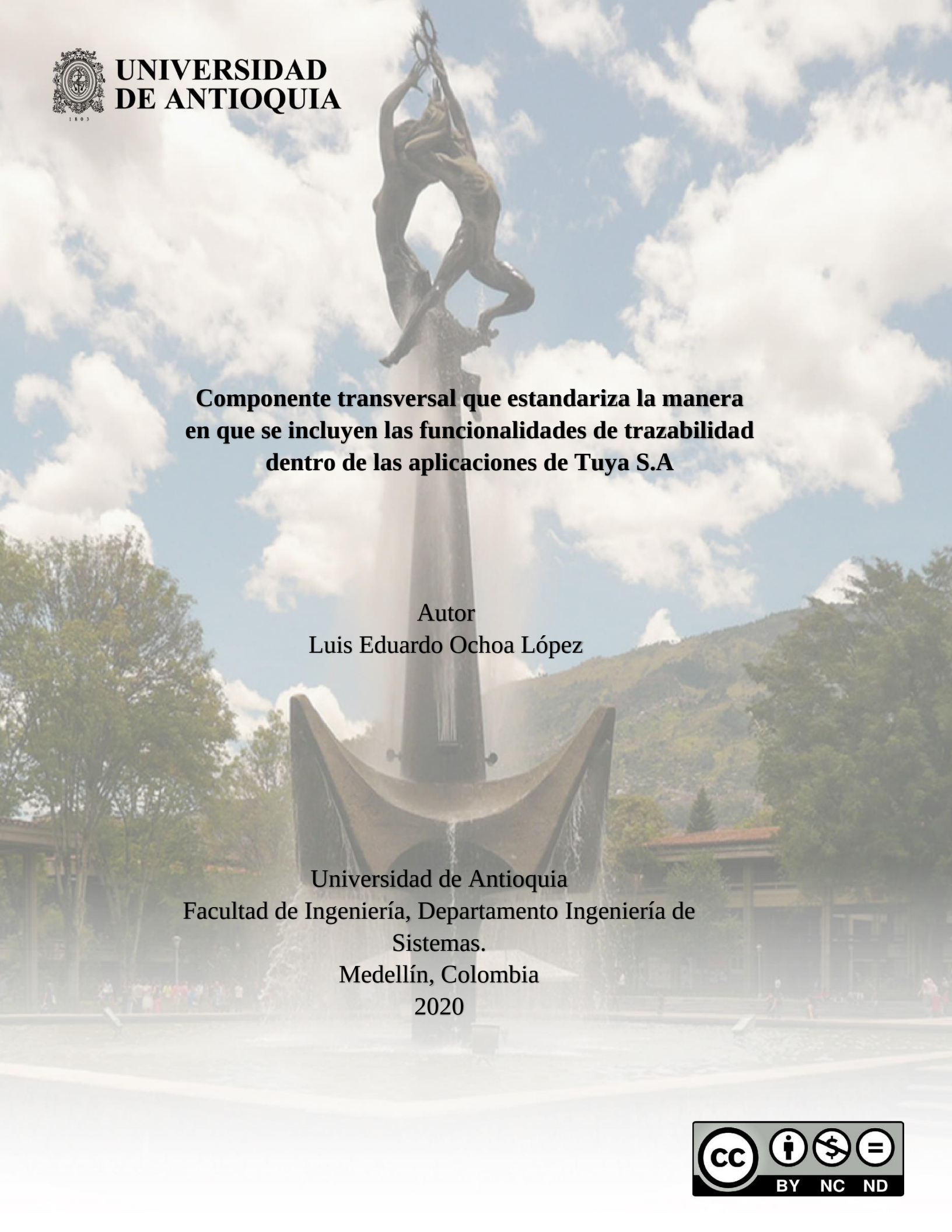

UNIVERSIDAD DE ANTIOQUIA

# Componente transversal que estandariza la manera en que se incluyen las funcionalidades de trazabilidad dentro de las aplicaciones de Tuya S.A


Autor
Luis Eduardo Ochoa López

Universidad de Antioquia
Facultad de Ingeniería, Departamento Ingeniería de Sistemas.
Medellín, Colombia

2020

# Componente transversal que estandariza la manera en que se incluyen las funcionalidades de trazabilidad dentro de las aplicaciones de Tuya S.A

**Luis Eduardo Ochoa López**

Tesis o trabajo de investigación presentada(o) como requisito parcial para optar al título de:

**Ingeniería de Sistemas**

Asesores (a):

Deisy Loaiza Berrío, Ingeniera de Sistemas

Universidad de Antioquia

Facultad de Ingeniería, Departamento Ingeniería de Sistemas.

Medellín, Colombia

2020.

**Componente transversal que estandariza la manera en que se incluyen las funcionalidades de trazabilidad dentro de las aplicaciones de Tuya S.A**

## Resumen

En Tuya se tiene un componente de trazabilidad *on premise* que se ha utilizado para a recolección de acciones e interacciones con los diferentes canales que expone la compañía. Este componente se ha venido convirtiendo en una carga debido a su obsolescencia, ubicación y a la carga que soporta. Se planteó como solución un componente repotenciado en nube con características que permitan la mantenibilidad y estabilidad del componente para este proceso tan importante, incluyendo pruebas de concepto y utilización de manejadores de colas que dejan una base para una futura migración a una arquitectura basada en microservicios en la compañía.

Al término de esta práctica, se obtiene una solución conformada por tres componentes que permiten la persistencia de datos de trazabilidad de los canales que lo usen. Dichos componentes alcanzaron a ser desplegados en el ambiente de desarrollo y pruebas. En dicho punto, los integrantes del equipo de QA y Ciberseguridad analizan los componentes en busca de posibles vulnerabilidades y problemas de implementación que puedan afectar su paso a producción. Además, se presenta un primer draft de la documentación de la solución.

