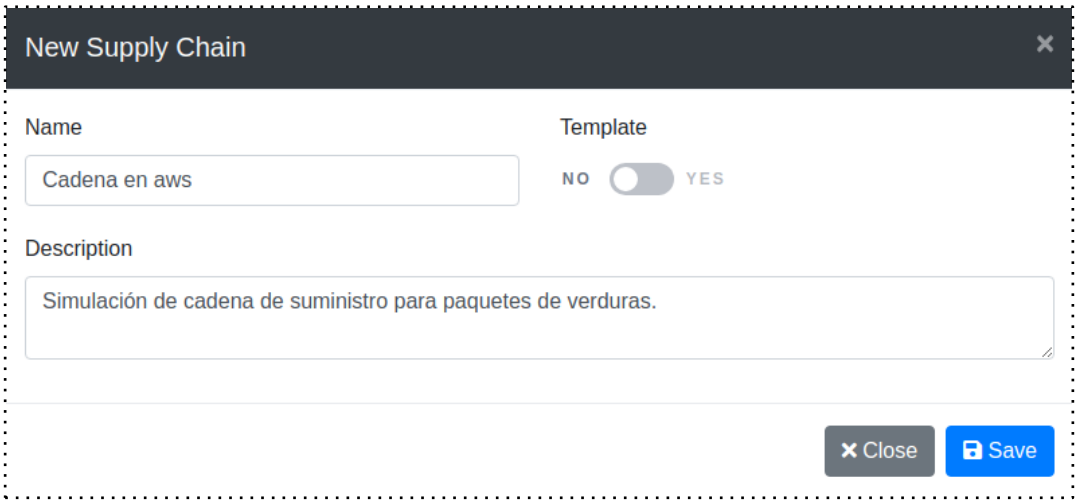

Ilustración 48. Imagen del formulario de creación de nueva cadena de suministros.

Si la creación es exitosa el sistema mostrará un mensaje similar a este:

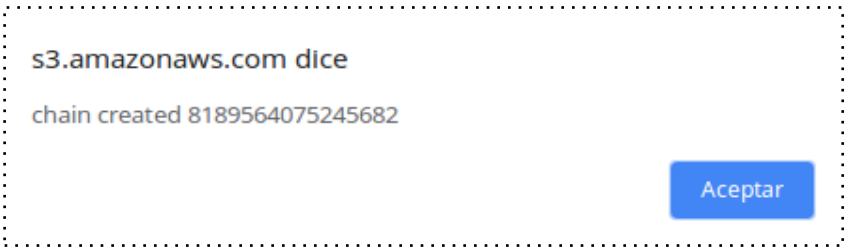

Ilustración 49. Mensaje que indica la creación exitosa de una cadena de suministros nueva.

Luego de la creación de la cadena, es posible ver que la misma se representa en la tabla de cadenas disponibles:

Ilustración 50. Tabla de cadenas de suministros disponibles.

Además de los datos ingresados para la creación de la cadena (*Name*, *Description* y *Template*), esta tabla también muestras las acciones que se pueden realizar en cada una de las cadenas. Las opciones son:

- **Go chain:** Ingresa a la cadena seleccionada. Allí se puede gestionar a los agentes que la conforman y ejecutar las simulaciones.
- **Edit chain:** Edita la cadena seleccionada.
- **Clone chain:** Clona la cadena seleccionada. Es decir, genera una copia idéntica de esta cadena, con sus agentes integrantes. Notar que si la cadena copiada es de tipo *Template*, la cadena copia no lo será.
- **Clear chain:** Limpia todas las conversaciones de la cadena.

Para proseguir con la creación de la cadena, ingresar a la opción *Go chain.* Aquí se pueden ver todos los datos de la cadena recién generada, además gestionar a los agentes que la conforman y realizar las simulaciones.

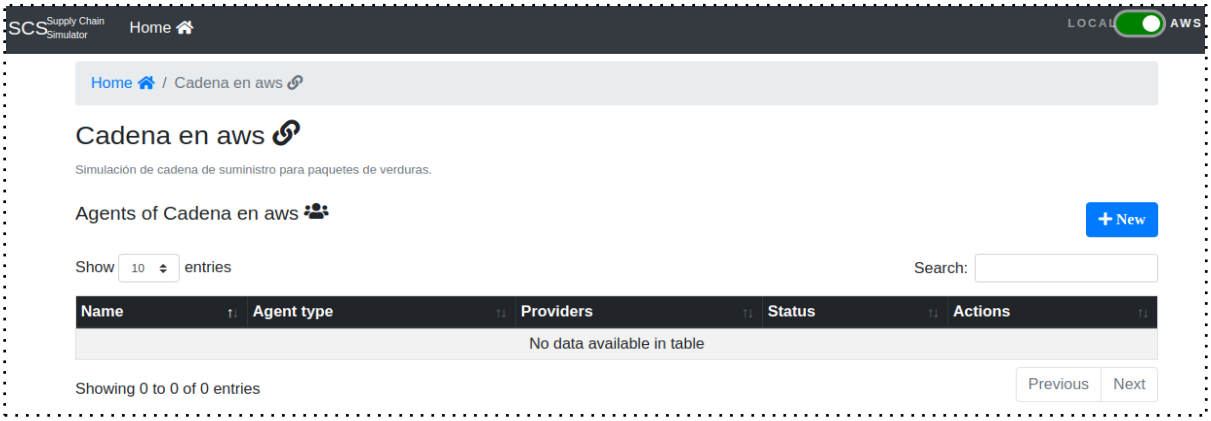

Ilustración 51. Ventana de gestión de agentes y simulaciones.

El siguiente paso es la creación de los diferentes agentes que conforman la cadena de suministros. Para este ejemplo se van a crear 5 agentes en total, los mismos serán 3 agentes de tipo *Granja* (es decir 2 *Granjas* y 1 *Envases*), 1 agente de tipo *Empacadora* y 1 agente de tipo *Distribuidora*.

Para la creación de agentes se debe hacer click en el botón *New*, el cual desplegará la siguiente ventana para ingresar los datos necesarios de cada agente. Estos datos son:

- **Name:** el nombre del agente.
- **Agent type:** el tipo de agente. Este dependerá de las clases desarrolladas anteriormente, en este caso *Granja, Empacadora* y *Distribuidora*.
- **Supplies y suppliers:** Lista de proveedores por producto del agente. En supplies se coloca el producto que se proveerá y el suppliers se establece la lista de proveedores para ese producto. Los mismos deben ser parte de la misma cadena de suministros.
- **Content:** datos que contiene cada agente. Por ejemplo el producto que elabora, el precio y cantidad en stock que tiene, los productos necesarios para lograr la generación de dicho producto, entre otras. El *Content* debe tener formato JSON para ser válido.
- **External agent:** Indica si el agente es un agente externo o no, es decir, que se encuentra en un sistema externo. Si el agente es externo, se habilita la opción *URL*.
- **URL:** indica la URL del agente externo para ser accedido desde este sistema.

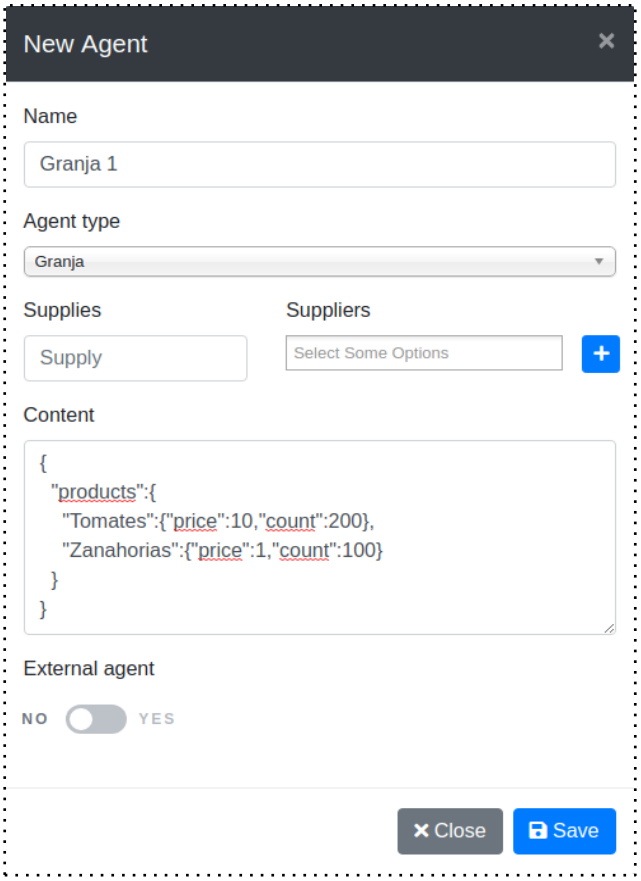

Ilustración 52. Ventana de creación de agente.

Primero se van a crear los agentes de tipo *Granja*, es decir, los 2 agentes *Granja* y el agente *Envases*. Para estos agentes en cuestión, se debe seleccionar *Agent type Granja*, no poseen proveedores, ya que son agentes productores de sus propios productos y no necesitan de ningún otro producto para poder generarlos, por ende su lista de proveedores va a ser vacía y el *Content* será un JSON con los productos que fabrica el agente, el precio y la cantidad de stock que posee. Para este ejemplo solo se utilizan agentes internos, entonces la opción de *External agent* debe ser *No*.

Luego de completar los datos solicitados, hacer click sobre el botón *Save* para guardar al agente. Una vez que el agente se guarda de forma exitosa, puede verse una tabla similar a la de cadenas, pero esta vez contiene a los agentes, algunos de sus datos y las acciones que son posibles de ejecutar para cada agente. Estas acciones son:

- **Go agent:** Ingresa al agente seleccionado. Allí se podrá enviar mensajes a los agentes y ver las conversaciones del agente con sus clientes y/o proveedores.
- **Send message:** Envía un mensaje al agente seleccionado, lo que desencadena la ejecución de una nueva simulación. Notar que si el agente integra una cadena de tipo *Template*, esta opción no se encontrará disponible, debido a que las cadenas de este tipo solo se utilizan como plantillas para crear otras cadenas de suministros.
- **Edit agent:** Edita al agente seleccionado.

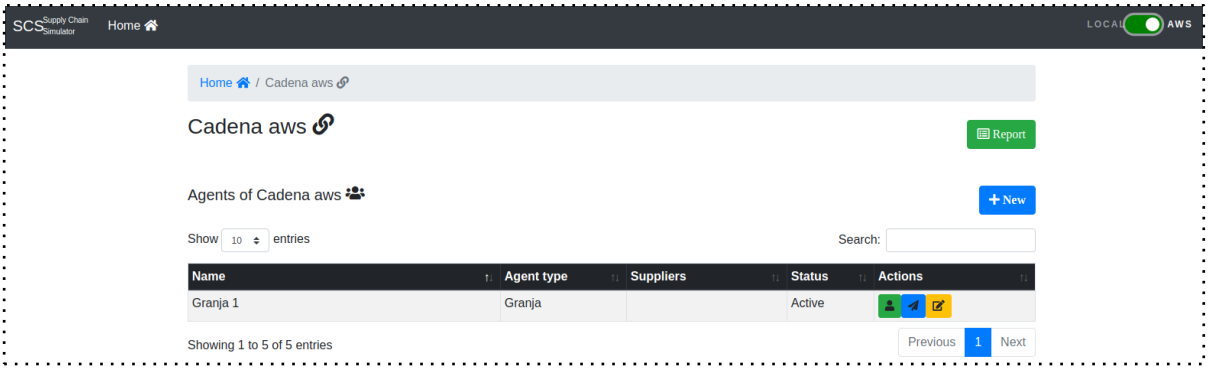

Ilustración 53. Tabla de agentes que componen la cadena de ejemplo.

Repetir lo mismo para los tres agentes de tipo *Granja*, es decir, los dos agentes *Granja* y el agente *Envases*. El agente *Envases* es similar a los agentes *Granja* y se muestra a continuación:

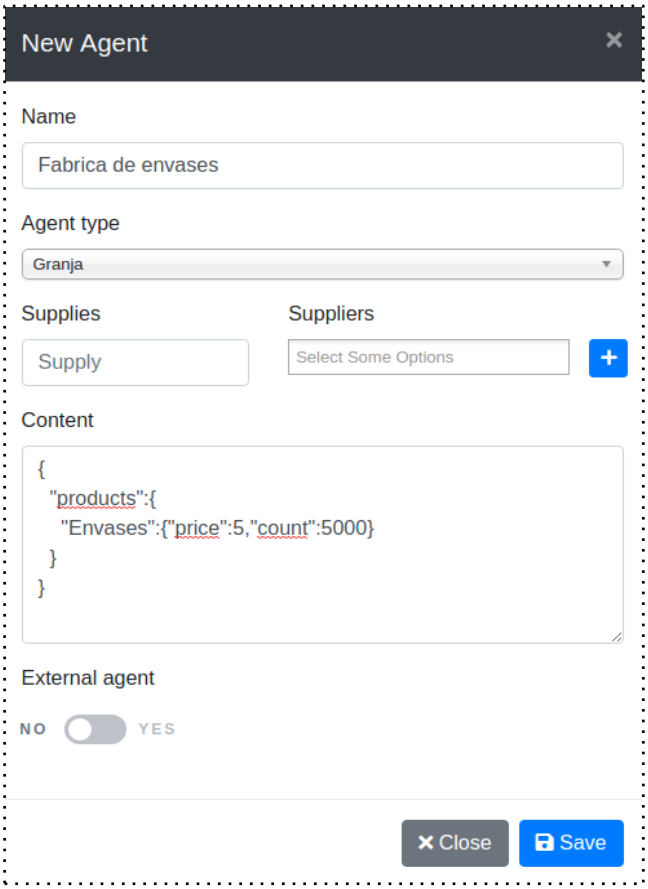

Ilustración 54. Ejemplo de agente Envases.

