Propuesta arquitectónica para la migración de un sistema monolítico a un sistema distribuido utilizando domain driven design y microservicios: caso de estudio

José Arteaga, Roberto Bonilla, Carlos Flores
Centro de estudios de postgrado,
Universidad Don Bosco,
San Salvador, El Salvador
jh.arteaga09@gmail.com, jrbonillach@gmail.com, mick.fm182@gmail.com

***Resumen*- Específicamente en El Salvador la arquitectura distribuida en microservicios no se ha incorporado ni se ha explotado en su totalidad por los departamentos de tecnología, que durante este estudio se ha hecho referencia con la siguiente sigla TI (Tecnologías de la Información), por tal motivo, la información de casos de éxito bajo este tipo de arquitectura es escasa.**

El objetivo de este trabajo fue proporcionar información sobre el proceso de migración de una arquitectura monolítica a una arquitectura distribuida en microservicios, que sirva como guía de referencia para todo el interesado en aplicar un proceso de transformación de este tipo. Para entender el proceso de transformación se abordarán dos conceptos que ayudan a rediseñar aplicaciones con características de ser monolíticas.

***Palabras clave*- arquitectura de software, microservicios, domain driven design, arquitectura distribuida, arquitectura monolítica.**

I. INTRODUCCIÓN

Específicamente en El Salvador la arquitectura distribuida en microservicios no ha sido incorporada ni explotada en su totalidad por los departamentos de tecnología, que durante este estudio se ha hecho referencia con la siguiente sigla TI (Tecnologías de la Información), por tal motivo, la información de casos de éxito bajo este tipo de arquitectura es escasa.

En este documento se ha proporcionado información sobre el proceso de migración de una arquitectura monolítica a una arquitectura distribuida en microservicios, que sirva como guía de referencia para todo el interesado en aplicar un proceso de transformación de este tipo. Para entender el proceso de transformación se abordarán dos conceptos que ayudan a re diseñar aplicaciones con características de ser monolíticas.

II. INFORMACIÓN BÁSICA DEL PROYECTO

A. Planteamiento del problema

Actualmente en El Salvador, es muy difícil obtener información de empresas que hayan implementado un proceso de rediseño de software utilizando domain driven design (su sigla DDD) y microservicios, debido a que este tipo de documentación puede ser interna de cada empresa, y por políticas de las mismas no se exponen al público. Por otra parte, dentro de los departamentos de TI la noción de lo que es una arquitectura distribuida no siempre se concibe más allá de una aplicación diseñada en capas que separan los componentes del software (usualmente de tres capas, presentación, lógica, y acceso a datos).

Por esta razón, en este documento se ha plasmado el proceso de transformación de un software monolítico hacia una arquitectura distribuida en microservicios, con base en un caso real dentro de una empresa dedicada a los servicios de transporte aéreo. La problemática se abordó mediante un sistema de software desarrollado dentro de la empresa, que debido al paso del tiempo, los requerimientos constantes del negocio y la creciente carga transaccional de nuevos clientes, su arquitectura inicial se ha vuelto difícil de escalar y mantener, por consiguiente, origina un retraso para adaptar nuevas funcionalidades y responder de forma eficaz a las necesidades de la organización.

El diseño inicial del sistema se basó en una arquitectura monolítica en capas, debido a que en ese momento era suficiente para cubrir las necesidades que tenía la compañía, y además era el estándar que

utilizaban para diseñar y desarrollar sus aplicaciones.

Sus procesos de negocio y su procesamiento de datos han sido desarrollados de forma secuencial y síncrona, al igual que el manejo de sus dos fuentes principales de información: una base de datos transaccional y un servicio que provee la información de los pasajeros según los vuelos y su ruta.

En base a métricas de rendimiento hechas por el departamento de calidad de la compañía, se ha comprobado que la aplicación presenta deficiencia en sus tiempos de respuesta, que oscila entre 20 a 30 minutos por vuelo a compensar en horas de alta transaccionalidad (en cada vuelo se puede transportar entre 80 a 160 pasajeros dependiendo la unidad de transporte y la distancia del recorrido). Cuando la afluencia de pasajeros por compensar aumenta, el rendimiento de la aplicación decae ante la cantidad masiva de peticiones que recibe a límites inaceptables (cabe mencionar que la aplicación es utilizada en más de 20 países en América y Europa).

El entorno donde se encuentra el sistema analizado ya no es viable para cubrir las necesidades técnicas y de negocio, debido a que el consumo de los recursos aumenta según la demanda, y esto tiene una afectación directa en el consumo de recursos y los tiempos de respuesta.

B. Justificación

Las tecnologías a lo largo del tiempo se van actualizando, sin embargo, quedan desfasadas con celeridad, las necesidades del negocio van cambiando y creciendo con el objetivo de satisfacer al usuario final y brindarle una experiencia sin igual. Debido a ello, las áreas de TI se ven obligadas continuamente a realizar cambios en sus sistemas de forma ágil para generar valor y respuesta lo más pronto posible. No obstante, muchas empresas se ven impactadas debido a las necesidades cambiantes del negocio, y a la falta de agilidad para adaptar sus sistemas de información a éstas.