## Introducción

En Tuya S.A, actualmente existen herramientas implementadas para el monitoreo de servicios y aplicativos, los cuales proveen información acerca de las interacciones que se efectúan contra las aplicaciones y los servicios actualmente utilizados en la compañía y los usuarios tanto internos o externos que los utilizan. Sin embargo, la manera en que se realizan dichos monitoreos es manual, incurriendo en malas prácticas que pueden llegar a acarrear tiempos de respuesta poco eficientes para el proceso en el que se implementa. Se planea realizar la arquitectura, diseño e implementación de un componente de trazabilidad transversal que pueda ser utilizado en cada proceso y servicio de la compañía. Dicho componente planea alinearse a estándares definidos por el equipo de Arquitectura de T.I de la compañía.

Este proyecto es definido de manera transversal, indicando que dicho componente será utilizado por diferentes áreas de la compañía buscando facilitar la creación de datos de trazabilidad de aplicativos con diversos fines.

Para el desarrollo de estos componentes se utilizó una metodología ágil siguiendo SCRUM, con sprints de 1 semana y aplicando el modelo de trabajo Flow to work, el cual consta del desplazamiento de integrantes de equipo hacia otros equipos con la finalidad de concluir el desarrollo del proyecto.

Para la realización de este componente se encontraron limitaciones a la hora de la integración con la infraestructura en la nube de Azure, debido a que la compañía en medio de la contingencia del Covid-19 enfocó sus esfuerzos en sacar a producción lo más rápido posible productos para mitigar las necesidad de sus clientes, haciendo así

que las demás implementaciones tuvieran retrasos en su entrega debido a que la obtención de permisos y aprovisionamiento de recursos en la nube tomó más tiempo de lo necesario. También a la hora de la implementación del código se encontraron limitaciones, debido a que, por políticas de despliegue en la compañía, se llegó a la decisión de re-implementar uno de los componentes que hacía parte de la solución.

Sin embargo, al término de la práctica, se pudo entregar una solución funcional desplegada en el ambiente de pruebas y certificación, para su adecuación final antes de dar el paso a producción.

## Objetivos

Objetivo General

Desarrollar un componente transversal que estandarice la manera en que se incluyen las funcionalidades de trazabilidad dentro de las aplicaciones de Tuya S.A

Objetivos Específicos

- Levantar historias técnicas y requisitos no-funcionales

- Formular arquitectura y diseño del componente.

- Definir estándares que debe cumplir el componente para homogeneizar su consumo.

- Definir criterios de medición que nos permitan evaluar el desempeño y porcentaje de éxito del proyecto.

- Construir un componente de tipo librería que pueda ser descargado y usado desde todos los proyectos.

- Evaluar el componente a la luz de los criterios de medición.

- Documentar la solución y las decisiones tomadas, para transferir el conocimiento de uso y soporte de la solución.

## Marco Teórico

Dado que el enfoque de este proyecto está orientado al desarrollo de un componente de trazabilidad, el desarrollo estuvo basado en el enfoque TDD [1] con el fin de generar un producto que haya sido desarrollado con buenas prácticas y produzca un componente de calidad, el desarrollo y las pruebas serán ejecutadas de forma paralela con la finalidad de mantener el nivel alto de calidad en todo momento del desarrollo.

En cuanto al tipo de trazabilidad manejado para este proyecto, será el denominado trazabilidad interna o trazabilidad de procesos [2], el cual se define como la implementación de procedimientos que nos permiten conocer cómo, cuándo y de qué

manera se está interactuando con nuestro sistema, además de recopilar y tener registros históricos que permitan ser consultados y procesados para generar valor sobre los datos [4]. Una de las definiciones que se tiene de las normas ISO, insta a que cada organización debería tener un sistema que identifique el estado de un servicio o producto y de los demás servicios que interactúen con este con la finalidad de darle cumplimiento a un requerimiento [6].

Se utilizará una metodología ágil basada en el marco de trabajo de SCRUM el cual brinda independencia al equipo, en este caso una persona, mediante la definición de consideraciones iniciales que se seguirán mediante el transcurso de un desarrollo iterativo llamado SPRINT [5].

Analizando las plataformas tecnológicas y/o servicios que serán implementados en este proyecto, tenemos que Azure además de ser un conjunto de servicios informáticos en la nube aprovisionado por Microsoft [3], es también el proveedor de servicios Cloud de Tuya S.A. Para ello, utilizaremos la herramienta Azure Artifacts que será donde se desplegará y estará contenido el componente, se utilizará el Azure API Management donde estará desplegado el API que expondrá la capacidad de consumir el componente realizado, además, el componente también estará comunicado con Azure SQL Database el cual contendrá el log de registros que serán persistidos.

Con respecto a la selección o cambios en las plataformas tecnológicas del proyecto, se utiliza el concepto de POC (Proof of concept). Una POC consiste en seleccionar una plataforma o solución tecnológica que podría solventar una necesidad del negocio, se investiga acerca de la herramienta o solución y se realiza una pequeña prueba funcional que será medida con unos criterios propuestos según el problema, al final se miden estos criterios con respecto a el performance que se tuvo de la prueba y se toma una decisión. [9]

La finalidad del proyecto es crear un componente reutilizable en todos los proyectos de la compañía, para esto, se creará un paquete en forma de Nugget que contenga todas las funcionalidades necesarias. Un Nugget es la forma en que Microsoft reúne en un paquete comprimido código e información acerca de él para que pueda ser compartido y reutilizado en otros proyectos, el Nugget puede ser tanto público como privado [10], en nuestro caso será una funcionalidad propia de Tuya

Se abordará la cultura DevOps, la cual consta de la continua unión entre equipos, prácticas y software con la finalidad de ser un equipo de alto rendimiento capaz de responder ante las necesidades del usuario, entregando productos de calidad de manera continua cumpliendo con metas propuestas, acelerando tiempos de ejecución de procesos y mejorando la mantenibilidad de los sistemas siguiendo buenas prácticas. [8]

## Metodología

El proyecto fue abordado mediante la metodología ágil bajo el marco de trabajo SCRUM [5], este marco de trabajo ágil permitió el autogestionamiento del equipo de trabajo (1 persona) mediante la definición de lineamientos para cada parte del proceso. Se realizará un desarrollo iterativo dividido en Sprints (Ver Figura 1). Se

estableció que el tiempo de un sprint sería de 4 semanas (Tiempo máximo recomendado) y al final de estos un release funcional del componente, las descripciones de las actividades a realizar en cada sprint fueron presentados en la tabla de cronograma de actividades propuesto.