Luego de los 3 agentes de tipo *Granja*, se va a crear al agente de tipo *Empacadora*. Para esto, darle un nombre, seleccionar el *Agent type Empacadora* y seleccionar a los tres agentes anteriormente creados como sus proveedores. Además, su *Content* va a estar formado por el producto que elabora (en este caso “Paquetes de verduras”) y su *bom*. Por ejemplo, el agente Empacadora necesita 5 Tomates, 6 Zanahorias, 7 Zapallos y 1 Envase, para generar 1 Paquete de verduras.

Ilustración 55. Ventana de creación del agente Empacadora.

El último agente a crear es el agente *Distribuidor*. Para su creación, se sigue un procedimiento similar a los anteriores, seleccionando como *Agent type Distribuidor* y en su lista de proveedores seleccionar a *Empacadora*. En este ejemplo, el agente *Distribuidor* comienza con stock 0 de Paquetes de verduras.

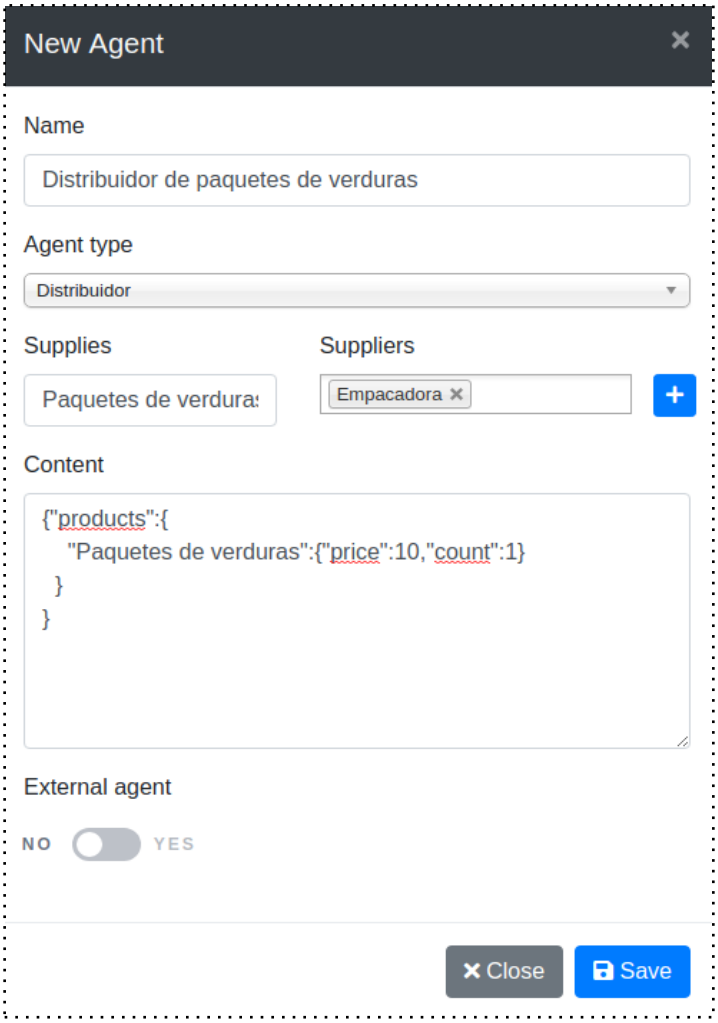

Ilustración 56. Ventana de creación del agente Distribuidor.

La cadena de suministros de “Paquetes de verduras para sopa” debe quedar modelada de la siguiente manera:

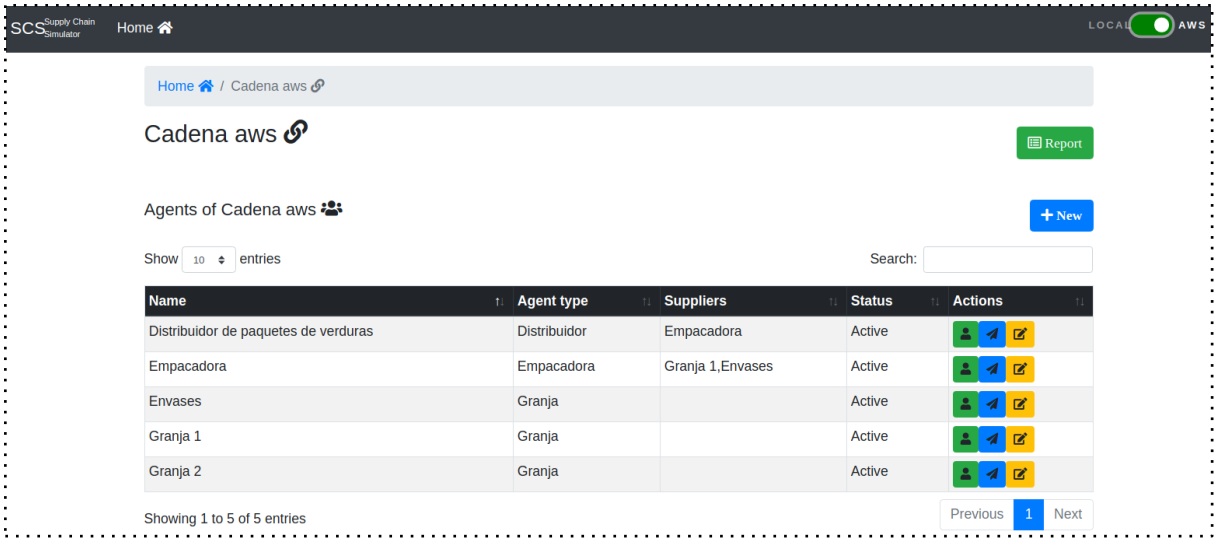

Ilustración 57. Tabla de agente de la cadena de ejemplo.

Una vez que el modelado de la cadena de suministros esté completo, es momento de realizar la primer simulación. Esto se puede realizar desde el botón *Send message*, en la tabla de agentes de la cadena o desde el interior de un agente. Para explorar una nueva ventana, adentrarse en uno de los agentes, más precisamente en el agente *Distribuidor*. Para esto es necesario hacer click en el botón *Go agent* del agente *Distribuidor*.

Una vez dentro de esta nueva pantalla, se pueden observar todos los datos del agente en cuestión, incluyendo una lista de conversaciones que actualmente está vacía. Para comenzar con la simulación hacer click en el botón *Send message*, el cual desplegará la ventana de envío de mensaje. En la misma, se puede observar el nombre del agente receptor del mensaje y el contenido del mensaje a enviar. Nótese que el mensaje cuenta con un contenido por defecto, dicho contenido fue programado en la clase del agente y, al igual que el contenido del agente, también posee formato JSON.

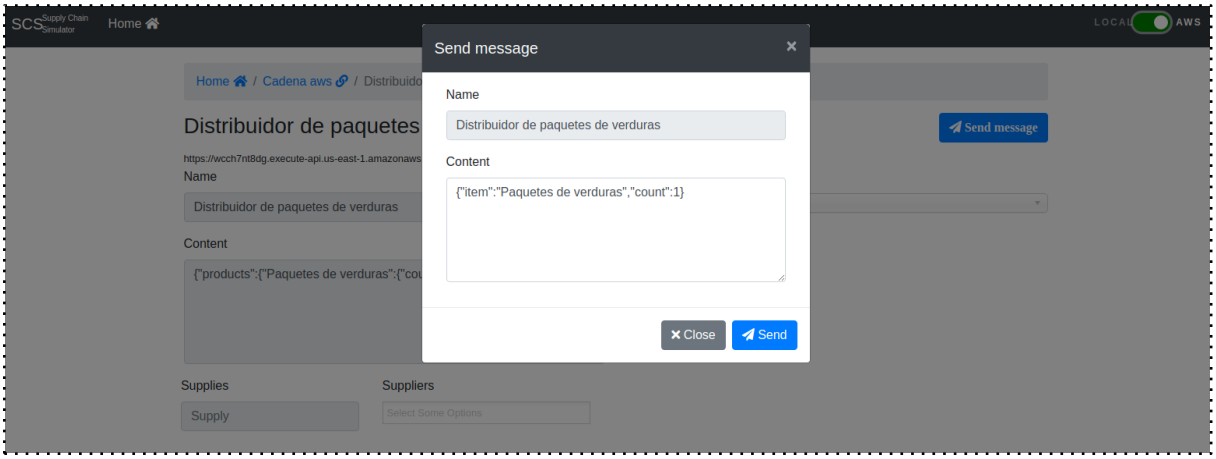

Ilustración 58. Imagen de envio de mensaje a un agente.

Luego de completar el mensaje, hacer click sobre el botón *Send* para enviar dicho mensaje al agente seleccionado. Esto dará comienzo a la ejecución de la simulación de la cadena de suministros, es decir, a la interacción de los distintos agentes de la cadena, que negocian entre sí para lograr obtener el mejor escenario posible para satisfacer el pedido hecho en el mensaje recién enviado.

Una vez que la simulación es finalizada, se genera una nueva conversación, donde pueden apreciarse todos los mensajes intercambiados entre los agentes. Esto puede verse en la lista de conversaciones, que ahora contiene una conversación.

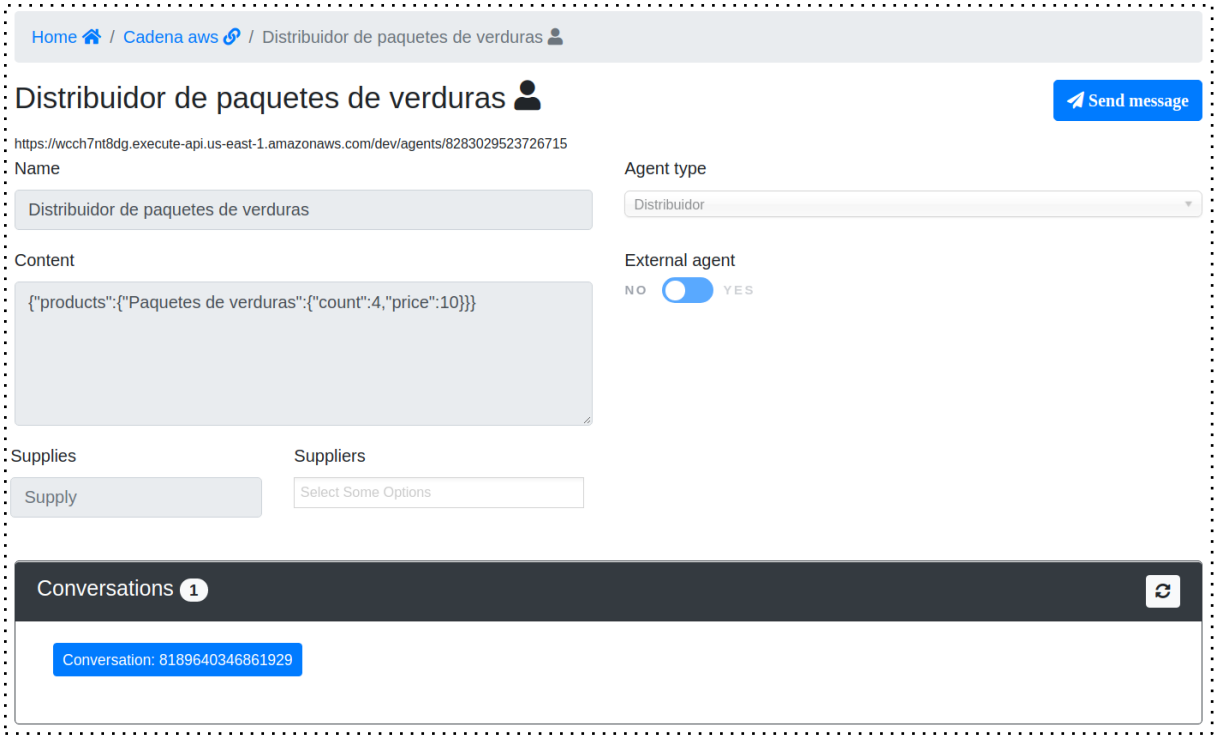

Ilustración 59. Listado de conversaciones del agente seleccionado.

Para acceder a ver los mensajes que conforman la conversación recién creada, hacer click sobre la misma, para así expandirla.

Todos los mensajes respetan la misma composición:

- **Tipo de mensaje:** En este ejemplo, los posibles tipos de mensaje son *CallForProposal*, *Propose*, *AcceptProposal*, *RejectProposal*, *InformDone*, *Refuse* o *Failure*.
- **Sender:** El agente emisor del mensaje.
- **Receiver:** El agente receptor del mensaje.
- **Content:** El contenido del mensaje, el cual varía dependiendo del tipo de mensaje.

Hay 2 cuestiones a tener en cuenta. Una, es que puede apreciarse que el primer mensaje de la conversación es el enviado por el usuario (*You*) desde el frontend del sistema al agente seleccionado. Y la segunda cuestión a tener en cuenta, es que sólo se pueden ver los mensajes que tienen que ver con el agente seleccionado, por ejemplo, estando dentro del agente *Distribuidor* no podrán verse los mensajes intercambiados entre los agentes *Envases* y *Empacadora*.