Como se ha mencionado, la aplicación analizada en este caso de estudio fue construida bajo una arquitectura monolítica, ésta cumplía con sus requerimientos, sus necesidades, y su visión en ese momento, sin embargo, con el crecimiento que ha tenido la compañía, el área de negocio ha requerido de nuevas funcionalidades, por lo que esta arquitectura ya no es adecuada y no responde a las nuevas exigencias. Es por ello que se inició la construcción de un nuevo diseño arquitectónico bajo una arquitectura distribuida para acelerar los despliegues, hacer cambios sin afectar otras funcionalidades de la aplicación, brindar alta disponibilidad, y no dar de baja a toda la aplicación cada vez que se realice un cambio sobre la misma.

Es importante analizar el contexto de la aplicación porque ayuda a comprender el problema y permite generar una alternativa de solución. Además de esta investigación, se creó una guía de recomendaciones y buenas prácticas de manera general para migrar una aplicación monolítica hacia una distribuida aplicando los conceptos de DDD y microservicios.

Utilizar una arquitectura de microservicios sugiere una gran ventaja competitiva para las empresas, en vista de que faculta a los departamentos de TI a construir aplicaciones que respondan con relativa rapidez a sus necesidades, y con esta investigación se buscó apoyarles para aumentar su nivel de madurez en cuanto a la integración de sus sistemas, y así puedan mejorar su acuerdo de nivel de servicio para con sus clientes.

Finalmente, y no menos importante, los temas de arquitectura de microservicios y DDD están en la mente de cada persona que ejerce esta profesión de Ciencias de la Computación, no obstante, no todos saben cuál es el camino para implementarlo, de dónde partir, y muchas veces únicamente quedan los conceptos e ideas a nivel de papel, y difícilmente pueden ser implementados en su día a día. Es por ello que con esta investigación se buscó proveer a personas informáticas una guía de recomendaciones y buenas prácticas para salir de la teoría llevando a la práctica los conceptos que muchos manejan al respecto.

C. Objetivos

1) Objetivo general

Diseñar una propuesta de arquitectura distribuida para una aplicación partiendo de su modelo de dominio existente utilizando el enfoque de domain driven design y microservicios.

2) Objetivos específicos

- Identificar los componentes del dominio que se migrarán a microservicios distribuidos.
- Definir una arquitectura distribuida usando microservicios partiendo del modelo de dominio.
- Elaborar un documento de recomendaciones para la migración de un sistema monolítico a un sistema distribuido.

III. CONCEPTOS FUNDAMENTALES

El primero es conocido como domain driven design, que de ahora en adelante será mencionado como DDD, este concepto ayuda a diseñar un modelo de dominio partiendo de la lógica del negocio o la problemática que se necesita resolver, de tal forma que se pueda enfocar el desarrollo de manera más eficiente en las actividades y requerimientos que realmente generan valor. Eric Evans (2003)

Por otro lado, el concepto de arquitectura distribuida en microservicios nos brinda un enfoque para construir unidades pequeñas de software conocidas como microservicios, que interactúan en un ambiente distribuido, cada uno de estos microservicios es "dueño" de una funcionalidad de negocio específica y única, de manera que permita hacer un cambio de negocio de forma fácil, escalable y sin afectar las demás funcionalidades del sistema. Coulouris G., Dollimore J., Kindberg T. y Blair, G. (2011)

Enlazando ambos conceptos, en este caso de estudio el proceso de migración lo iniciaremos con el diseño de un modelo de dominio utilizando DDD, dado que proporciona una estructura bien definida y centrada en las funcionalidades del negocio, la cual se constituye de un modelo de dominio conformado por un conjunto de subdominios, los cuales deben ser flexibles, independientes y dueños de una única funcionalidad de negocio. Y a partir de este punto se identificó los microservicios, de forma tal que fueron coherentes y con bajo acoplamiento para finalizar la investigación con una propuesta de arquitectura distribuida basada en microservicios.

IV. MARCO SITUACIONAL

A. *Descripción de la compañía*

Esta compañía es una empresa de transporte aéreo y fue fundada en 1919. Es la segunda aerolínea más grande de Sudamérica después de LATAM. Sus principales centros de conexión se encuentran en el aeropuerto el Dorado de Bogotá, el aeropuerto internacional de El Salvador y el aeropuerto internacional Jorge Chávez de Lima.

La compañía tiene una planilla por más de 20,500 colaboradores y una flota compuesta por más de 200 aeronaves que operan a más de 100 destinos en 26 países de América y Europa, y ofrece un promedio de 5,100 vuelos semanales.

Luego de un proceso de evolución de la compañía, en 2010 oficializaron su fusión con otra aerolínea latinoamericana, lo cual dio marcha a un riguroso proceso de reorganización administrativa, así como de integración de sus redes de rutas, homologación de procesos y captura de sinergias. Y actualmente la compañía está constituida por un grupo de 7 aerolíneas subsidiarias.

B. *Descripción general del sistema*

El sistema analizado para este caso de estudio fue desarrollado en el año 2011 en respuesta a la fusión de dos compañías del transporte aéreo, el cual tenía como propósito la homologación de los procesos de negocio de dos sistemas diferentes bajo un mismo core.

La definición de compensación en lenguaje del negocio para este rubro es el medio por el cual se trata de compensar al cliente con el objetivo de retribuir cualquier inconveniente o falla que pueda generarse en el servicio que le brinda la compañía, y de esta forma tratar de conservar su lealtad.