Las actividades realizadas durante este proyecto fueron:

- Familiarización con plataformas tecnológicas de la compañía y modelo del negocio.

- Proposición y formación de ideas.

- Se realizaron POCs (Proof of concept) o pruebas de concepto iniciales para realizar acercamientos iniciales a las herramientas a utilizar.

- Realización de encuentros entre el equipo de arquitectura buscando hallar y seguir lineamientos planteados en la compañía.

- Definición de estándares que debía cumplir el componente para que cada consumo se haga de manera homogénea.

- Definición de los criterios de estandarización para la utilización de las herramientas de monitoreo.

- Realización de diagramas de componentes y despliegue para definir esquemas de conexión entre componentes.

- Definición de métodos de conexión con los otros servicios y plataformas utilizados en la compañía.

- Definición de criterios de medición que nos permitan evaluar el desempeño y porcentaje de éxito del proyecto.

- Desarrollo componentes que cumplan con los estándares definidos previamente.

- Estabilización del componente.

- Evaluación el componente a la luz de los criterios de medición.

- Realización de pruebas funcionales que nos permitan medir la tasa de éxito del proyecto.

- Release funcional y publicación en la plataforma de artefactos de la compañía.

- Despliegue del proyecto funcional para su utilización por parte de la compañía en los procesos internos.

## Resultados y análisis

### Familiarización, ideación, definición y diseño.

Para la creación de estos componentes se desarrollaron pruebas de concepto para familiarización y aplicación en ambientes productivos de la plataforma CloudAMQP, obteniendo así un prototipo funcional de publicador en mensajería de colas que posteriormente fue soporte del producto final. También se realizaron pequeñas pruebas con los diversos componentes que hacen parte de la infraestructura en la nube de la compañía, como parte de la familiarización con las plataformas.

También, a lo largo de la práctica, se realizaron diversos encuentros con el COE de Arquitectura, con la finalidad de brindar directrices e indicaciones de las reglas, definiciones, estructuras y lineamientos que debía seguir el proyecto. Una de las definiciones más importantes que se dieron en estos espacios, fue la definición del consumo y utilización del componente por parte de los interesados, ya que la finalidad de la construcción del componente era además de facilitar su uso, evitar cualquier delay o problema al servicio que lo llama.

### Refinación y mejoras.

En medio de estas sincronizaciones con el COE, el componente se vio reformulado en varias ocasiones, esto debido a que en un principio, se había decido crear un componente que pudiera ser importado en cualquiera de los proyectos que necesitasen almacenamiento de trazabilidad de sus interacciones y que directamente este componente se comunicara con la base de datos que se tenía dispuesta. Luego de esto, el componente pasó a ser reformulado utilizando un manejador de mensajería de colas, esto, ya que en la compañía se empezó a formular la iniciativa de hacer una migración hacia el trabajo sobre una arquitectura basada en microservicios. Siendo el manejador de colas su componente principal. Este proyecto, serviría como prueba piloto para implementar dicha arquitectura.

### Diagramación.

Luego de tener las definiciones, se procedió a realizar los diagramas de componentes y despliegue que mapean la estructura final del proyecto. Para esto, se tuvo en cuenta qué componentes de infraestructura habían sido definidos en el proyecto y sobre qué proveedor de nube serían desplegados. Para este proyecto, se trabajó bajo el proveedor de servicios en la nube Azure, utilizando diversos servicios ofertados, tales como, Azure Kubernetes Service, Azure App Service, Azure API Manager, Azure Key Vault, entre otros. Para

información un poco más detallada, se puede consultar el anexo de este documento.

**Pruebas de concepto.**

Luego de tener clara la estructura del proyecto y cómo se conectan sus componentes, se procedió a realizar unos mini proyectos y pruebas de concepto funcionales con el fin de familiarizarse con los servicios del proveedor de nube. Probando así en entornos de pruebas el funcionamiento y conexión contra el Azure KeyVault, el consumo de servicios desplegados en un API manager y el despliegue de un componente en AKS.

**Aprovisionamiento e impedimentos.**

Con la información previa, se procedió a gestionar el aprovisionamiento de los repositorios de los distintos componentes que hacen parte de la solución y también, el proceso de creación de las configuraciones del pipeline de despliegue de la solución. Además, se empezó con la gestión del aprovisionamiento de los componentes de infraestructura en nube, tales como unidades de despliegue como AKS y APIM, tanto como las bases de datos y message broker en los distintos ambientes.

En este punto del proyecto, se hallaron una gran cantidad de problemas relacionados al aprovisionamiento y concesión de permisos. En el aprovisionamiento, me encontré con el problema de que las instancias de bases de datos que me fueron entregadas para el proyecto, fueron creadas en un resource group distinto al del proyecto, esto causó que la comunicación entre el manejador de colas y la base de datos no se pudiera dar a causa de permisos y el debug realizado para encontrar este error tomó una gran cantidad de tiempo.