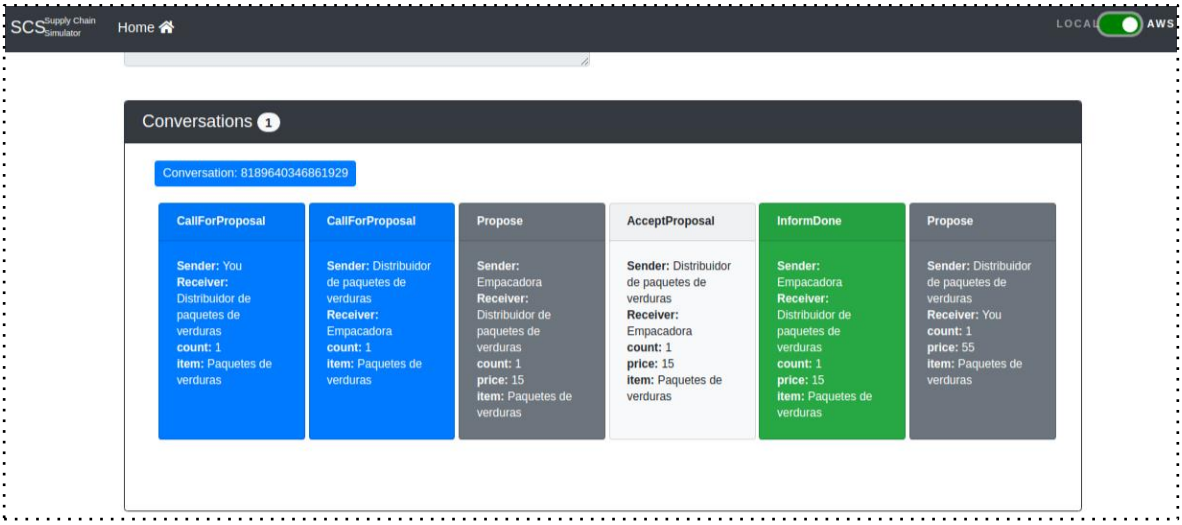

Ilustración 60. Conversación generada como resultado de la simulación del modelo de cadena de suministros.

Además de poder observar los mensajes, la herramienta permite visualizar un reporte sobre las conversaciones que posee una cadena. Para esto, dentro de la cadena se puede ver el botón *Report*, que al hacerle click mostrará todas las conversaciones que posea la cadena.

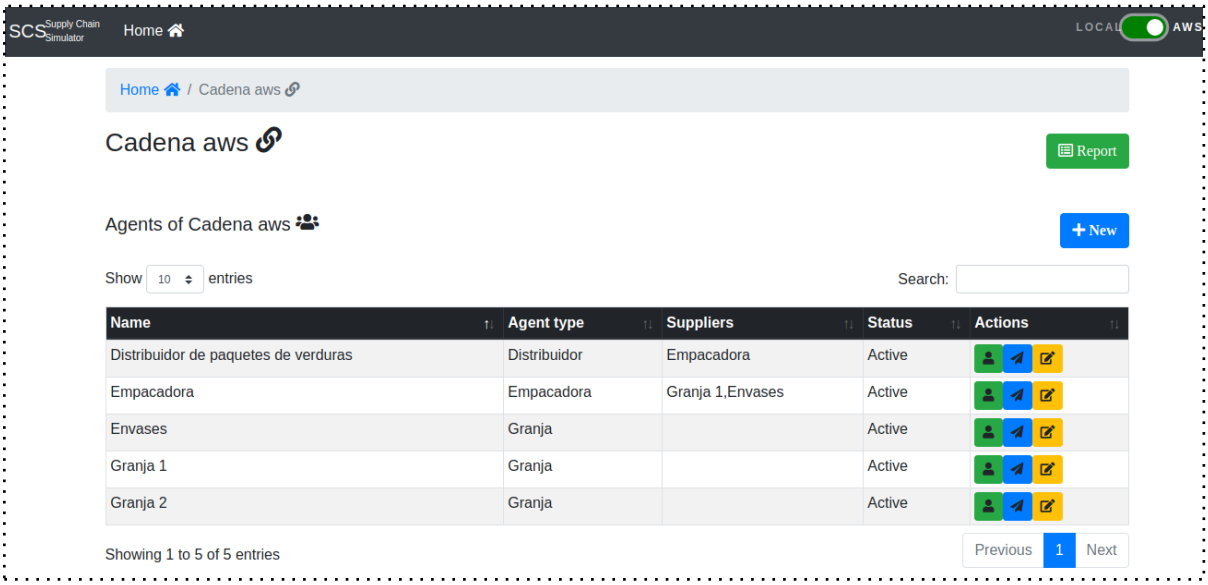

Ilustración 61. Cadena aws con botón Report.

En este caso, la *Cadena aws* solo posee una conversación, que al hacerle click muestra un conjunto de métricas resultantes de dicha conversación.

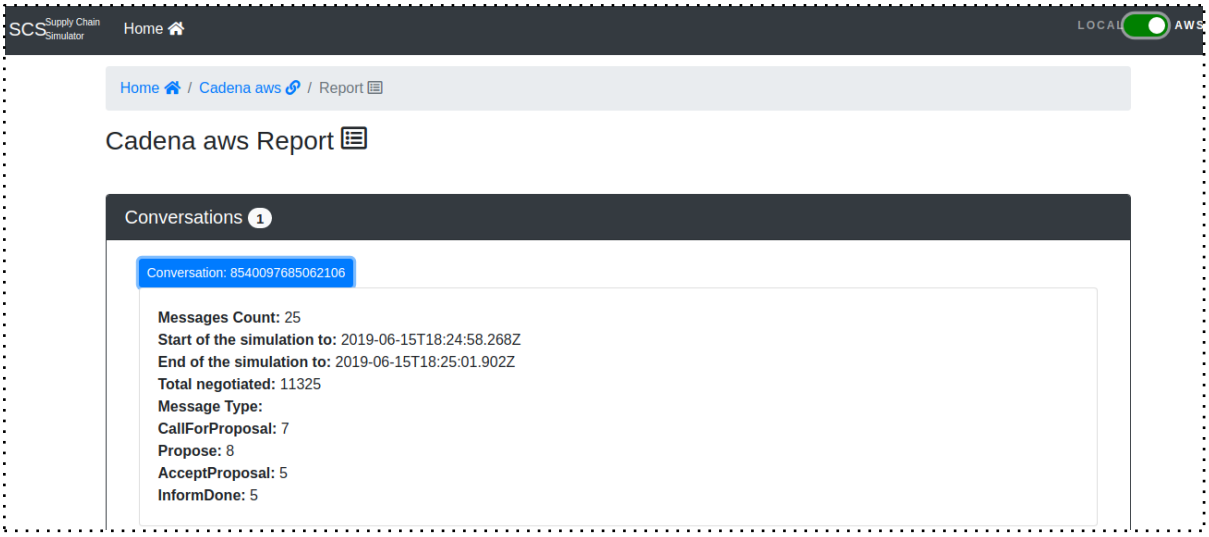

Ilustración 62. Reporte de conversación.

Estas métricas tienen en cuenta todos los mensajes que se intercambiaron en la conversación entre los distintos agentes.

En este reporte se puede observar:

- **Messages Count:** Cantidad de mensajes intercambiados, se cuentan todos los mensajes que se enviaron en dicha conversación.
- **Start of the simulation to:** Fecha y hora de inicio de la simulación, en base al primer mensaje podemos obtener el momento en que inicia la conversación.
- **End of the simulation to:** Fecha y hora de fin de la simulación, en base al último mensaje podemos obtener el momento preciso cuando finaliza la simulación.
- **Total negotiated:** Dinero total negociado. Es la cantidad de dinero que fue negociada entre todos los agentes de la cadena.
- **Message Type:** Además, por cada tipo de mensajes se cuenta la cantidad que aparecen en la conversación.

Agente interactivo

A continuación se realizará una demostración de cómo un agente externo puede interactuar con la cadena previamente armada en esta tesina.

Desde un agente creado en el cliente interactivo se envían mensajes al *Distribuidor de paquetes de verduras*.

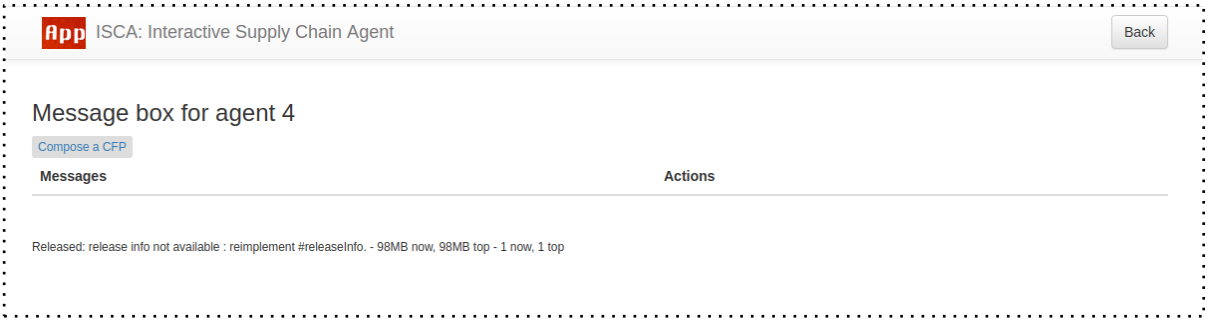

Ilustración 63. Imagen del cliente interactivo, en este caso el agente 4.

Para enviar un mensaje desde el cliente interactivo al agente *Distribuidor* se debe hacer click en *Compose a CFP* y luego completar los datos del mensaje.

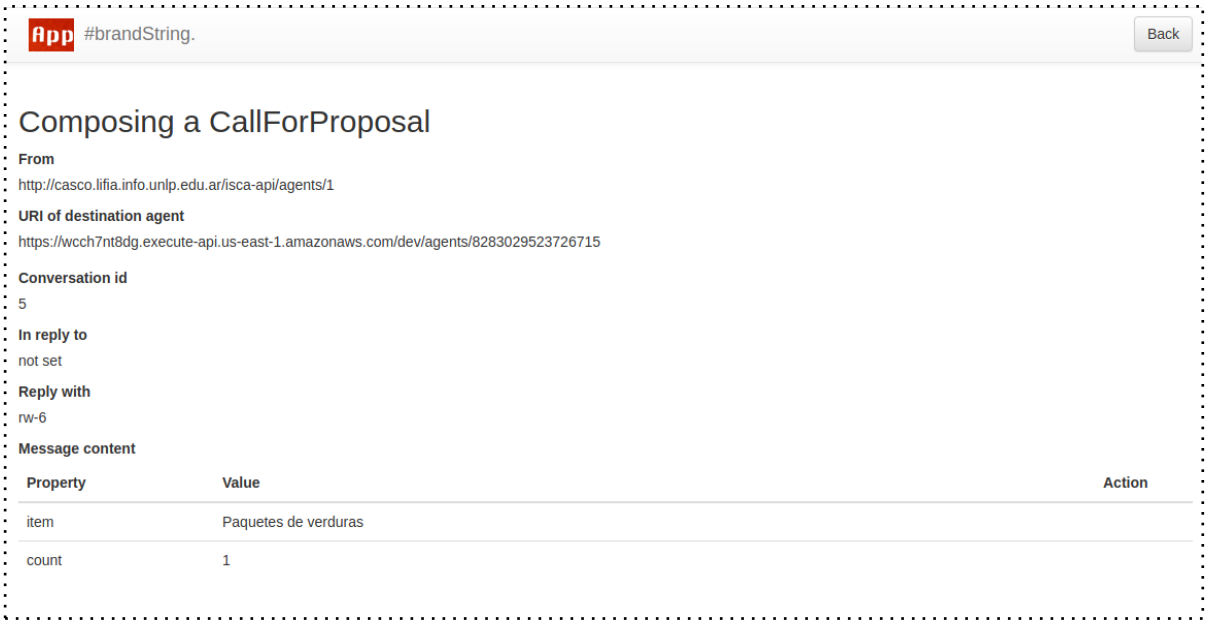

Ilustración 64. Imagen que representa la composición de un CFP.

Una vez que se arma el mensaje, se clickea *Send* para realizar el envío del mismo. Luego accediendo a la herramienta de gestión de cadenas del framework se puede observar el mensaje enviado al *Distribuidor de paquetes de verduras*.

El agente que lo recibe, dependiendo en el estado en que se encuentre, deberá o no intercambiar mensajes con sus proveedores de suministros para poder cumplir con el *cfp* que recibió. En este caso, se pidió 1 paquete de verduras y el *Distribuidor* cuenta con el stock necesario para satisfacer el pedido, por lo tanto, arma un *propose* como respuesta del *cfp* recibido.

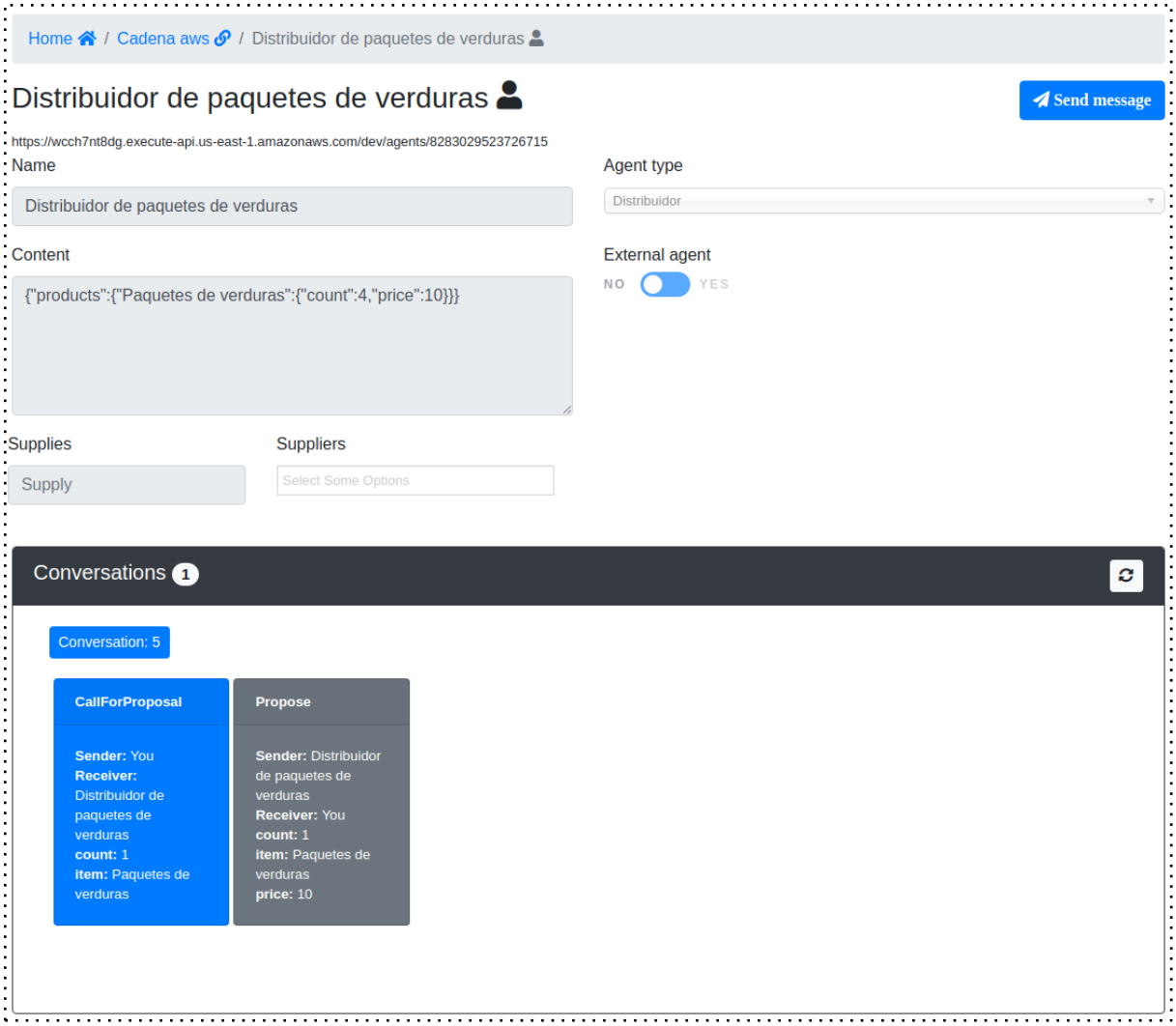

Ilustración 65. Imagen del distribuidor de paquetes de verduras.