En el día a día es necesario registrar y contabilizar las compensaciones generadas en los puntos de abordaje por diversos motivos, por ejemplo: demoras, cancelaciones, fallas mecánicas de última hora, entre otros. Existen además diversas áreas de negocio que hacen uso de compensaciones, entre ellas están el call center, carga, equipajes, aeropuertos, y puntos de ventas.

Para dar respuesta a la necesidad de controlar las compensaciones, el sistema cuenta con un portal web que es utilizado específicamente por el área de puntos de abordaje, y a su vez expone un servicio con toda la lógica para el manejo de una compensación. Este servicio es utilizado por aplicaciones de otras áreas de negocio que hacen uso de la compensación a clientes.

La arquitectura de la solución ha sido diseñada utilizando un patrón en capas, en la que cada capa está constituida por componentes específicos según su función. Las capas diseñadas están constituidas por las siguientes: la capa de vista, que es la encargada de la presentación a los usuarios, una capa lógica, la cual maneja toda la lógica de negocio de la aplicación, y una capa de datos, la cual maneja todas las interfaces tanto internas como externas hacia otros sistemas.

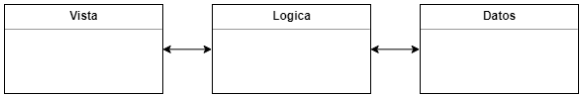

Fig. 1. Arquitectura monolítica en capas. Proporcionado por la compañía.

En la solución del sistema la capa de vista está representada por el proyecto de ICare.Web, la capa de lógica está representada por el proyecto ICare.App, la capa de datos está representada por los proyectos ICare.DAO y ICare.DTO, el proyecto ICareService representa el servicio web expuesto por el sistema y que comparte la misma lógica y datos que la aplicación web.

Estas capas de lógica es la que se espera llevar a microservicios para ser componentes reusables y aislados que puedan ser mantenidos independientes de cualquier otro sistema. Sam Newman (2015).

A continuación, se presenta la información general del sistema:

Campo	Descripción
Descripción de la aplicación	Sistema de registro y emisión de compensaciones generados por diferentes canales y áreas de negocio de la compañía. También da soporte al proceso de compensación por irregularidades de vuelos y sobre ventas.
Naturaleza	Aplicación Web
Servidor de Aplicación	Dos servidores balanceados.
Servicios Que Expone	Web Service para: Registro de pasajero afectado Registros de motivos de afectación Solicitud de emisión de compensaciones Solicitud de anulación de compensaciones.
Segregación	Desarrollo y Producción.
Alta Disponibilidad	Si
Estándar de desarrollo	Utilización de Programación Orientada a Objetos para encapsular métodos y funcionalidad utilizada por otros sistemas o componentes. Control de excepciones Tres Capas
Proceso de Negocio que soporta	Emisión de compensaciones por irregularidades de vuelo Control de política de compensación. Contabilización de servicios.

Tabla 1. Información adicional del sistema. Proporcionado por la compañía

C. Componentes arquitectónicos del sistema

Con los componentes identificados, es necesario comprender cómo es la interacción dentro de la lógica de la aplicación. Para esto, se explicará el diagrama de componentes arquitectónico.

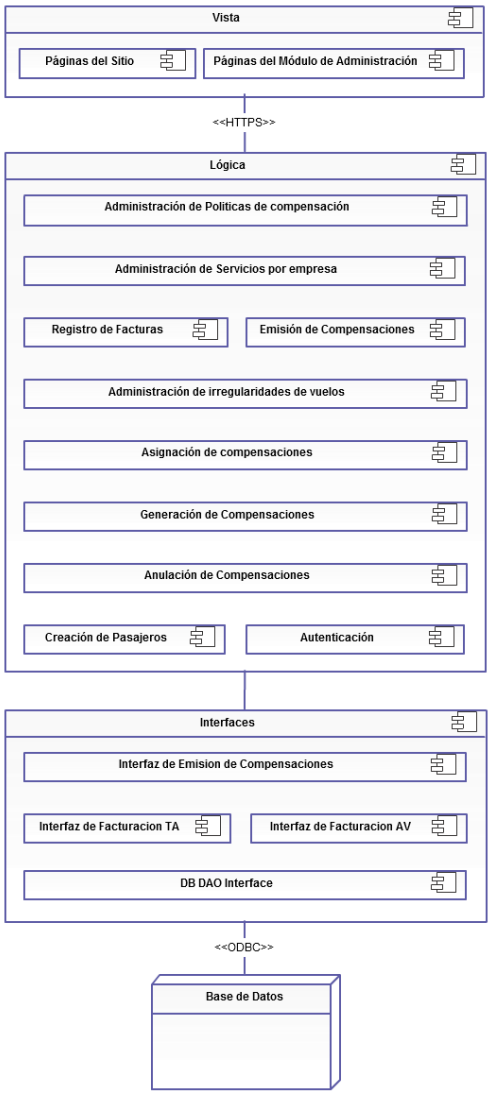

Fig. 2. Componentes arquitectónicos. Proporcionado por la compañía.