Otro de los problemas que se presentaron en este proyecto, estuvo relacionado con el manejador de colas. Como se explicó anteriormente, esta iniciativa de comenzar a implementar una arquitectura de microservicios se dio al mismo tiempo que la definición de este proyecto, por lo cual, las definiciones y estándares de uso de los dos proyectos se construyeron en simultáneo, haciendo que este proyecto sufriera retrasos debido a que no se presentó un estándar de uso y nombramiento de componentes.

Una de las problemáticas más fuertes y que consumió más tiempo en solucionar fue la obtención de un usuario de servicio y permisos para lectura y escritura sobre bases de datos. Debido a las políticas internas de seguridad en la compañía, este proceso tomó mucho tiempo para ser aprobado y, una vez obtenida la aprobación, se evidenció que al usuario de servicio le fueron otorgados permisos a una base de datos distinta a la utilizada en este proyecto. Además, fue necesario solicitar los permisos alrededor de 4 veces

para que los permisos necesarios para el proyecto fueran otorgados. Esta problemática se presentó, debido a que la persona encargada de la gestión de dicho usuario y permisos, era una persona que no contaba con un conocimiento técnico y había sido asignada a dicho rol hace poco tiempo.

**Adaptación y refactorización.**

En el desarrollo de este proyecto, también nos encontramos con un bache técnico, debido a que uno de los 3 componentes que se tuvieron que construir para la solución, fue creado como un console app, un tipo de demonio cuya funcionalidad consiste en escuchar constantemente el manejador de colas y disparar un evento cada vez que llegue un mensaje nuevo a la cola que se está escuchando. El problema aquí fue que desde el equipo de DevOps se me notificó que en la compañía con la infraestructura actual, no era posible desplegar un aplicativo tipo console app para el ambiente productivo, por lo que sería necesario un completo refactor del componente tipo demonio para convertirlo en un web application que sí se puede desplegar en un ambiente productivo. Esto trajo consigo una investigación corta de la manera en la cual se puede convertir un console app a un web application que escucha constantemente eventos. Como una pequeña nota en esta problemática, al final de la práctica y en conversaciones con el Líder de Arquitectura y asesor externo de mi práctica, hallamos que sí era posible el despliegue de la solución como un console app y la situación que aconteció fue catalogada como negligencia técnica.

**Desarrollo e integración.**

Con todo lo anterior, se procedió al desarrollo de uno de los componentes que hacen parte de la solución, el nugget. Un nugget en .net es una librería que se puede importar para hacer uso de sus funcionalidades internas en cualquier proyecto que lo importe. Este nugget, se encargará de la conexión con la instancia de CloudAMQP y publicará los mensajes que reciba como parámetro en dicha instancia. Además de hacer parte de la solución del proyecto de trazabilidad, se optó por desarrollar el nugget de una manera que pueda ser usado para comunicarse con la instancia de CloudAMQP en cualquier proyecto que sea distinto dicha iniciativa de trazabilidad.

Luego, se procedió a la construcción de un API que recibe las peticiones entrantes de los interesados en publicar mensajes en colas. Esta API, recibe el payload que se desea publicar y persistir en colas y bases de datos, el API incluye el nugget previo para configurar su destino y enviar el payload codificado para su posterior procesamiento.

Por último, se construyó el componente encargado de crear una conexión con la instancia de CloudAMQP y esperar mensajes que lleguen a la instancia que está escuchando para comenzar con el procesamiento, decodificar el

mensaje y abrir la conexión con la base de datos para proceder con el guardado de la información si cumple con la estructura definida del mensaje.

Para la realización de esta solución, se tuvo que tener varias sesiones de estabilización de los componentes, esto debido a problemas con los permisos y scopes definidos al inicio, ya que estos, impedían la conexión entre los diferentes componentes desplegados en la nube, y se hacía muy difícil su localización y solución.

Al momento de la redacción de este documento, el componente se encuentra desplegado en ambiente de pruebas, ad portas de pasar al ambiente de certificación, lamentablemente en mi tiempo en la compañía no se pudo llevar el proyecto al ambiente productivo, esto a diversos contratiempos tales como definición de esquemas de publicación en mensajería de colas, ya que esta es una nueva tecnología implementada en la compañía y este proyecto es el primero en utilizarla. También, mencionado previamente, se encontraron varias dificultades a la hora de gestión de permisos, tanto en tiempo, como en errores a la hora del otorgamiento.

**Conclusiones**

Las pruebas de concepto son una herramienta muy poderosa para la familiarización de conceptos, herramientas y plataformas. Facilitan en gran manera la implementación de dichos conceptos en una solución productiva.

Luego de realizar las pruebas de concepto sobre las plataformas de Azure Service Bus y RabbitMQ, se pudo observar que el rendimiento de rabbit es muy superior y por ende, la mejor opción para implementar un message broker.

Es de gran importancia poseer en nuestro flujo de trabajo un componente capaz de almacenar datos de trazabilidad, esto con la finalidad de poder recurrir a un paso a paso de las etapas de un proceso y poder analizar esta información en aras de realizar mejoras.

Uno de los pilares para un buen diseño y construcción de una solución es entender el negocio, esto nos da bases sólidas para la toma de decisiones arquitectónicas que ayuden a que la solución tenga un rendimiento óptimo.

Para la construcción de una solución, se deben conocer muy bien el paso a paso de las diferentes etapas que hayan en la compañía, esto para evitar re-implementaciones por problemas de compatibilidad y despliegue.

Una de las grandes ventajas que trae una metodología de trabajo ágil, es el poder presentar avances, stoppers y problemas que se puedan presentar en la implementación de una solución en una etapa temprana.

Para la adopción de metodologías de trabajo ágiles a nivel de compañía, es de gran importancia hacer un trabajo riguroso en todos los niveles de la compañía de mapear sus funciones, objetivos y metas.