Luego en el cliente interactivo, se puede ver, en los mensajes del agente, que tiene un nuevo mensaje de tipo *propose*. En el mismo, se dispone de dos acciones posibles, aceptar o rechazar.

Ilustración 66. Imagen de los mensajes del agente interactivo.

En este ejemplo se va a realizar la aceptación de la propuesta, lo que genera un mensaje de respuesta con el formato que se observa en la siguiente imagen:

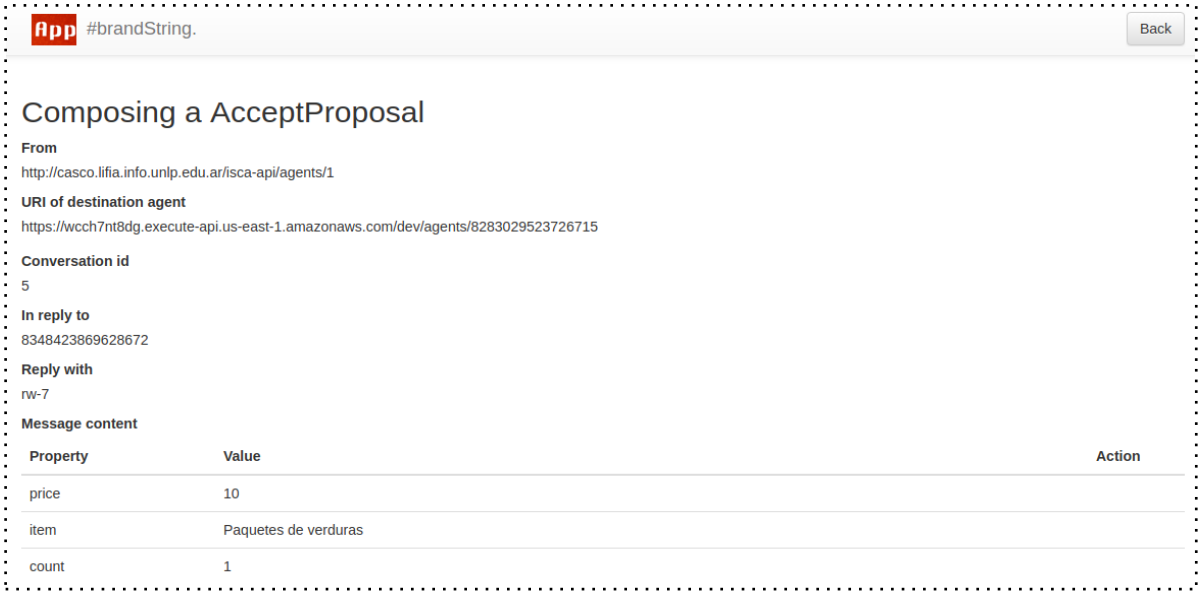

Ilustración 67. imagen de un mensaje de aceptación.

Cuando al *Distribuidor* le llega la aceptación de la propuesta, evalúa si puede satisfacer la misma, es decir, comprueba que tenga stock disponible para tal propuesta. Esto lo debe de realizar, dado que un agente puede en un mismo instante estar negociando con varios agentes a la vez.

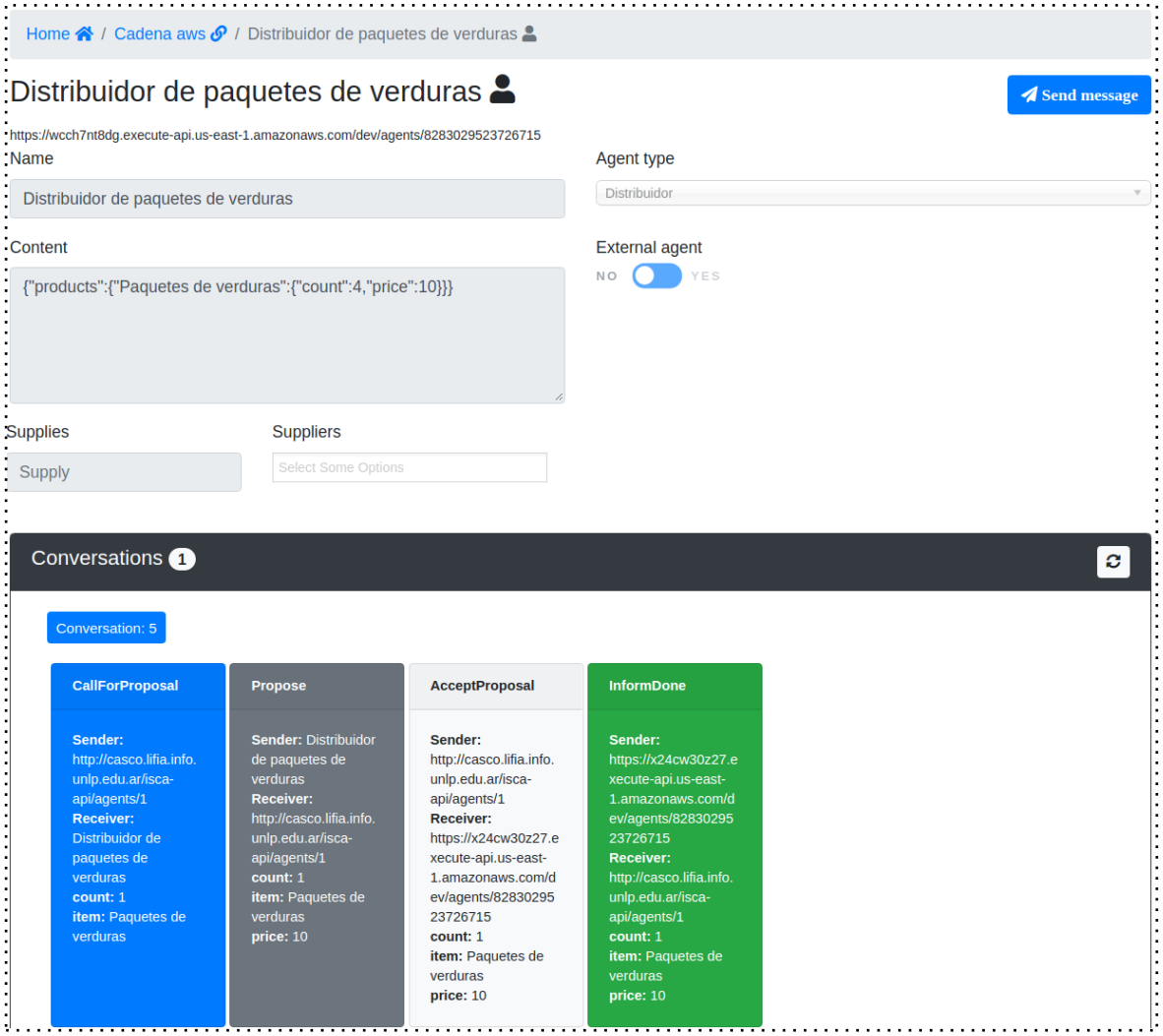

Ilustración 68. Imagen de mensajes del Distribuidor de paquetes de verduras.

En este caso, el agente cuenta con stock disponible, por lo tanto, envía un mensaje de tipo *InforDone* al cliente interactivo.

En el cliente interactivo se puede observar la recepción del *InforDone*. Esto significa que la propuesta fue realizada correctamente por el agente *Distribuidor*.

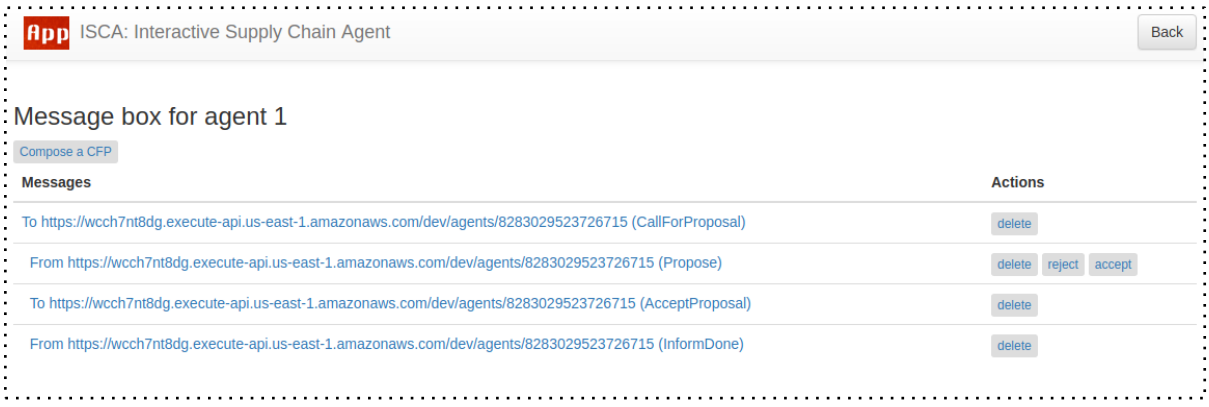

Ilustración 69. Imagen cliente interactivo negociación completada.

Como puede verse, se ha realizado una conversación completa entre un agente interactivo y un agente interno, el cual fue implementado utilizando el framework desarrollado en esta tesina de grado. Con esto se puede observar, que el framework no solamente permite que cadenas generadas en sistemas desarrollados por el mismo interactúen entre sí, sino que además ofrece la posibilidad de que las cadenas se comuniquen con cadenas y/o agentes externos a la misma.

Capítulo 8

Conclusiones

Gracias al desarrollo de este framework se logró simplificar el trabajo de creación de sistemas de modelado y simulación de cadenas de suministros, brindando de esta forma que los sistemas desarrollados a partir del mismo requieran menor costo y alcancen una mejor calidad, agregando una capa de abstracción sobre temas primordiales como el almacenamiento, la comunicación entre agentes y el control sobre las simulaciones realizadas, temas que el usuario de este framework no necesita saber con anterioridad para poder desarrollar sus propios sistemas de modelado y simulación de cadenas de suministros.

En relación a estrategias ya existentes, se lograron mejoras en:

- Flexibilidad, debido a que gracias a este framework, el usuario puede desarrollar un sistema a su medida. Capacidad ausente en los sistemas actuales, ya que son de código cerrado y no se permite la modificación de los mismos para ajustarse mejor a las necesidades de los usuarios.
- Implementación, ejecución y despliegue, gracias a los posibles entornos que poseen la herramienta visual y el framework. Esta posibilidad, de tener un entorno local y otro en la nube, es algo con lo que no cuentan los sistemas actuales, debido a que los mismos corren simplemente sobre un servidor centralizado.
- Interacción, debido a la utilización de nuevas tecnologías y estándares. Gracias a esto, es posible desarrollar sistemas con la habilidad de interactuar con otros sistemas externos, sin la necesidad de que los mismos estén desarrollados con las mismas herramientas o estén montados sobre las mismas arquitecturas, formando así una interconexión de sistemas distribuidos para modelar y simular cadenas de suministros.

Además, el editor gráfico de cadenas ofrece al usuario la posibilidad de monitorear y simular de forma amigable las cadenas de suministros creadas con esta herramienta. Realizando una gestión completa del modelado de la cadena, los agentes que la conforman, utilizando agentes locales o externos, simulando las negociaciones, limpiando todas las conversaciones realizadas con anterioridad y obteniendo reportes con las métricas obtenidas como resultado de las simulaciones ejecutadas.

Se pudo experimentar y demostrar todas estas conclusiones en un caso concreto, ejemplificando de forma sencilla una problemática compleja del mundo real. Se pudo ver el caso concreto de una cadena de suministros de “Paquetes de verduras para sopa”, como es posible desarrollar un sistema capaz de modelarla y simularla a partir del framework creado para esta tesina y quedó demostrado como el mismo se puede utilizar para modelar y simular múltiples escenarios de cadenas de suministros reales.