1. *Páginas del sitio*: En este componente se encapsulan todas las páginas de la solución.
2. *Páginas del módulo de administración*: En este componente se encapsulan todas las páginas de la solución que tengan relación con los módulos de administración descritos en los siguientes puntos.
3. *Autenticación*: Este componente se encarga de la autenticación de los usuarios en la aplicación el cual hará uso del módulo de autenticación de la compañía mediante servicio web de Single Sign-On.
4. *Administración de políticas de compensación*: En este componente se realiza la administración de compensación, creación y asignación de políticas por motivo.

5. *Administración de servicios por empres*a: En este componente se puede realizar la administración de los proveedores y asociar los servicios que pueden brindar las empresas a los pasajeros.
6. *Registro de facturas*: Se realiza el proceso de validación y registro de las facturas para su posterior procesamiento en el sistema contable.
7. *Asignación de compensaciones*: En este módulo se determina qué tipo de compensación se le brindará al pasajero afectado según motivos de compensación asignados al vuelo, apegándose a las políticas de compensación.
8. *Generación de compensaciones*: En este módulo se relaciona la retribución exacta que se le brindará al pasajero afectado en cada uno de los lugares en los que se puede emitir una compensación.
9. *Emisión de compensaciones*: En este módulo se ejecuta la generación de las compensaciones a entregar al pasajero afectado, acreditar millas, efectivo, etc.
10. *Anulación de compensaciones*: En este módulo se realizan las anulaciones de la compensación que tienen un estado de emitido para poder hacer el respectivo rollback según aplique para cada caso.
11. *Administración de irregularidades de vuelos*: Componente que permite la administración de los vuelos con irregularidades y que son sujetos de compensaciones para los pasajeros que han sido afectados.
12. *Creación de pasajeros*: Módulo en el cual se administrará la lista de pasajeros afectados por irregularidades en los vuelos.
13. *Interfaz de Emisión de compensaciones*: Interfaz que se expone como servicio para poder registrar las compensaciones emitidas por sistemas de terceros.
14. *Interfaz de facturación 1*: Componente que permite realizar el procesamiento de los datos de las facturas en el sistema contable utilizado en SAP (Sistemas, Aplicaciones y Productos), con el fin de realizar el pago a proveedores. Esta interfaz ha sido expuesta como un servicio web.
15. *Interfaz de facturación 2*: Componente que permite realizar el procesamiento de los datos de las facturas en el sistema contable utilizado en Oracle Financial, con el fin de realizar el pago a proveedores. Esta interfaz fue expuesta como un servicio web.
16. *Interfaz para base de datos usando DAO (Objeto de Acceso a Datos, por sus siglas en inglés)*: Es una interfaz con enlace directo a la base de datos que realiza toda la administración de las cadenas de conexión.

La capa de vista es la encargada de mostrar a los usuarios que se encuentran en los puntos de abordaje, la interfaz gráfica. Esta interfaz interactúa con la capa de lógica, la cual contiene las clases para cada uno de los componentes de negocio. Uno de los principales problemas, es que la capa de lógica contiene un alto acoplamiento, debido a que toda la lógica de negocio está situada en una sola capa. Las interfaces proveen el acceso a la persistencia y a los demás servicios de terceros.

V. PROPUESTA ARQUITECTÓNICA USANDO MICROSERVICIOS

Con el desacoplamiento de la lógica de negocio como subdominios en la capa domain de la solución del sistema, se ha abarcado gran parte de la transformación del sistema. Esto debido a que DDD provee herramientas para el modelado de dominios complejos, lo que garantiza que cada subdominio es de responsabilidad única y que no dependen de otro Eric Evans (2003); por lo tanto, hemos reducido el acoplamiento de la funcionalidad de negocio con las capas de presentación, aplicación e infraestructura.

Aun con este proceso de modelado por DDD la solución aún se presenta como un monolito, debido a que toda la lógica de negocio aún forma parte de este compilado y como tal se publica en el mismo servidor junto con las demás capas, vistas y demás componentes. Lo que se ha logrado hasta este punto es mejorar la calidad del código, escalabilidad, reusabilidad, abstracción de las capas, dependencia mínima en la infraestructura, y mejora en el rendimiento del sistema, puesto que se ha reducido la complejidad ciclomática, la duplicidad de código y se ha pasado de un entorno síncrono a un asíncrono en la mayor parte de la comunicación entre capas.

Esta mejora en el rendimiento no se reflejó en la reducción de los tiempos de respuesta en horas pico con alta transaccionalidad, porque como lo mencionamos anteriormente, el sistema aún es monolito por lo que su lógica y el procesamiento de datos se hace bajo el mismo servidor; de igual forma otras aplicaciones alojadas hacen uso de los recursos.

Para mitigar esta situación se propuso una arquitectura distribuida de microservicios, la cual sugiere migrar los subdominios definidos en la capa domain a microservicios, dado que estos cumplen con las propiedades de ser independientes, pequeños y manejan una sola funcionalidad.

Clientes: Los clientes son la aplicación web que mantendrá la lógica de los CRUD y todo el módulo de administración del sistema, y el servicio web que es expuesto a otras aplicaciones que necesitan emitir compensaciones. Para los clientes se planeó dejar la infraestructura tal como se encuentra en este momento, debido a que se extrajo la lógica y el acceso a datos de la aplicación, por lo que sólo serán encargadas de solicitar y renderizar la información al usuario. Esto reduce en gran medida el uso de los recursos de los servidores actuales. Estos clientes son el ICARE web y microservicios.