Personalmente, aprendí la importancia de adoptar una metodología ágil a nivel de compañía, ya que hace que todos estemos sincronizados hacia los mismos objetivos y que el agilismo brinde flexibilidad y velocidad a la hora de solucionar problemas.

También, aprendí que en un proceso de construcción y desarrollo de software, se presentan muchos impedimentos que a pesar de que pueden parecer muy grandes, se pueden solucionar expresando y comunicando tempranamente dichas situaciones.

Por otro lado, también aprendí que las definiciones arquitectónicas de una solución son el pilar fundamental para un buen diseño de software, sin estas directrices y estándares se tendría que estar re construyendo la solución constantemente.

**Referencias Bibliográficas**


[1] Introduction to Test Driven Development (TDD). (2020). Retrieved 9 February 2020, from http://agiledata.org/essays/tdd.html

[2] Trazabilidad en Calidad. (2020). Retrieved 9 February 2020, from http://gestion-calidad.com/trazabilidad-en-calidad

[3] Conozca Azure | Microsoft Azure. (2020). Retrieved 9 February 2020, from https://azure.microsoft.com/es-es/overview/

[4] Rastreabilidad: ¿qué es y cuál es su importancia?. (2020). Retrieved 9 February 2020, from https://www.teknisa.com/es/erp/rastreabilidad/

[5] What is Scrum?. (2020). Retrieved 9 February 2020, from https://www.scrum.org/resources/what-is-scrum

[6] (2020). Retrieved 9 February 2020, from https://www.gs1.org.ar/documentos/TRAZABILIDAD.pdf

[7] Adaptar nuestra metodología de trabajo a una metodología Ágil - JustDigital | Awesome Digital Products. (2020). Retrieved 9 February 2020, from https://justdigital.agency/metodologia-trabajo-agil

[8] Azure.microsoft.com. (2020). *What is DevOps? DevOps Explained | Microsoft Azure*. [online] Available at: https://azure.microsoft.com/en-us/overview/what-is-devops/#overview [Accessed 9 Feb. 2020].

[9] *Four Steps That Will Guarantee a Profitable Technology Proof of Concept*. Worthwhile.com. (2020). Retrieved 9 February 2020, from https://worthwhile.com/insights/2018/01/22/technology-proof-of-concept/.

[10] *What is NuGet and what does it do?*. Docs.microsoft.com. (2020). Retrieved 9 February 2020, from https://docs.microsoft.com/en-us/nuget/what-is-nuget.

# Anexo.

- ***Consideraciones iniciales (Apache Kafka).***

Desde la página inicial de Apache kafka, es vendido como una plataforma de streaming de datos, posee las funcionalidades de Message Broker pero es más poderoso. Kafka trabaja bajo la estrategia pull. De acuerdo a Apache kafka, uno de los casos de uso que presenta en su documentación es que se requiera la publicación de más de 100.000 mensajes por segundo con complejos enrutamientos, lo que presenta una solución muy robusta.
Otro de los casos de uso que presenta Apache es al momento de utilizar streams de datos para su procesamiento analítico.

- **Definición de palabras clave.**

Partición: Con este concepto, nos referimos a que pueden existir una o varias instancias de una cola o exchange/topic en varios almacenes de mensajes, multiplicando así la capacidad de procesamiento de mensajes.

Exchange o Topic: Es un modelo parecido a una cola, con la diferencia de que puede tener cero o muchas suscripciones sobre ella y cada una de ellas es independiente de otra, donde varias de las suscripciones pueden recibir el mismo mensaje y procesarlo independientemente.

Lock: En este contexto es un mecanismo que nos permite asegurar que una operación se ejecutará de manera atómica.

Routing: Es una característica de los message brokers que nos permite direccionar los mensajes publicados por medio de propiedades y reglas definidas.

- **Definición de características.**

| Características | Rabbit MQ | Azure Service Bus |
|---|---|---|
| OAuth2.0 | Sí, (experimental) | Sí |
| Multiprotocol | Sí, 4 protocolos | Sí, 2 protocolos |
| Auto Delete on Idle | Sí | Sí |
| Message Delay | Sí | Sí |
| Acknowledgement | Sí | Sí |
| Exchange Federation (Replica in another instance) | Sí | Sí |
| Retries | Sí | Sí |
| Multicloud | Sí | No |
| TTL (Time To Live) | Sí | Sí |
| Exchange Binding | Sí | No |
| Sessions | Sí | Sí |
| Partitioning - Clustering | Sí | Sí |
| Routing - Filtering | Sí, 5 métodos | Sí, 3 métodos |
| Dead Lettering | Sí | Sí |
| Autoforwarding | Sí, vía Plugin | Sí |
| Single Active Consumer (SAC) | Sí | Sí |

**Checklist de features.**

- ***OAuth 2.0.***

Es un estándar (framework) de autorización expedido por entidades federadas para control de flujos de acceso mediante intercambios de tokens.

- ***Multiprotocol.***

En este apartado nos referimos a cuántos protocolos de mensajería de colas son soportados por las plataformas seleccionadas, tenemos.

- **AMQP**: Advanced Messaging Queuing Protocol. Es un protocolo estándar para la realización de intercambios de mensajería entre sistemas sin importar el broker o plataforma. Algunas de las características del protocolo son: Multicanal, eficiente, seguro, permite enrutamiento y orientación del mensaje, permite mensajería punto a punto y publish subscriber.

- **MQTT**: Message Queuing Telemetry Transport. Es un protocolo de conectividad M2M (Máquina a máquina) principalmente usado para IoT, fue creado como un protocolo extremadamente liviano para pub/sub. No utiliza colas y no posee header.

- **STOMP**: Streaming Text Oriented Messaging Protocol. Es un protocolo de transferencia de texto simple en formato XML o JSON. Está compuesto de un simple Comando, tipo HTTP, varios headers y un body.