Trabajos futuros

Como resultado de esta tesina se logró concretar el desarrollo de un framework para la creación de sistemas de modelado y simulación de cadenas de suministros. Este framework puede extenderse aún más de lo que fue desarrollado. Extensiones posibles de este trabajo pueden ser:

- Extender el framework para darle la posibilidad de crear sistemas con manejo de usuarios registrados. Lograr que los sistemas desarrollados por el framework puedan contar con la funcionalidad de un manejo completo de usuarios, es decir, la registración de nuevos usuarios y el login de los mismos en el sistema.
- Explotar en mayor medida el frontend de los sistemas desarrollados por el framework, utilizando por ejemplo geolocalización para los agentes de las cadenas, brindando así una mejor experiencia para el usuario y un mayor detalle geográfico de la disposición de las cadenas de suministros distribuidas, o agregar la posibilidad de que el usuario del sistema pueda responder en tiempo real a los diferentes tipos de mensajes que recibe un agente de la cadena, logrando así una mayor interacción con el usuario, también la posibilidad de actualizar el frontend en tiempo real, a medida que va ejecutando la simulación, para así poder apreciar mejor el flujo de mensajes, permitir ejecutar múltiples simulaciones en paralelo con un sólo click, entre otras.
- Utilizar por completo el framework ya implementado para modelar, simular y analizar una cadena de suministros más amplia y detallada, creando un sistema más complejo, por ejemplo con mayor número agentes y más sofisticados sistemas de toma de decisiones, conectividad con más y distintos sistemas externos, explotando las múltiples métricas que se pueden crear en el servicio *CloudWatch* de Amazon, logrando así un ejemplo mucho más realista, detallado y potente, utilizando las posibilidades que el framework brinda.
- Complementar el framework con una herramienta de depuración para facilitar la detección y depuración de errores. Al ejecutarse en un ambiente en la nube es muy difícil poder seguir qué es lo que está sucediendo y dónde está fallando, por lo tanto una herramienta de depuración que se encargue de realizar búsquedas en los logs de la nube ayudaría a la detección de errores.

Capítulo 9

Anexos

Diagrama de clases

Diagrama de clases del framework SCS desarrollado en esta tesina de grado. En él se muestran las diferentes clases que conforman el framework. La clase *Chain* que está compuesta por un conjunto de objetos de la clase *Agent,* los cuales poseen un conjunto de objetos de clase *Conversation.* Esta clase contiene objetos de la clase *Message* y esta hereda su comportamiento a sus subclases *Refuse, CallForProposal, Propose, AcceptProposal, RejectProposal, InformDone, InformResult* y *Failure.*

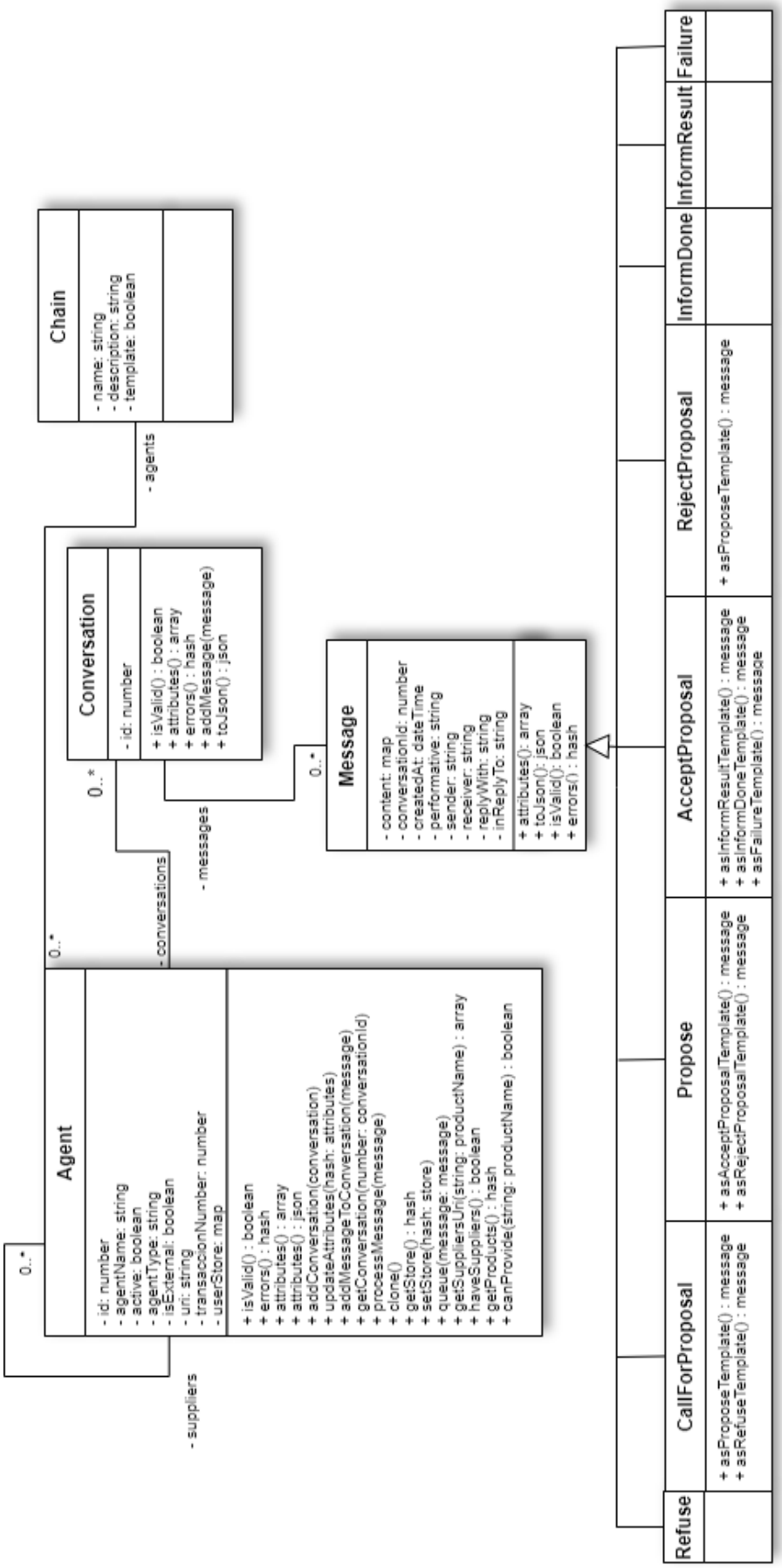

Ilustración 70. Diagrama de clases del framework SCS.

Casos de uso

Un total de 13 casos de uso que ejemplifican el flujo de ejecución del framework SCS.

CU-01	Hacer nuevo pedido (CFP).	
Versión	4 (29/03/18)	
Dependencias	-	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente cliente realiza un pedido a uno de sus Agentes proveedores.	
Precondición	El Agente cliente tiene una lista de sus proveedores, en donde sabe que elemento le provee cada uno de ellos.	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente cliente busca en su lista de proveedores quién puede atender su pedido.
	2	El Agente cliente crea un nuevo pedido.
	3	El Agente cliente envía el pedido al agente proveedor elegido.(CFP)
Postcondición	El agente cliente inició una nueva conversación.	
Excepciones	**Paso**	**Acción**
	1	Si el Agente cliente no tiene ningún agente proveedor que atienda su pedido termina la ejecución.
Comentarios		

Tabla 3. Hacer nuevo pedido

CU-02	Recibir un pedido (CFP).	
Versión	5 (13/06/19)	
Dependencias	● CU-01 Hacer nuevo pedido.	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente proveedor recibe un pedido (CFP) hecho por uno de sus Agentes clientes, el agente se encarga de buscar el stock para dicho pedido.	
Precondición	El Agente cliente envió un pedido al Agente proveedor.	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente proveedor recibe el pedido enviado por el Agente cliente
	2	El Agente proveedor tiene stock para el pedido
	3	El Agente proveedor cuenta con todo lo necesario para el pedido. Se ejecuta el CU-03.
Postcondición	El Agente proveedor encontró el stock necesario	
Excepciones	**Paso**	**Acción**
	2	El Agente proveedor no tiene stock para el pedido.
	3	El Agente proveedor ejecuta CU-01.
	2	El Agente proveedor no puede responder al pedido del Agente cliente. Se ejecuta el CU-07
Comentarios		

Tabla 4. Recibir un pedido

CU-03	Enviar propuesta (Propose).
Versión	5 (13/06/19)

Dependencias	● CU-02 Recibir un pedido.	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente proveedor tiene stock para responder a un CFP realizado por un Agente cliente	
Precondición	El Agente proveedor cuenta con todo lo necesario para el pedido.	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente proveedor crea una nueva propuesta (Propose).
	2	El Agente proveedor envía el Propose al Agente cliente.
Postcondición	Se envió una propuesta al Agente cliente.	
Excepciones	**Paso**	**Acción**
Comentarios		

Tabla 5. Enviar propuesta

CU-04	Recibir y analizar propuesta (Propose).	
Versión	5 (13/06/19)	
Dependencias	● CU-03 Enviar propuesta.	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente cliente recibe una oferta de su Agente proveedor y la analiza.	
Precondición	El Agente proveedor envió una oferta al Agente cliente.	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente cliente recibe la oferta (Propose) enviada por el Agente proveedor.
	2	El Agente cliente analiza la oferta.
	3	La oferta es la mejor, entonces el Agente cliente acepta la oferta.
	4	Se ejecuta el CU-05.
Postcondición	El Agente cliente contestó la oferta al Agente proveedor, aceptando la mejor oferta recibida.	
Excepciones	**Paso**	**Acción**
	3	El Agente cliente ve que la oferta no es la mejor.
	4	Se ejecuta el CU-06.
Comentarios		

Tabla 6. Recibir y analizar propuesta

CU-05	Enviar aceptación (AcceptProposal).	
Versión	5 (13/06/19)	
Dependencias	● CU-04 Recibir y analizar propuesta.	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente cliente acepta la oferta del Agente proveedor.	
Precondición	El Agente proveedor envió la mejor oferta, entonces se acepta.	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente cliente crea una aceptación (AcceptProposal) de la propuesta enviada por su proveedor.
	2	El Agente cliente envía un AcceptProposal al Agente proveedor.
Postcondición	Se aceptó la propuesta (Propose) del Agente proveedor	

Excepciones	**Paso**	**Acción**
Comentarios		

Tabla 7. Enviar aceptación

CU-06	Enviar rechazo (RejectProposal).	
Versión	5 (13/06/19)	
Dependencias	● CU-04 Recibir y analizar oferta.	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente cliente rechaza la propuesta enviada por su proveedor.	
Precondición	El Agente proveedor envió una oferta que no es la mejor.	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente cliente crea un rechazo (RejectProposal) de la propuesta enviada por su proveedor.
	2	El Agente cliente envía un RejectProposal al Agente proveedor.
Postcondición	Se rechazó una propuesta.	
Excepciones	**Paso**	**Acción**
Comentarios		

Tabla 8. Enviar rechazo

CU-07	Enviar negación (Refuse).	
Versión	5 (13/06/19)	
Dependencias	● CU-01 Hacer nuevo pedido.	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente proveedor se niega al pedido del Agente cliente.	
Precondición	El Agente proveedor no puede cumplir el pedido hecho por su cliente.	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente proveedor crea una negación (Refuse).
	2	El Agente proveedor envía una negación (Refuse) a su cliente.
Postcondición	Se cerró la conversación.	
Excepciones	**Paso**	**Acción**
Comentarios		

Tabla 9. Enviar negación

CU-08	Recibir y analizar aceptación (AcceptProposal).	
Versión	5 (13/06/19)	
Dependencias	● CU-05 Enviar aceptación.	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente proveedor analiza si puede cumplir con la propuesta que ha hecho a su cliente.	
Precondición	El Agente proveedor recibe un AcceptProposal de su cliente.	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente proveedor analiza si puede cumplir con la propuesta que hizo a su cliente.
	2	El Agente proveedor puede cumplir con la propuesta.

	3	Se ejecuta el CU-09
Postcondición	El Agente proveedor puede cumplir con la oferta enviada a su cliente.	
Excepciones	**Paso**	**Acción**
	2	El Agente proveedor no puede cumplir con la propuesta.
	3	Se ejecuta el CU-10
Comentarios		

Tabla 10. Recibir y analizar aceptación

CU-09	Informar pedido realizado (InformDone).	
Versión	5 (13/06/19)	
Dependencias	● CU-08 Recibir y analizar aceptación.	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente proveedor realiza el pedido solicitado por su cliente.	
Precondición	El Agente proveedor puede cumplir con la propuesta enviada a su cliente	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente proveedor crea un informe de pedido realizado (InformDone).
	2	El Agente proveedor descuenta su stock disponible.
	3	El Agente proveedor envía un InformDone a su cliente.
Postcondición	El Agente proveedor avisó a su cliente que el pedido está hecho y descuenta su stock y cierra la conversación.	
Excepciones	**Paso**	**Acción**
Comentarios		

Tabla 11. Informar pedido realizado

CU-10	Enviar fallo (Failure).	
Versión	5 (13/06/19)	
Dependencias	● CU-08 Recibir y analizar aceptación.	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente proveedor le envía un fallo a su Agente cliente	
Precondición	El Agente proveedor no puede cumplir con la propuesta enviada a su cliente	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente proveedor crea un Failure.
	2	El Agente proveedor envía un Failure a su cliente
Postcondición	El Agente proveedor avisó a su cliente que el pedido no se va a hacer y cierra la conversación	
Excepciones	**Paso**	**Acción**
Comentarios		

Tabla 12. Enviar fallo

CU-11	Recibir negación (Refuse).
Versión	5 (13/06/19)
Dependencias	● CU-07 Enviar negación.
Autores	Esteban Sanchez, Pankow Matias.
Descripción	El Agente cliente recibe la negación de su proveedor y analiza qué hacer