Api gateway: es el encargado del descubrimiento y de manejo de los endpoints de los servicios para que puedan ser consumidos. El proceso de registro de los servicios se hará automáticamente en el tiempo del bootstrap. Este api gateway es el encargado de orquestar los microservicios, a causa de que es el único punto de entrada para los clientes y él será quien conozca los endpoints y redirija las peticiones entrantes al servicio correspondiente, así los clientes sólo tienen la necesidad de conocer un endpoint y no todos los N endpoint de cada microservicio.

Se planteó una capa de seguridad Oauth2 para garantizar que sólo los clientes certificados puedan conectarse y consumir los servicios expuestos. Bajo esta propuesta todas las peticiones y servicios deberán ser bajo el protocolo REST y JSON como request y response, pero no se descarta la posibilidad de utilizar un protocolo de comunicación como SOAP.

Microservicios: Son el resultado de la identificación de los subdominios, contienen una única función de negocio y no dependen de otros servicios. Se planeó un despliegue por medio de dockers y swagger para la documentación, y así definir el api. Su consumo será bajo el protocolo REST y JSON como request y response. Cada microservicio contiene una capa de acceso a datos ya sea a base de datos, texto plano, ftp, servicios web, entre otros.

Adicional a los subdominios identificados por la empresa y los subsecuentes microservicios obtenidos de cada uno de ellos, se agregaron servicios para manejar las peticiones CRUD para enviar y recibir peticiones de la base de datos, así como un servicio para conversión de cantidades a moneda, puesto que muchos de los montos presentados se expresarán como moneda y la conversión a moneda será de una ejecución recurrente entre los diferentes servicios que coexisten en el sistema.

Load balancer: Se sugirió usar un balanceador para permitir la distribución de la carga en distintas instancias de forma transparente al momento de consumir el servicio. Muchas herramientas cloud permiten la creación automática de instancias de los servicios por medio de diferentes parámetros, por ejemplo, horas pico, uso de recursos, entre otros. Esto proporciona un plus, porque así se garantiza que mientras más transacciones haya, habrá un mejor manejo y escalamiento horizontal de instancias para atender la demanda.

Productor/consumidor: Se planteó usar la tolerancia a fallos para manejar de forma óptima los fallos o errores en los servicios, debido a que si un servicio falla esto evitará que se propague en cascada, de forma que se pueda gestionar el error y controlarlo a nivel del servicio donde se produjo. Por tal motivo se propuso utilizar el patrón de diseño productor/consumidor para organizar las peticiones de los microservicios en colas que almacenen las peticiones ya sean SOAP, REST y JSON para que cada vez que un servicio lance una petición de actualización y/o validación de datos de datos, estas peticiones se almacenen en una cola esperando ser procesadas por el servicio pertinente.

Circuit breaker: Se planteó usar la tolerancia a fallos para manejar de forma óptima los fallos o errores en los servicios, en vista que si un servicio falla esto evitará que se propague en cascada, de forma que se pueda gestionar el error y controlar a nivel del servicio dónde se produjo. Por tal motivo, se utilizan llamadas a los servicios de validación y/o recuperación de datos para que se tome en cuenta un tiempo de espera para recibir la respuesta del microservicio encargado de devolver los datos y si este time-out es alcanzado la petición se desestima, por lo que el sistema se detendrá hasta recibir una nueva petición, y esto se repetirá hasta que el servicio de validación y/o recuperación de datos esté activado de nuevo.

Este patrón complementa a lo realizado en el patrón productor/consumidor, en dicho patrón las peticiones son almacenadas en una cola de peticiones, circuit breaker verifica en dichas colas de petición si la respuesta en forma de mensaje en la cola está presente; si no está presente, es decir si el time-out de la petición es alcanzado, se hace una nueva verificación con un nuevo periodo de tiempo para ver si la respuesta está presente en la cola, de esta manera el sistema puede esperar de manera

indefinida la respuesta en la cola y puede configurarse un límite de intentos para lanzar una advertencia de que los datos no están disponibles, y además, puede ser configurada en el api gateway para tres peticiones fallidas (que han alcanzado un time-out), por ejemplo.

Datos/aplicaciones legacy: Se planteó reutilizar la base de datos transaccional del sistema, debido a que esta cuenta con alta disponibilidad, por lo tanto, cada microservicio sólo envía una petición con las tramas de SQL a ejecutar, luego estas se almacenarán en una cola de peticiones en espera de que sean procesadas por la misma base de datos o por un manejador externo que redireccione estas tramas a la base de datos. No se propuso crear una base de datos por cada microservicio en visto a que involucra altos costos económicos y un proceso de migración de los datos a cada instancia. Los microservicios harán el consumo de los web services que manejan la información de los vuelos.

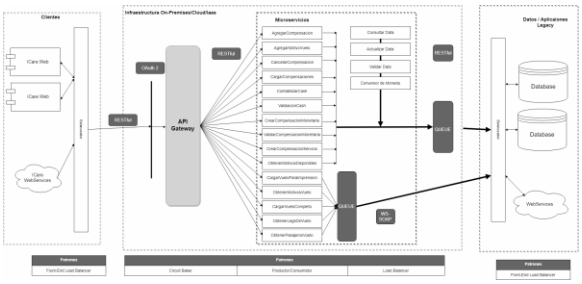

Fig. 3. Arquitectura de microservicios propuesta. Autoría propia.