- **HTTP**: Hypertext Transport Protocol. A pesar de no ser un protocolo de mensajería, es soportado por algunos de los brokers mencionados.

- ***Autodelete on Idle.***

Con esta característica, nos referimos a la capacidad que tenemos de configurar una cola de mensajes para que sea eliminada de manera automática cuando el último suscriptor que tenga dicha cola se desuscriba.

- ***Message Delay.***

Este feature es considerado un tipo de exchange especial. Se trata de los mensajes que quieren ser demorados cierto tiempo antes de ser enviados a su destino. El mensaje es enviado a este exchange especial con un header que así lo indique y el tiempo que se quiere esperar. Luego de que el tiempo se cumpla, el broker trata de enrutar el mensaje hacia donde debe ser enviado.

- ***Acknowledgment.***

Se trata del mecanismo que poseen los consumidores de hacerles saber al broker que el mensaje fue recibido con éxito, luego, el broker le hace saber al publicador que se recibió el mensaje. Esto a su vez, permite desencolar el mensaje, ya que cumplió su finalidad.

- ***Exchange Federation.***

Esta capacidad se refiere a la duplicación de mensajes de un exchange a otro, siendo este último parte de un clúster diferente. Podría hacer parte de un protocolo de recuperación en caso de desastre o una técnica para utilización de varios clústeres.

- ***Retries.***

Retries es la capacidad que tiene un broker de mensajería de configurar la política de reintentos en el envío de un mensaje.

- ***Multicloud.***

Posibilidad de crear y administrar el recurso desde diferentes proveedores de servicios en la nube.

- ***TTL (Time To Live).***

Esta característica nos da la capacidad de definir un tiempo de expiración tanto para los mensajes que son enviados a los exchanges/topics tanto como para las colas que están suscritas.

-

- ***Exchange Binding.***

Conocemos la capacidad de suscripción de una cola a un exchange/topic para recibir los mensajes que hayan sido publicados allí. Exchange binding es la posibilidad de “suscribir” o crear un link entre dos exchanges, por lo general en clusters distintos para el paso de mensajes entre una y la otra.

- ***Sessions.***

Es la capacidad que tiene un cliente de crear una sesión que está relacionada a un sessionId, que le permite a un cliente que envía, realizar publicaciones de mensajes que son relacionados con el sessionId únicamente, dichos mensajes pueden ser enviados en cualquier lapso de tiempo entre cada uno y se puede asegurar el orden. Del lado del receptor, se acepta la sesión y se toma un lock que nos asegura que solo dicho receptor puede aceptar los mensajes de la sesión en cuestión y en orden.

- ***Partitioning - Clustering.***

Las particiones, nos permiten fragmentar un topic/exchange o una cola (entidades) en varios pedazos funcionales y estos a su vez, son repartidos entre varios message stores. Cuando se envía un mensaje a entidad particionada, nuestro message broker definirá de manera aleatoria cuál de los stores que se tiene será el que recibirá dicho mensaje para su procesamiento.

- ***Routing - Filtering.***

Esta es la capacidad que nos permite direccionar un mensaje a un destino puntual, por medio de reglas o filtros aplicados que se evalúan de atributos sacados de los metadatos del mensaje publicado. Existen distintos tipos de routings (exchanges/topics).

- **RabbitMQ**:

- ***Direct***: El enrutamiento directo, es el cual a la hora de realizar la publicación de un mensaje se brinda una llave de tipo string que enruta al exchange de ese nombre.
- ***Default***: El enrutamiento default es cuando un mensaje se dirige al default exchange ya que no se brindó una llave de enrutamiento o un string vacío
- ***Topic***: Este enrutamiento se da brindando un lista de palabras separadas por un punto más unos símbolos cómo '*' que precedida de una palabra hace que tenga que estar en ese preciso lugar, otro símbolo es '#' que nos indica que pueden haber cero o más palabras allí. Este enrutamiento se utiliza, cuando se necesita enrutar hacia un exchange específico y dar un poco de flexibilidad para enrutar el mensaje a las colas que estén suscritas a dicho exchange. Podrían considerarse sub-filtros.
- ***Fanout***: Este enrutamiento envía el mensaje publicado hacia todas las colas que estén suscritas all exchange al que se envió.
- ***Headers***: Es muy parecido al Topic, se diferencia en que no se envía llaves de enrutamiento sino encabezados con distintas reglas que pueden o no cumplirse para ser enrutadas, existe una regla especial llamada x-match que nos dice si se tienen que cumplir todas las reglas o solo una para poder enrutar.
- ***Deadletter***: Si ninguno de los routing anteriores es proveído, se enviará el mensaje a la Dead Letter Queue.

- **Azure Service Bus**:

- Boolean filter: Simple enrutamiento que si es true, direcciona a el destino y si es false no.
- SQLFilter: Se usa una sentencia tipo SQL que es evaluada por el broker con la información que llega del mensaje a ser enrutado.
- Correlation filter: En este tipo de filtro, plantea una o varias reglas que son comparadas contra los metadatos del mensaje publicado. Algunas de las reglas son: ContentType, Label, MessageId, ReplyTo, ReplyToSessionId, SessionId, To.
Si existe más de una regla, para que el mensaje sea enrutado con éxito se tienen que cumplir todas.

- ***Dead Lettering.***

Esta característica nos brinda la posibilidad de tener un exchange/topic destinada para el enrutamiento de los mensajes que no cumplen ninguno de los filtros aplicados o cuando se intenta recibir un mensaje y luego de los mecanismos de reintentos no es posible.

- ***Autoforwarding.***

Esta característica es bastante parecida al partitioning, en esta ocasión una cola, exchange/topic es distribuida hacia otra cola o exchange/topic puntual, mientras que en el partitioning se hace de manera aleatoria.