Precondición	Se envía una negación (Refuse) al Agente cliente.	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente cliente recibe un Refuse.
	2	El Agente cliente analiza si es proveedor de otro Agente.
	3	El Agente no es proveedor de ningún otro Agente.
Postcondición	Se cerró la conversación.	
Excepciones	**Paso**	**Acción**
	3	El Agente es proveedor de otro Agente
	4	Se ejecuta el CU-07
Comentarios		

Tabla 13. Recibir negación

CU-12	Recibir informe de pedido realizado (InformDone).	
Versión	5 (13/06/19)	
Dependencias	● CU-09 Informar pedido realizado.	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente cliente recibe el pedido solicitado.	
Precondición	El Agente proveedor informa a su cliente que el pedido está listo.	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente cliente recibe el InformDone.
	2	El Agente cliente incrementa su stock.
	3	El Agente cliente analiza si es proveedor de otro Agente
	4	El Agente cliente no es proveedor de ningún otro Agente
Postcondición	Se cierra la conversación.	
Excepciones	**Paso**	**Acción**
	4	El Agente es proveedor de otro Agente
	5	El Agente cliente analiza si recibió todos los InformDone que necesita para armar la oferta a su cliente.
	5.1	El Agente cliente recibió todos los InformDone que estaba esperando.
	5.2	Se ejecuta el CU-03.
Comentarios		

Tabla 14. Recibir informe de pedido realizado

CU-13	Recibir fallo (Failure).	
Versión	5 (13/06/19)	
Dependencias	● CU-10 Enviar fallo.	
Autores	Esteban Sanchez, Pankow Matias.	
Descripción	El Agente cliente recibe un fallo de su proveedor y lo analiza.	
Precondición	El Agente proveedor envía un fallo a su cliente.	
Secuencia Normal	**Paso**	**Acción**
	1	El Agente cliente recibe un Failure de su proveedor
	2	El Agente cliente analiza si es proveedor de otro Agente.
	3	El Agente no es proveedor de ningún otro Agente

Postcondición	Se cierra la conversación.	
Excepciones	**Paso**	**Acción**
	3	El Agente es proveedor de otro Agente.
	4	Se ejecuta el CU-07.
Comentarios		

Tabla 15. Recibir fallo

Diagramas de flujo

Un total de 13 diagramas de flujo creados a partir de los casos de uso antes mencionados.

CU-01 Hacer nuevo pedido (CFP)

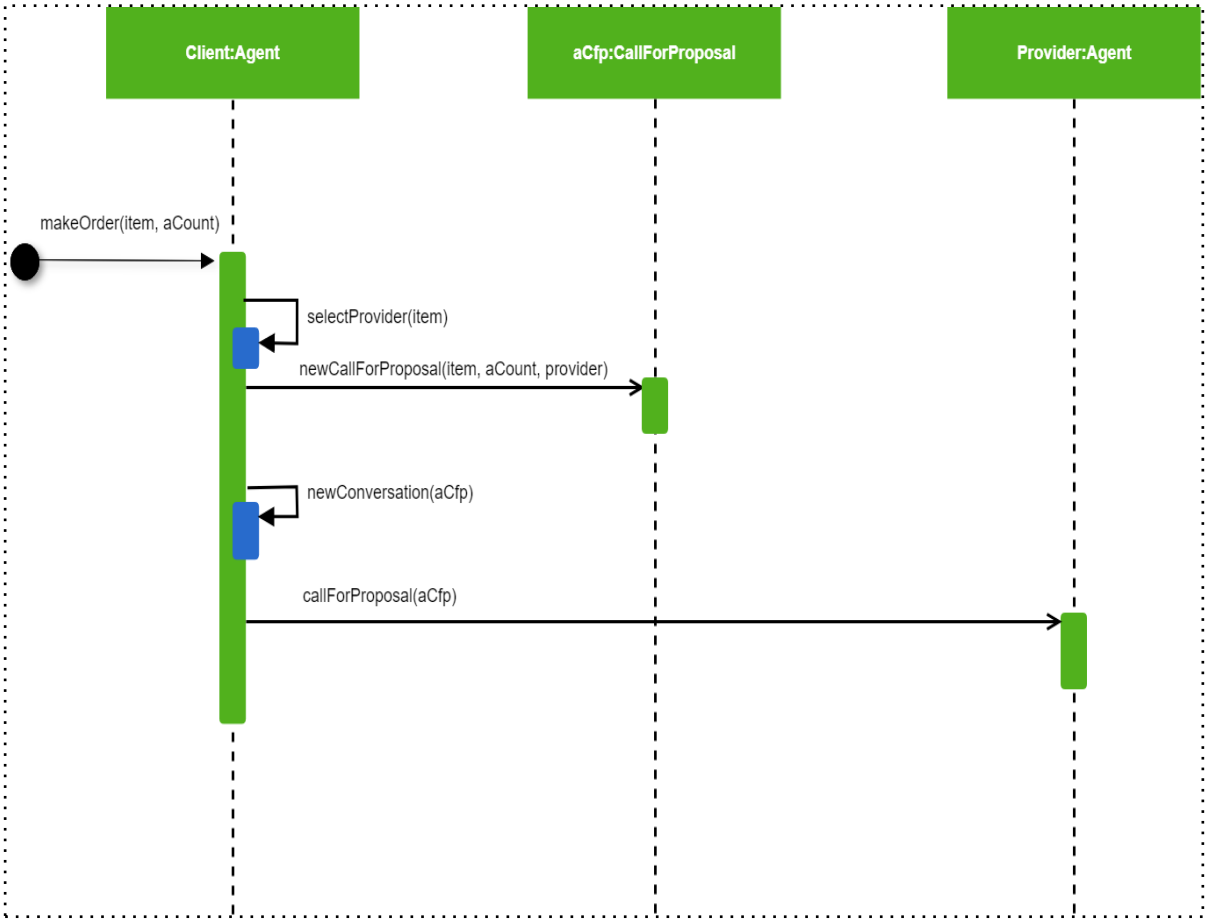

Ilustración 71. Hacer nuevo pedido

CU-02 Recibir un pedido (CFP)

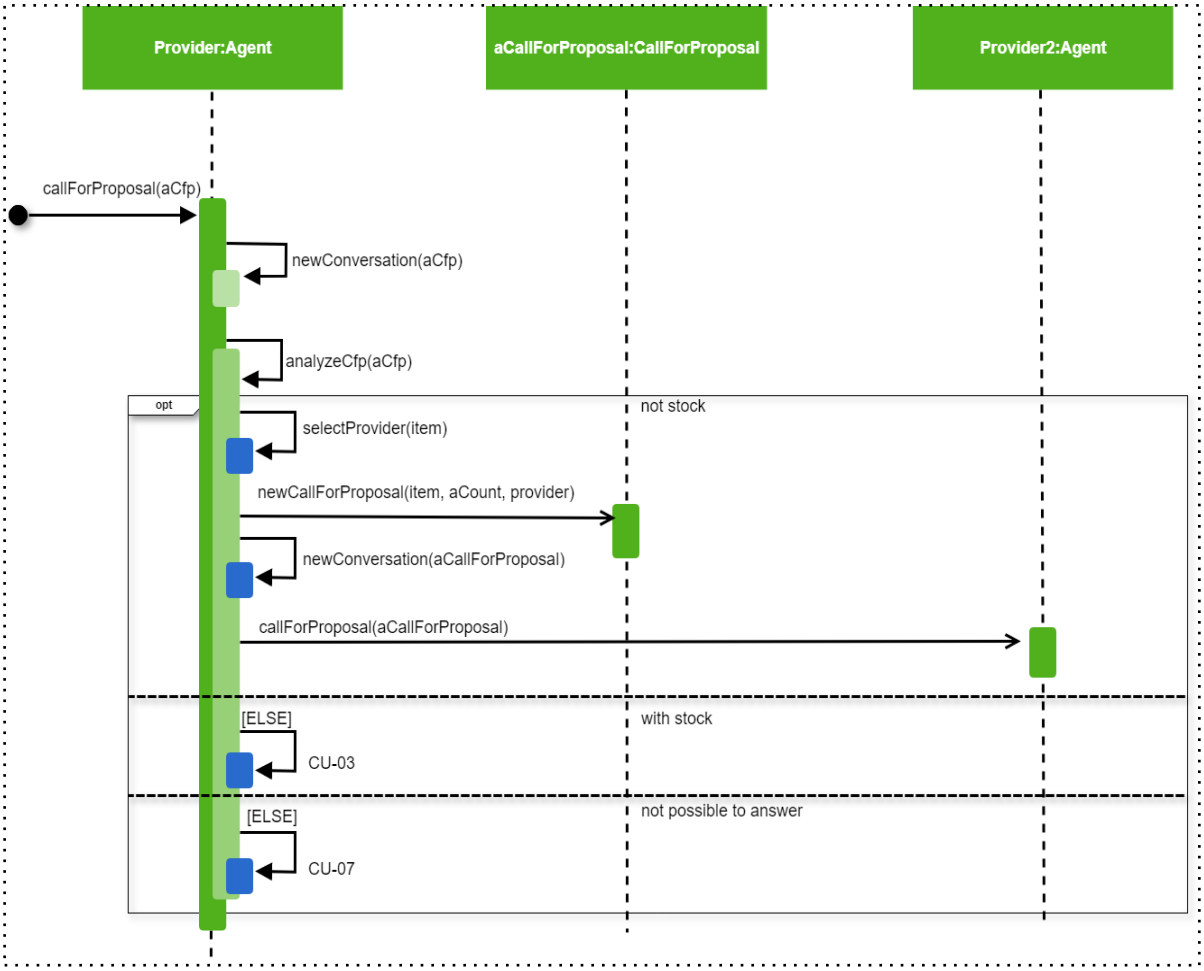

Ilustración 72. Recibir un pedido

CU-03 Enviar propuesta (Propose)

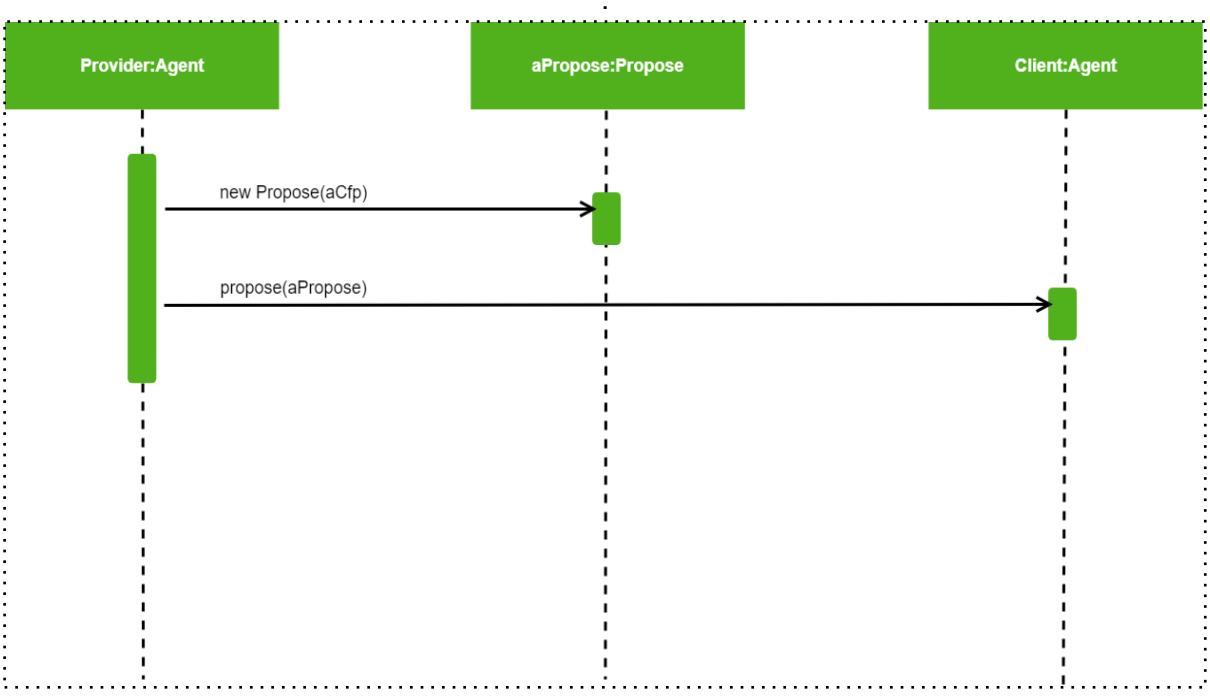

Ilustración 73. Enviar propuesta

CU-04 Recibir y analizar propuesta (Propose)

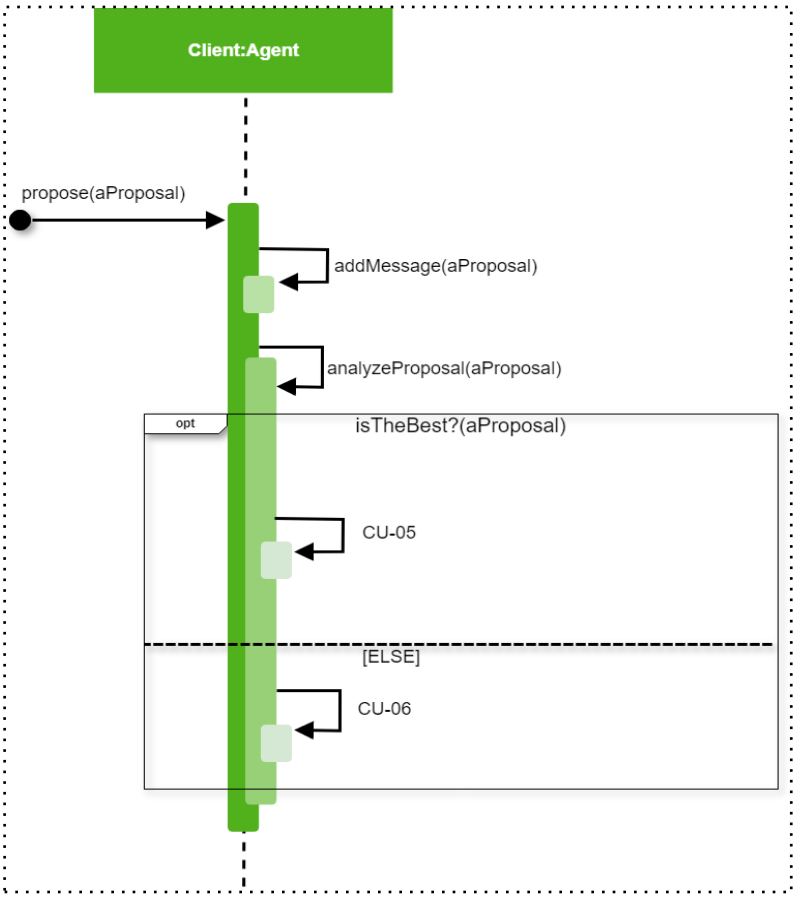

Ilustración 74. Recibir y analizar propuesta

CU-05 Enviar aceptación (AcceptProposal)