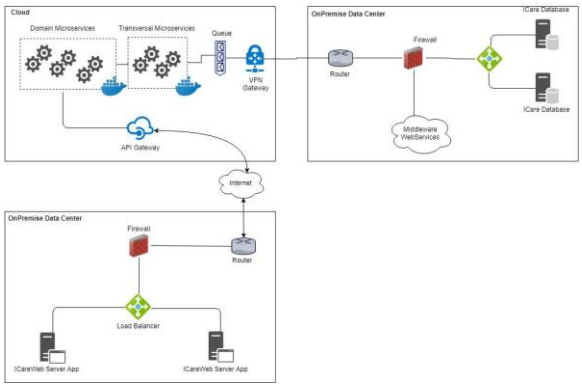

Fig. 4. Arquitectura de infraestructura. Autoría propia

En esta propuesta se implementaron los servicios en la nube, debido a que hay diversos proveedores que ofrecen herramientas y servicios que ayudan a la administración, escalabilidad, seguridad y monitoreo para este tipo de arquitecturas. Es a consideración de la compañía poder elegir el proveedor de servicios en la nube, debido a que está sujeto a contratos marcos que actualmente estén vigentes, cotizaciones de los servicios, licitaciones, entre otros puntos que la compañía estime.

Se recomendó hacer el despliegue de los microservicios en docker, dado que se puede hacer el despliegue sin importar el lenguaje de programación o sistemas operativos, ya sea Linux o Windows, y en servicios en la nube se puede hacer un escalado dependiendo de la demanda que se obtenga.

El api gateway es el canal de entrada que nos ofrece seguridad, enrutamiento y monitoreo de los microservicios. Este es accedido a través de internet, por lo que el front-end podrá hacer llamadas a directorios, como por ejemplo con AJAX.

En la infraestructura on premise, se mantendrá el front-end para que únicamente haga peticiones al api gateway. Esto reducirá el costo de desarrollo, dado que si se pretende hacer una migración completa hacia la nube es necesario hacer una re-plataforma tanto en back-end como en front-end. Cabe mencionar que por el alto grado de demanda el front-end permanecerá en el balanceador tal como se encuentra en la infraestructura actual.

Es importante considerar para una segunda fase la migración de la base de datos transaccional, debido a que en esta propuesta se ha considerado mantenerla en infraestructura on premise, lo cual genera más latencia por los saltos hasta llegar ella, y a su vez hay un alto grado de riesgo por si el servidor presenta fallas.

En esta propuesta los microservicios en la nube serán los encargados de consultar y operar contra la base de datos transaccional, y para esto se necesita habilitar un túnel VPN para poder abrir la conexión entre ambas infraestructuras. El consumo del servicio para obtener información de los vuelos se hizo mediante el mismo túnel VPN y será necesario crear reglas de firewall para que tanto la base de datos y el servicio de vuelos puedan ser consumidos por los microservicios.

VI. CONCLUSIONES Y RECOMENDACIONES

Si bien las aplicaciones monolíticas no son malas, en vista de que permiten un desarrollo más rápido porque tenemos todas las capas en un mismo proyecto, y en la mayoría de los casos basta con una aplicación de este tipo para solucionar los problemas de negocio; se debe tener en consideración que a medida que el sistema va creciendo y se agregan más funcionalidades, se vuelve muy complejo de mantener. Durante el desarrollo de este estudio pudimos constatar bajo un escenario real, que una aplicación diseñada con una arquitectura monolítica

puede enfrentar dificultades cuando el tamaño del código fuente crece y son constantes sus requerimientos o mejoras, por ende, el desarrollo comienza a ser cada vez más difícil y aumenta la complejidad de entendimiento, acoplamiento, mantenibilidad y escalabilidad.

Así mismo a medida crece una compañía, las aplicaciones deben adaptarse no sólo a nuevos procesos de negocio, sino a responder a un aumento en demandas de transacciones, usuarios concurrentes, entre otros aspectos. La infraestructura forma parte de esta adaptabilidad, pero es más importante que la aplicación esté bien diseñada para que pueda aprovechar los recursos de la mejor forma y ser escalable sin recurrir a una gran inversión en infraestructura como en muchas ocasiones sucede.

El objetivo de esta tesis fue examinar cuáles son las diferencias entre una arquitectura monolítica y de microservicios, y dejar claro cómo funciona cada una de estas arquitecturas mostrando sus ventajas, desventajas y el proceso para migrar de una arquitectura a otra. Es claro que un proceso de migración de arquitectura no aplique a todas las aplicaciones monolíticas, debido a que para determinar esto se debe tener en cuenta diversos factores tanto económicos, técnicos y de negocio. En muchos casos una aplicación monolítica puede ser suficiente para subsanar la demanda del negocio y sus clientes.