- ***SAC (Single Active Consumer).***

Esta capacidad es bastante parecida a los sessions, nos permite que solo haya un consumidor de mensajes activo, si el consumidor falla, se activa otro que será el único en funcionamiento.

- **Performance**

Para estas pruebas de performance se analizaron los dos tipos de estrategia que se manejan en un broker de mensajería. Pull strategy y Push strategy.

Pull strategy: Se trata cuando al realizarse la publicación de los mensajes en el broker, estos quedan guardados en colas que están suscritas a los topics/exchanges que existan allí. El worker que se encargará de sacar los mensajes de la cola preguntará si hay más mensajes para ser procesados e irá sacando mensajes de esta misma forma.

Push strategy: El worker se encontrará pendiente de la cola observada mientras el broker el exchange/topic que está recibiendo mensajes enrutará hacia la cola suscrita y empujará los mensajes hacia el o los workers que se encuentren disponibles para recepción.

- ***Azure Service Bus.***

Envío mensajes en Azure Service Bus.

Para la publicación de mensajes en Azure Service Bus se creó una instancia en la región East 2 de Estados Unidos, un topic llamado Test con un tamaño máximo de 1GB, un TTL de 14 días y sin particiones y una suscripción llamada sub1.

La porción de código implementada para la publicación de mensajes fue:

```
//Creación del cliente con URL de service bus y nombre del Topic
topicClient = new TopicClient(ServiceBusConnectionString, TopicName);
for (var i = 0; i < 10000; i++)
{
    // Creación de nuevo mensaje
    string messageBody = $"Message {i}";
    //Encoding del mensaje.
    var message = new Message(Encoding.UTF8.GetBytes(messageBody));
    // Escribir mensaje a enviar en consola
    Console.WriteLine($"Sending message: {messageBody}");
    // Enviar mensaje a la cola de manera asíncrona
    await topicClient.SendAsync(message).ConfigureAwait(false);
}
```

Los resultados obtenidos en el envío de mensajes fueron.

| Send Message | Quantity | Time | MPS |
|---|---|---|---|
| | 10.000 | 18:09:07 | 9.18 |
| | 1000 | 1:49:17 | 9.17 |
| | 1.000 | 1.42.81 | 9.80 |
| | 10.000 | 16.44.49 | 9.96 |
| | 20.000 | 32:57.07 | 10.11 |
| | 1000 | 1:39;63 | 10.10 |
| | 10000 | 21:25:58 | 7.78 |
| | 20.000 | 34:25:46 | 9.68 |
| | 10.000 | 16:52:58 | 9.88 |
| | 10.000 | 16:45:02 | 9.95 |

En dónde la desviación estándar de los mensajes por segundo fue de 0.510.

Los resultados del worker en modo Pull fueron:

| Receive Message | Quantity | Time | Concurrent Calls | MPS |
|---|---|---|---|---|
| | 1000 | 4:26:23 | 1 | 3.7 |
| | 10.000 | 42:14:30 | 1 | 3.94 |
| | 10.000 | 42:13:29 | 1 | 3.94 |
| | 10.000 | 43:25:01 | 1 | 3.83 |
| | 10.000 | 44:56:38 | 1 | 3.70 |
| | 10.000 | 43:51:07 | 1 | 3.80 |
| | 10.000 | 42:24:22 | 1 | 3.93 |
| | 10.000 | 41:59:56 | 1 | 3.96 |
| | 10.000 | 4:05:58 | 10 | 40.81 |
| | 10.000 | 17:18:02 | 10 | 10.22 |

En este apartado cabe resaltar que se movieron parámetros del broker como el Concurrent Calls que nos permite definir cuántos mensajes puede procesar en paralelo el worker.

El resultado del worker en modo Push fueron:

| Receive Message | Quantity | Time | Concurrent Calls | MPS |
|---|---|---|---|---|
| | 1000 | 1:52:19 | 1 | 8.9 |
| | 10.000 | 17:22:56 | 1 | 9.59 |
| | 10.000 | 16:56:41 | 1 | 9.84 |
| | 10.000 | 17:01:33 | 1 | 9.79 |
| | 10.000 | 17:11:36 | 1 | 9.78 |
| | 10.000 | 17:15:18 | 1 | 9.66 |
| | 10.000 | 17:07:96 | 1 | 9.73 |
| | 10.000 | 16:58:92 | 1 | 9.82 |
| | 10.000 | 17:25:24 | 1 | 10.22 |
| | 20.000 | 35:25:33 | 1 | 9.41 |

*****En este modo cabe anotar que el worker procesa prácticamente a la misma velocidad que se puedan publicar mensajes en la cola. Se desconoce cuál pueda ser el tope. Estos datos fueron tomados del plan Standard que ofrece Azure Service Bus.