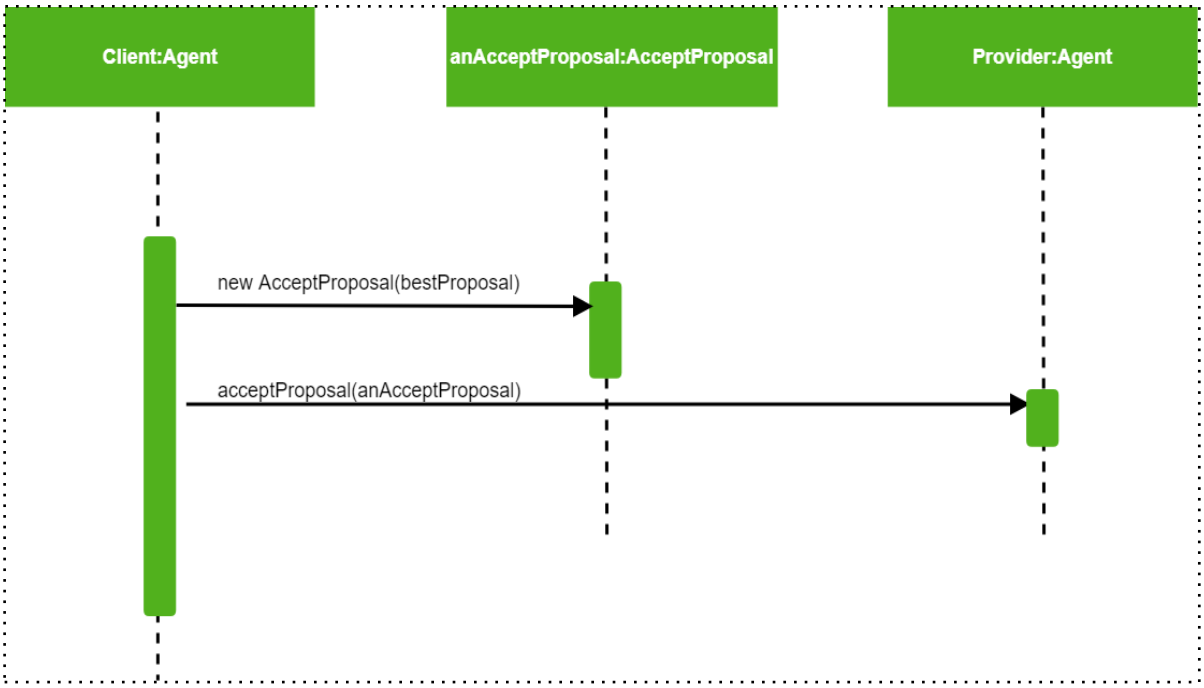

Ilustración 75. Enviar aceptación

CU-06 Enviar rechazo (RejectProposal)

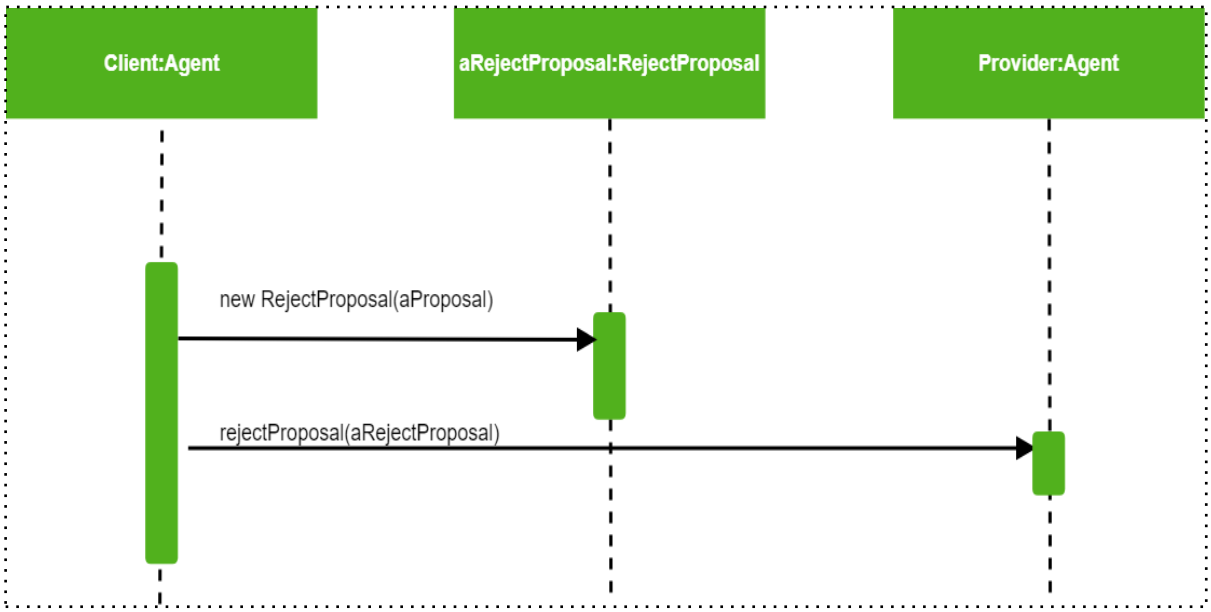

Ilustración 76. Enviar rechazo

CU-07 Enviar negación (Refuse)

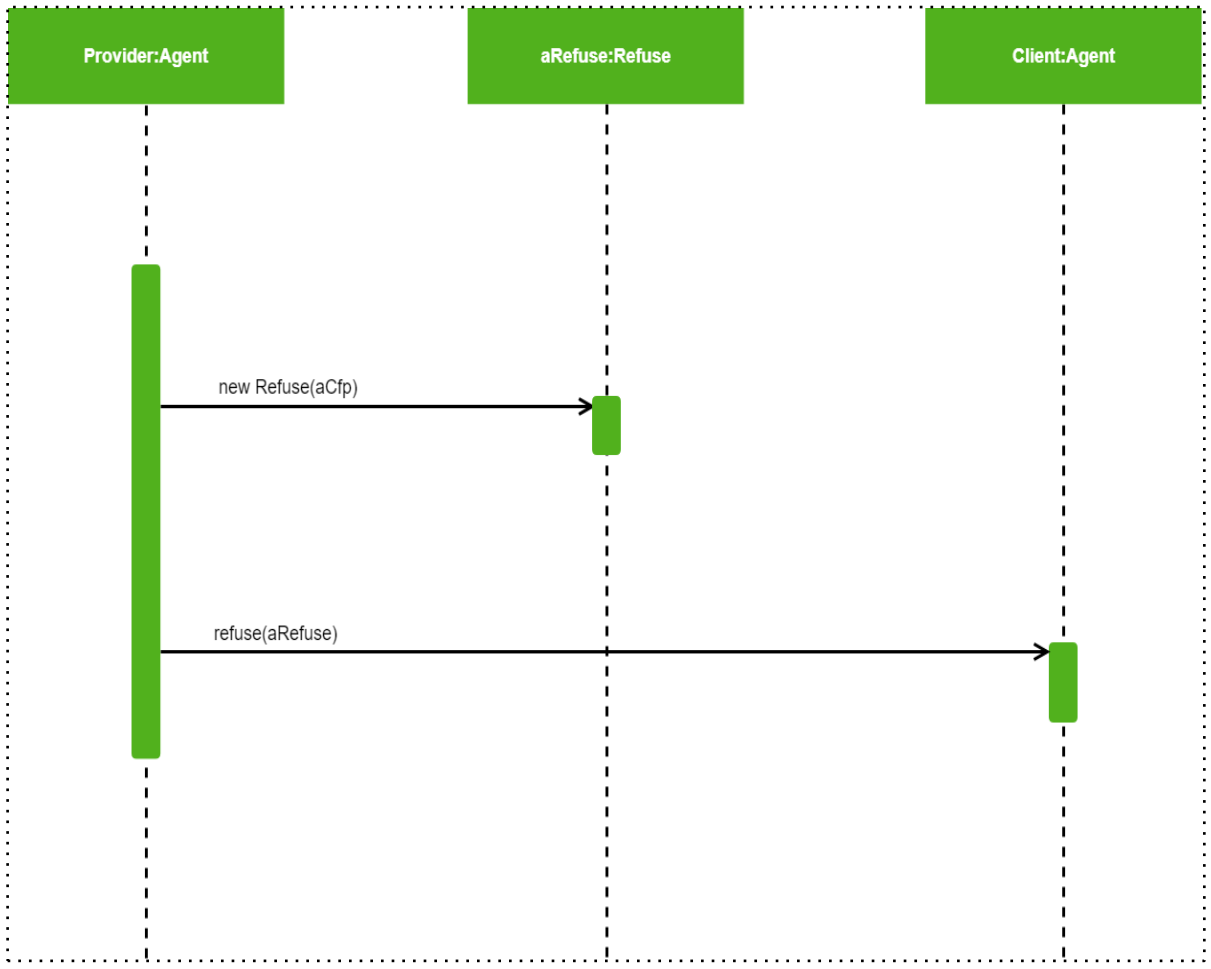

Ilustración 77. Enviar negación

CU-08 Recibir y analizar aceptación (AcceptProposal)

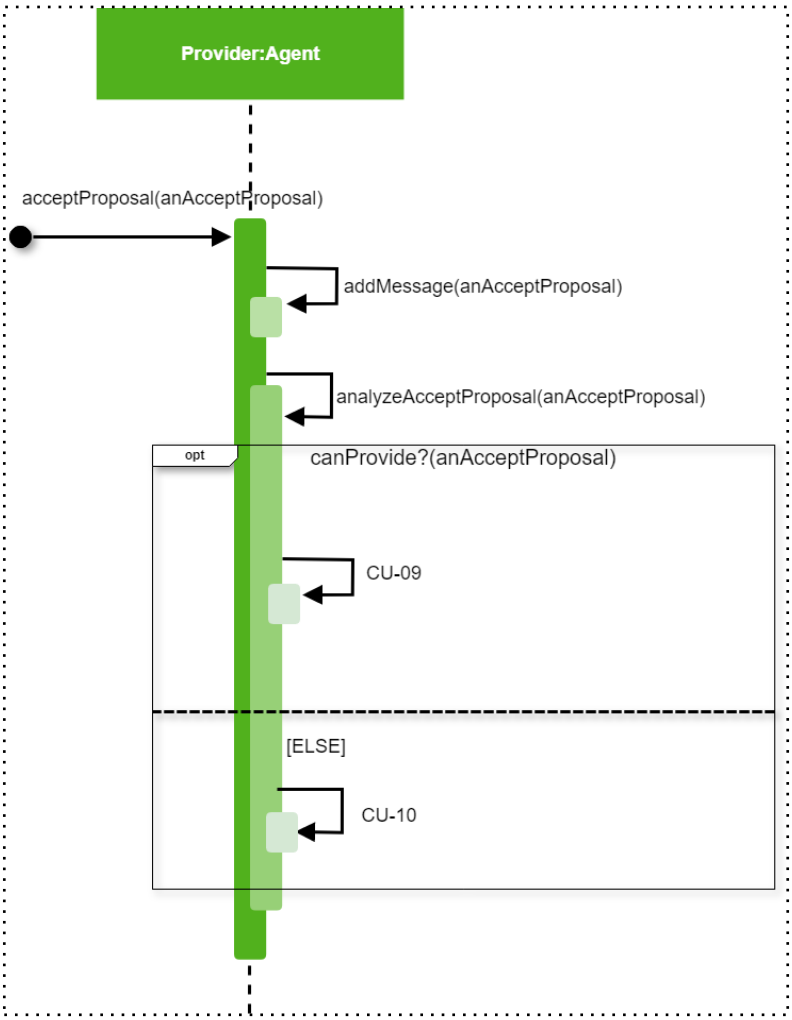

Ilustración 78. Recibir y analizar aceptación

CU-09 Informar pedido realizado (InformDone)