El proceso de una migración de arquitectura puede ser más complejo que el desarrollo de una aplicación desde cero, porque se debe analizar detenidamente la funcionalidad existente e identificar su núcleo, la descomposición de componentes, delimitar contextos, entre otras actividades. Para tener claro este proceso se tomó como base fundamental el concepto domain driven design de Eric Evans y se plasmó en una guía de recomendaciones y buenas prácticas que sirve como referencia para cualquier equipo de TI que tengan proyectos de este tipo.

En la actualidad, es muy común escuchar hablar sobre la arquitectura de microservicios, pero aún no existe una definición clara de cómo poder implementar este tipo de arquitectura. Todo dependerá de los recursos y conocimientos que cuente el departamento de TI, por lo que en este estudio se propuso un diseño de arquitectura que no está ligado a un lenguaje de programación, framework o proveedor de servicio en la nube.

Como un resumen de este estudio, y de lo que se logró recopilar de las personas expertas que han apoyado en este trabajo, se comparte a continuación los siguientes puntos que se debe tener en cuenta al llevar a cabo este proceso:

A. El primer paso para poder iniciar este tipo de migración de un sistema no es con respecto a tecnología, se debería dar inicio con el factor humano. Trabajar con microservicios primeramente es un cambio de mentalidad en el área de IT con respecto a la programación de una forma lineal y estructural a una forma distribuida, y en segundo lugar que entiendan muy bien los conceptos involucrados en esta nueva arquitectura de sistemas. Para este tipo de migraciones y desarrollo de microservicios, se requiere de un equipo de desarrollo capacitado y que mejor se adapte a los cambios.

B. Establecer un marco común y acordado, utilizando domain driven design, y con ello entender cuáles son los actores y procesos que están involucrados en el entorno del sistema, cuáles procesos se verían afectados en la migración, y entender los subdominios, porque a partir de ellos se podrá identificar los nuevos microservicios.

C. Se debe entender y definir el tamaño y el alcance del microservicio, puesto que un microservicio debe tener una única responsabilidad dentro del marco de necesidades del cliente y del negocio.

D. No empezar a diseñar todos los procesos de la empresa como microservicios, sino que empezar con pocos y que no sean de gran impacto para la empresa. A medida que se vaya comprendiendo dicha arquitectura y el proceso de migración, se irán incluyendo acorde a la necesidad de la empresa más procesos a la migración.

E. Tener en cuenta que, al montar una arquitectura de microservicios, se recomienda implementar a futuro un flujo de integración continua, con el objetivo de facilitar la administración y despliegues de estos microservicios.

F. Se recomienda trabajar desde un inicio con un marco de trabajo ágil para llevar a cabo la migración de un sistema monolito a un sistema distribuido con microservicios, debido a que las necesidades del negocio van cambiando día con día, y la solución debe apegarse a dichos cambios.

G. Se recomienda definir y tener claridad de los requerimientos, hacer un buen diseño arquitectónico de la solución, previo al desarrollo.

La arquitectura de microservicios es una realidad para muchas empresas grandes como Microsoft, Amazon, Google, Netflix, entre otras, las cuales han demostrado que esta arquitectura es completamente viable para soluciones empresariales de alta demanda, y despliegues continuos montados en un entorno ágil en el proceso de desarrollo. No se debe pensar que sólo las empresas con un capital millonario para un departamento de TI pueden implementar este tipo de arquitecturas y soluciones, depende más del recurso humano y el conocimiento que puedan tener para poder implementarlo.

RECONOCIMIENTO

Con especial agradecimiento a Juan Carlos Aquino por fungir como asesor de tesis, y formar parte del equipo de trabajo que permitió culminar esta investigación.

REFERENCIAS

[1] Robert C. Martin, Martin Micah (2006), Agile Principles, Patterns and Practices in C#, Prentice Hall, ISBN-13 978-0-13-185724-4

[2] Bess L., Clements P., Kazman R. (2012). Software Architecture in Practice, 3ra edición. Westford, Massachusetts. ISBN-13 978-0321815736

[3] Alberto Lafuente (2012), Introducción a los Sistemas distribuidos, Departamento de Arquitectura y Tecnología de Computadores, UPV/EHU.

[4] Coulouris G., Dollimore J., Kindberg T., Blair, G. Distributed Systems: Concepts and Design, 5ta edición. Upper Saddle River, Nueva Jersey: Pearson Education, Inc. ISBN-13: 978-0132143011.

[5] Carlos Billy Reynoso (2004). Introducción a la Arquitectura de Software. UNIVERSIDAD DE BUENOS AIRES

[6] Dan Haywood. (2009). Domain-Driven Design, the Pragmatic Bookshelf, Raleigh, North Carolina Dallas, Texas. ISBN-13 978-1-934356-44-9

[7] E. Evans (2015) Domain Driven Design: Reference Definitions and Pattern Summaries. 1ra edición, Pearson Education, Inc. Boston, Massachusetts. ISBN- 978-1-4575-0119-7

[8] E. Evans (2003). Domain-Driven Design: Tackling Complexity in the Heart of Soft, 1ra edición. Boston, Massachusetts: Pearson Education, Inc. ISBN-13: 978-0321125217.

[9] E. Gamma, R Helm, R. Johnson, J. Vlissides. (1994) Design Patterns: Elements of reusable object-oriented software, Allyson Wesley, Hawthorne, New York. ISBN 978-0201633610

[10] Garlan D., Shaw M. January 1994. An Introduction to Software Architecture.. CMU-CS-94-166. School of Computer Science. Carnegie Mellon University. Pittsburgh, PA 15213-3890.