El código utilizado fue:

```
public static async Task Main(string[] args)
{
    subscriptionClient = new SubscriptionClient(ServiceBusConnectionString, TopicName, SubscriptionName);

    Console.WriteLine("======================================================");
    Console.WriteLine("Press ENTER key to exit after receiving all the messages.");
    Console.WriteLine("======================================================");

    // Register subscription message handler and receive messages in a loop

    stopWatch.Start();
    RegisterOnMessageHandlerAndReceiveMessages();

    Console.ReadKey();
    stopWatch.Stop();
    Console.WriteLine("Run: " + stopWatch.Elapsed);

    await subscriptionClient.CloseAsync();
}
```

```
static void RegisterOnMessageHandlerAndReceiveMessages()
{
    // Configure the message handler options in terms of exception handling, number of concurrent messages to deliver, etc.
    var messageHandlerOptions = new MessageHandlerOptions(ExceptionReceivedHandler)
    {
        // Maximum number of concurrent calls to the callback ProcessMessagesAsync(), set to 1 for simplicity.
        // Set it according to how many messages the application wants to process in parallel.
        MaxConcurrentCalls = 10,

        // Indicates whether MessagePump should automatically complete the messages after returning from User Callback.
        // False below indicates the Complete will be handled by the User Callback as in `ProcessMessagesAsync` below.
        AutoComplete = false
    };

    // Register the function that processes messages.
    subscriptionClient.RegisterMessageHandler(ProcessMessagesAsync, messageHandlerOptions);
}
```

```
1 reference
static async Task ProcessMessagesAsync(Message message, CancellationToken token)
{
    // Process the message.
    Console.WriteLine($"Received message: SequenceNumber:{message.SystemProperties.SequenceNumber} Body:{Encoding.UTF8.GetString(message.Body)}");

    // Complete the message so that it is not received again.
    // This can be done only if the subscriptionClient is created in ReceiveMode.PeekLock mode (which is the default).
    await subscriptionClient.CompleteAsync(message.SystemProperties.LockToken);
}
```

- ***RabbitMQ (CloudAMQP).***

Para las pruebas realizadas sobre CloudAMQP se utilizó el plan Little Lemur que brinda un máximo de 100 colas, 20 conexiones simultáneas y 1 millón de mensajes. No se tiene claridad sobre la cantidad de recursos que se brindan en este plan. Este plan se desplegó utilizando Azure en la región East 2.

La porción de código implementada para la publicación de mensajes es:

```
var factory = new ConnectionFactory() {
    HostName = "turkey.rmq.cloudamqp.com",
    UserName = "hsqgadgk",
    Password = "2jeAkJoXSLMfHQec_i3BHFLjs4PsNcwl",
    VirtualHost= "hsqgadgk"
};
using (var connection = factory.CreateConnection())
using (var channel = connection.CreateModel())
{
    //Pregunta si existe el exchange ó lo crea si no existe...se configuran propiedades.
    channel.ExchangeDeclare(exchange: "testing", type: "topic", durable:true);
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.Start();

    for (var i = 0; i < 20000; i++)
    {
        //Llave de enrutamiento en el exchange
        var routingKey = "anonymous.info";
        var message = "Hello World! " + i;
        //Encoding del mensaje
        var body = Encoding.UTF8.GetBytes(message);
        //Publicación del mensaje al exchange seleccionado con llame de enrutamiento y propiedades.
        channel.BasicPublish(exchange: "testing",routingKey: routingKey,basicProperties: null,body: body);
        Console.WriteLine(" [x] Sent '{0}':'{1}' : '{2}'", routingKey, message, i);
    }
    stopWatch.Stop();
    Console.WriteLine("The time elapsed is: " + stopWatch.Elapsed);
}
```

Los resultados obtenidos aplicando el código anterior fueron:

| Send Message | Quantity | Time | MPS |
|---|---|---|---|
| | 1000 | 0:02:68 | 500 |
| | 10.000 | 0:21:54 | 476 |
| | 20.000 | 0:28:51 | 689 |
| | 20.000 | 0:28:74 | 688 |
| | 10.000 | 0:22:34 | 454 |
| | 30.000 | 0:40:45 | 731 |
| | 20.000 | 0:33:53 | 606 |
| | 20.000 | 0:33:42 | 606 |
| | 20.000 | 0:33:18 | 606 |
| | 20.000 | 0:33:93 | 588 |

La desviación estándar de los MPS fue de 71,92.

Los resultados del Worker en modo pull fueron:

| Receive Message | Quantity | Time | MPS |
|---|---|---|---|
| | 1000 | 1:27:64 | 11.49 |
| | 10.000 | 16:11:04 | 10.29 |
| | 20.000 | 28:32:89 | 11.68 |
| | 20.000 | 26:26:20 | 12.61 |
| | 20.000 | 29:12:05 | 11.41 |
| | 20.000 | 26:26:47 | 12.61 |
| | 20.000 | 26:39:47 | 12.50 |
| | 20.000 | 29:23:02 | 11.34 |
| | 20.000 | 29:34:57 | 11.27 |
| | 20.000 | 29:05:27 | 11.46 |

Los resultados del Worker en modo push fueron:

| Send Message | Quantity | Time | MPS |
|---|---|---|---|
| | 1000 | 0:02:68 | 500 |
| | 10.000 | 0:21:54 | 476 |
| | 20.000 | 0:28:51 | 689 |
| | 20.000 | 0:28:74 | 688 |
| | 10.000 | 0:22:34 | 454 |
| | 30.000 | 0:40:45 | 731 |
| | 20.000 | 0:33:53 | 606 |
| | 20.000 | 0:33:42 | 606 |
| | 20.000 | 0:33:18 | 606 |
| | 20.000 | 0:33:93 | 588 |

Los mensajes se procesan a la misma velocidad en que se publican, se podría decir que en “tiempo real”.

## Conclusiones

- Se pudo observar que en las pruebas de performance CloudAMQP obtuvo mejor rendimiento tanto en la publicación como en la recepción de mensajes. Tanto en el push strategy como en el pull strategy.

- CloudAMQP mostró un rendimiento superior utilizando el free tier que proveen de prueba, mientras que las pruebas sobre Azure Service Bus se realizaron bajo un plan Standard pago.

- Azure Service Bus solo puede ser desplegado en Azure lo cual en cierto modo lo hace vendor lock-in, sin embargo, ya que uno de los protocolos de comunicación que usa es AMQP, esto le permite comunicarse con otros Message brokers fácilmente.

- Ambos Message Brokers presentan unas características muy similares, además, presentan plugins que amplían sus capacidades.

- Para la creación de este documento se encontró que la documentación de CloudAMQP (RabbitMQ) es clara y presenta ejemplos variados concisos que ayudan a la implementación de las funcionalidades.