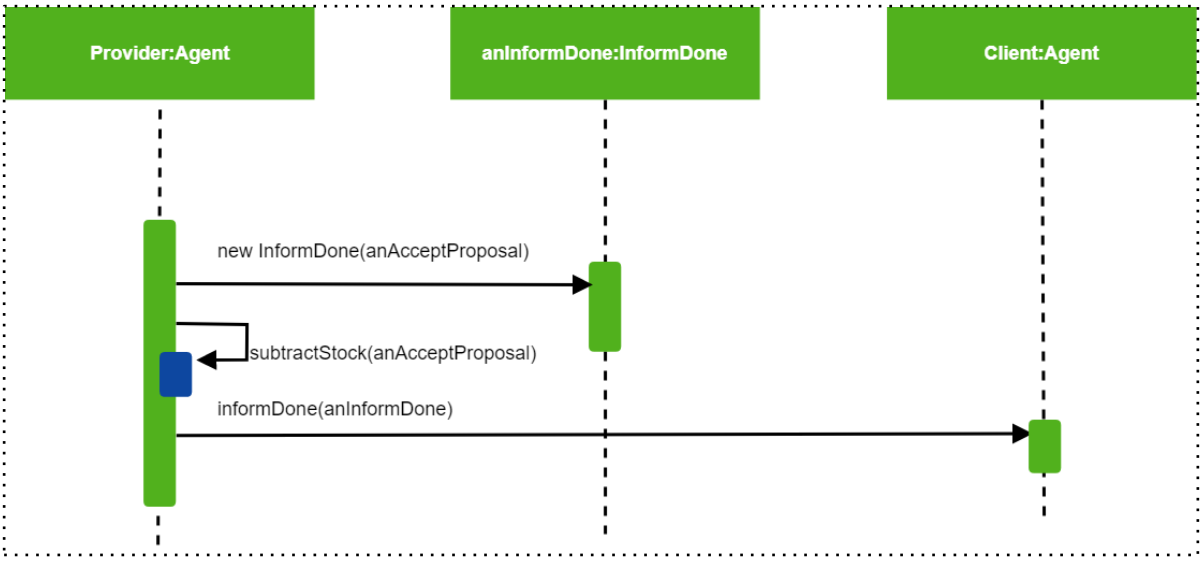

Ilustración 79. Informar pedido realizado

CU-10 Enviar fallo (Failure)

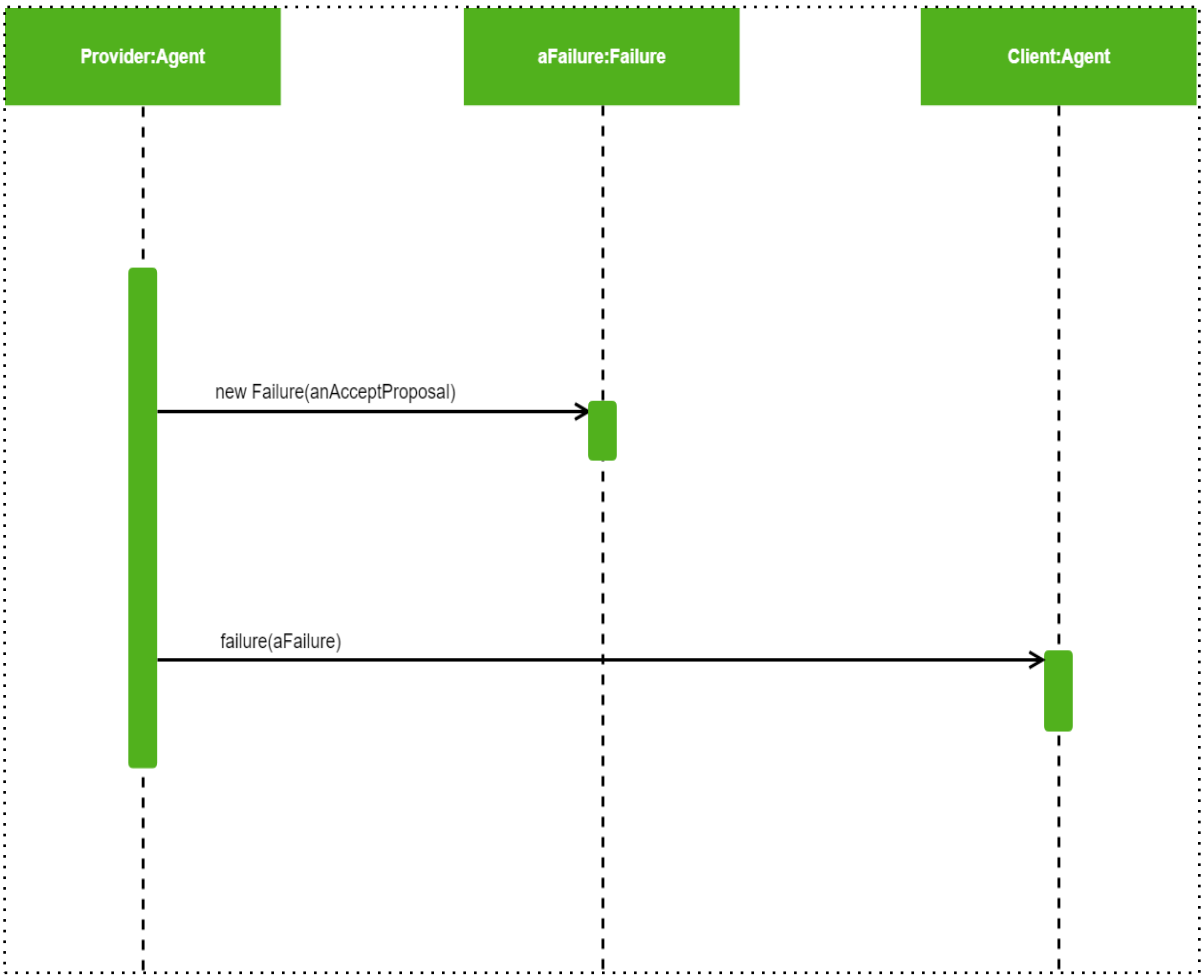

Ilustración 80. Enviar fallo

CU-11 Recibir negación (Refuse)

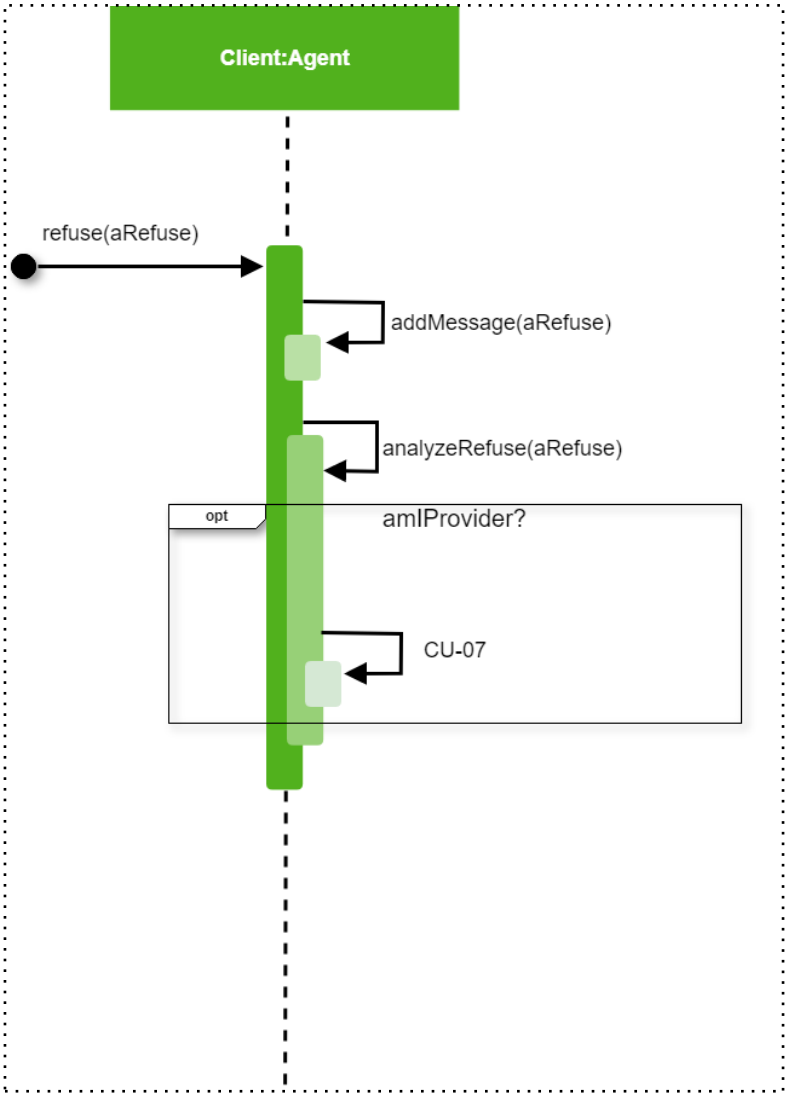

Ilustración 81. Recibir negación

CU-12 Recibir informe de pedido realizado (InformDone)

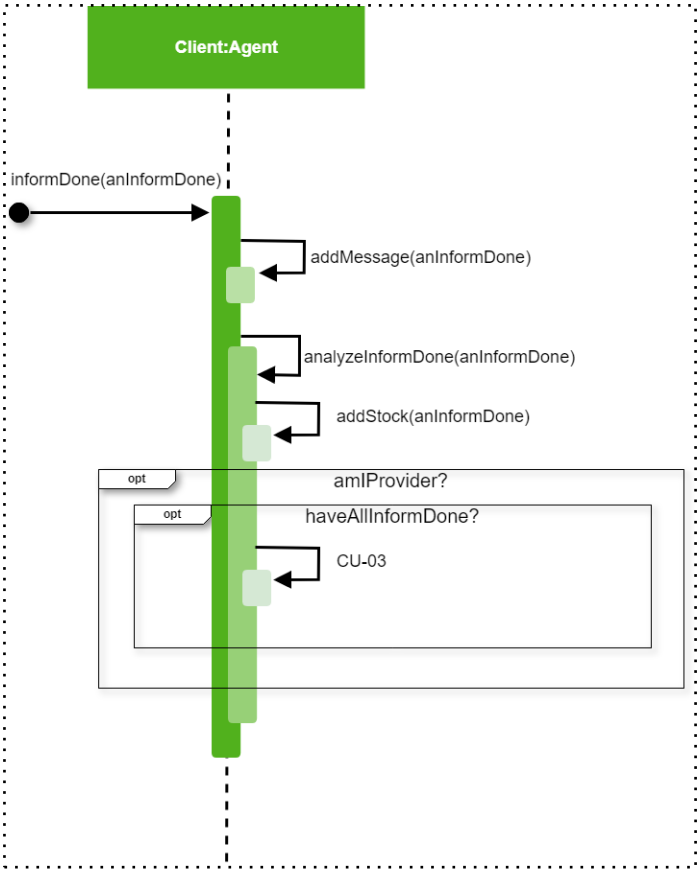

Ilustración 82. Recibir informe de pedido realizado

CU-13 Recibir fallo (Failure)

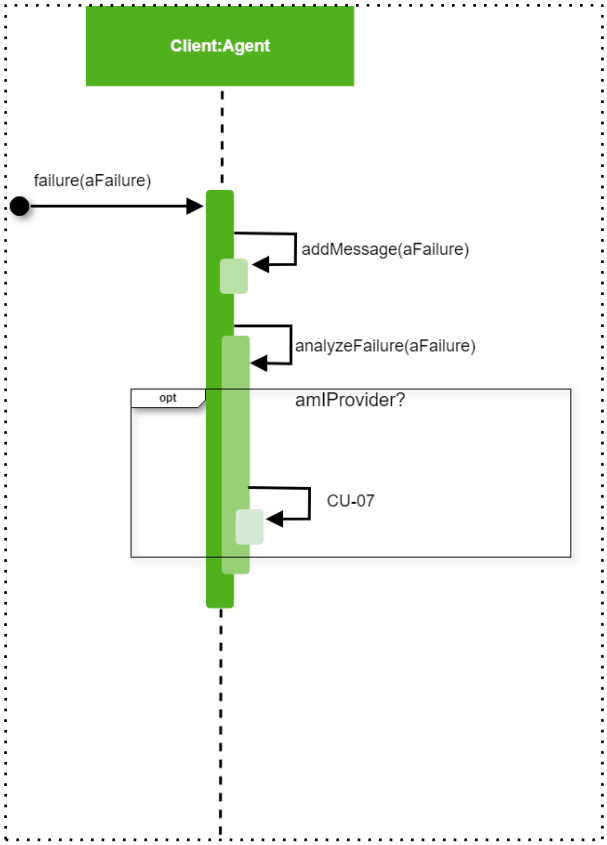

Ilustración 83. Recibir fallo

Capítulo 10

Referencias y Bibliografía

David Jacoby, “The Economist Guide To Supply Chain Management”. Economist Books, 2009.

La Londe, Bernard J. and James M. Masters. “Emerging Logistics Strategies: Blueprints for the Next Century”. International Journal of Physical Distribution and Logistics Management, 1994, Vol. 24, No. 7, pp. 35-47.

Christopher, Martin L. “Logistics and Supply Chain Management”. London: Pitman Publishing, 1992.

Selic, B. “The pragmatics of model-driven development”. IEEE Softw. 20(5), September 2003, 19–25.

Pérez Porto J. y Merino M. (2011). “Definición de simulación”. Retrieved from https://definicion.de/simulacion

Parsons D., Rashid A., Telea A.,Speck A. “An architectural pattern for designing component- based application frameworks”. Published online 22 November 2005 in Wiley InterScience.

Savage, N., Going serverless. Communications of the ACM, 61(2), 2018, 15–16.

World Wide Web Consortium & Internet Engineering Task Force (1999), RFC 2616.

Roy Thomas Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. University of California, Irvine 2000.

RFC 7231 (2014, June). Retrieved from https://tools.ietf.org/html/rfc7231.

Foundation For Intelligent Physical Agents. (2002, December 3). ”Especificación FIPA ACL Message Structure” Retrieved from http://www.fipa.org/specs/fipa00061/SC00061G.html.

Foundation For Intelligent Physical Agents. (2002, December 3). “FIPA Communicative Act Library Specification”. Retrieved from http://www.fipa.org/specs/fipa00037/SC00037J.html.

Foundation For Intelligent Physical Agents. (2002, December 3). “Especificación FIPA Contract Net Interaction Protocol”. Retrieved from http://www.fipa.org/specs/fipa00029/SC00029H.html.

Alan Beaulie. “Learning SQL”. O’Reilly Media, 2009.

Jamison, D. C. “Structured Query Language (SQL) Fundamentals”. Current Protocols in Bioinformatics, 00(1), 9.2.1–9.2.29, 2003.

Li, Y., & Manoharan, S. “A performance comparison of SQL and NoSQL databases”. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), 2013.