[11] F. Bachmann, L Bass, P. Clements, D. Garlan, J. Ivers, R Little, Robert Nord, and Judy Stafford (2011). Documenting Software Architectures: Views and Beyond, 2da Edicion, Pearson Education, Westford, Massachusetts. ISBN-13: 978-0-321-55268-6.

[12] Lenz, Gunther y Wienands (2006), Christoph. Practical Software Factories in .NET. 1ra edición. Springer-Verlag New York Inc. ISBN-13: 978-1-59059-665-4.

[13] Lori MacVittie (2016). On monoliths versus microservices, F5 Networks, Inc. https://f5.com/Portals/1/Cache/Pdfs/5041/on-monoliths-versus-microservices.pdf

[14] M. Flowers, J Lewis. (2014). Microservices. 25 March 2014, de MartinFlowers.com Sitio web: http://martinfowler.com/articles/microservices.html

[15] Millett, Scott (2015). Patterns, Principles, and practices of DDD, 1a edición, John Wiley & Sons, Inc. Indianapolis, Indiana. ISBN- 978-1-118-71470-6.

[16] Nadareishvili, I., Mitra. R., McLarty M., Amundsen M. Microservice Architecture: Aligning Principles, Practices, and Culture, 1ra edición. Sebastopol, Ciudad de California: O'Reilly Media, Inc. ISBN-13: 978-1491956250.

[17] Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems, 1ra edición. Sebastopol, Ciudad de California: O'Reilly Media, Inc. ISBN-13: 978-1491950357.

[18] Richard, M (2016), Microservices Vs. Service Oriented Architecture, Primera edición, 1005 Gravenstein Highway North, Sebastopol, CA 95472,

O'Reilly Media, Inc ISBN 978-1-491-94161-4

[19] Nick Rozanski, Eion Woods (2011). Software Systems Architecture: Working with Stakeholders and viewpoints, 2da Edicion, Allyson Wesley, Westford, Massachusetts. ISBN-13: 978-0-321-71833-4.

[20] Oquendo F., Leite J., Batista T. Software Architecture in Action: Designing and Executing Architectural Models with SysADL Grounded on the OMG SysML Standard (Undergraduate Topics in Computer Science), 1ra edición. Springer. Switzerland. ISBN-13 978-3319443379

[21] Rosen, M. Lublinsky, B. Smith, K (2008). Applied SOA: Service-Oriented Architecture and Design Strategies. 1ra edición. Wiley Publishing, Inc. Indianapolis. ISBN-978-0-470-22365-9.

[22] Sommerville, I. (2012). Ingeniería de Software, 9na edición. Naucalpan de Juárez, Estado de México: Pearson Educación de México, S.A. de C.V. ISBN-13: 978-6073206037.

[23] Sommerville, I. (2015). Software Engineering, 10ma edición. England: Pearson Education Limited. ISBN-13: 978-0133943030.

[24] Stephens, R. (2015). Beginning Software Engineering, 1ra edición. Indianápolis, Indiana: John Wiley & Sons, Inc. ISBN-13: 978-1118969144.

[25] Stephen D. Burd (2011). Systems Architecture, Sixth Edition, Boston, MA. ISBN-13 978-0-538-47533-4

[26] Tanenbaum, A. S., Van Steen, M. Distributed Systems: Principles and Paradigms, 2da edición. Upper Saddle River, Nueva Jersey: Pearson Education, Inc. ISBN-13: 978-0132392273.

[27] Vernon, Vaughn (2013), Implementing Domain Driven Design, 1a edición. Pearson Education, Inc. Westford, Massachusetts. ISBN-13: 978-0-321-83457-7

[28] Wolff, E. (2016). Microservices: Flexible Software Architecture, 1ra edición. Reading, Massachusetts: Addison-Wesley. ISBN-13: 978-0134602417.

[29] G. G. Maigua (2012), Buenas Prácticas en la Gestión y Dirección de Proyectos Informáticos, Facultad Regional Tucumán Universidad Tecnológica Nacional – U.T.N. Argentina ISBN: 978-987-1896-01-1

[30] PMI (2013), GUÍA DE LOS FUNDAMENTOS PARA LA DIRECCIÓN DE PROYECTOS 5ta Edición, Dirección de Proyectos. I. Project Management Institute. II. Título: Guía del PMBOK. ISBN 978-1-62825-009-1

[31] F. G. Meza (2015), Introduccion a la Ingenieria Industrial 1ª Edición, Universidad Continental Jr. Junín 355, Miraflores, Lima-18

[32] M. S. Figueroa (2001), Gestión integrada de proyectos, Edicions de la Universitat Politècnica de Catalunya, SL, ISBN: 84-8301-453-X

[33]Dirk Krafzig, Karl Banke, Dirk Slama (2004), Enterprise SOA: Service-Oriented Architecture Best Practices (Coad), Prentice Hall, First Edition, ISBN-13: 978-0131465756

[34]Len Bass Paul Clements RickKazman (2012),Software Architecture in Practice 3th Edition, Addison-Wesley ,Westford, Massachusetts, ISBN 978-0-321-81573-6