Integración del patrón de bandeja de salida transaccional en arquitecturas de microservicios basadas en SAGA

Integration of the Transactional Outbox Pattern in SAGA-Based Microservices Architectures

Carlos Peliza[1] y Leandro Alfaro[1]

Ingeniería en Informática/ Desarrollo tecnológico

Citar: Peliza, C. y Alfaro, L. (2025). Integración del Patrón de Bandeja de Salida Transaccional en Arquitecturas de Microservicios basadas en SAGA. *Cuadernos de Ingeniería* (16). http://revistas.ucasal.edu.ar

Recibido: junio/2025
Aceptado: setiembre/2025

Resumen

El proyecto de investigación "Desarrollo de un sistema de gestión de un sistema FTTH" por parte de alumnos de la Universidad Nacional de La Matanza (UNLaM), bajo la premisa de enfrentar problemas de diseño reales en un entorno controlado, ha obligado al grupo de trabajo a reconsiderar las opciones de diseño elegidas.

Este artículo presenta una propuesta de rediseño arquitectónico de un sistema de gestión Fiber to the Home (FTTH). Se analiza la transición de una arquitectura sincrónica basada en API Gateway hacia una arquitectura basada en eventos con comunicación asincrónica. La solución se apoya en la aplicación combinada del patrón SAGA y del Transactional Outbox Pattern (TOP), con el objetivo de garantizar consistencia y resiliencia en transacciones distribuidas. Se discuten beneficios, desafíos y recomendaciones para su implementación efectiva.

Palabras claves: Microservicios, SAGA, Outbox, Kafka, arquitectura de software, FTTH, transacciones distribuidas

Abstract

The research project "Development of a Management System for an FTTH System" conducted by UNLaM students, under the premise of addressing realworld design problems in a controlled environment, forced the working group to reconsider their chosen design options.

This article presents a proposed architectural redesign of an FTTH (Fiber to the Home) management system. The transition from a

[1] Departamento de Ingeniería e Investigaciones Tecnológicas - Universidad Nacional de La Matanza (UNLaM). San Justo, Argentina.

synchronous API Gatewaybased architecture to an eventbased architecture with asynchronous communication is analyzed. The solution relies on the combined application of the SAGA pattern and the Transactional Outbox Pattern (TOP), with the goal of ensuring consistency and resilience in distributed transactions. Benefits, challenges, and recommendations for its effective implementation are discussed.

Keywords: Microservices, SAGA, Outbox, Kafka, software architecture, FTTH, distributed transactions

1. Introducción

La creciente complejidad de los sistemas distribuidos y la necesidad de escalabilidad han llevado a la adopción de arquitecturas de microservicios. Inicialmente, el sistema bajo estudio implementó una arquitectura basada en API Gateway y comunicación HTTP sincrónica, que en la realidad de la implementación evidenció limitaciones en eficiencia.

Inicialmente, se seleccionó el patrón API Gateway como estrategia de enrutamiento y control de acceso. Un punto de entrada o salida llamado Gateway se sitúa entre las aplicaciones de clientes y los microservicios, en tanto que funciona como un intermediario entre ellos y como un *proxy* inverso que redirige las solicitudes del cliente a los servicios y la aplicación (Richardson, 2019, p. 259).

La utilización de API Gateway permite que el usuario final se abstraiga de visibilizar a cuál microservicio se direcciona su consulta, solamente le manda una petición HTTP (que es un protocolo de comunicación sincrónico) a un servidor simple (API Gateway) y este se encarga de ver a qué servidor corresponde enviar la información. El usuario final permanece abstraído del direccionamiento interno entre microservicios, ya que la API Gateway actúa como intermediario transparente. De la teoría de la evolución de sistemas de *software* de tipo E (aquellos que interactúan con un entorno real) (Lehman et al., 1985) podemos extraer ocho leyes; sin embargo, hemos de profundizar en aquella que nos ha obligado a reconsiderar la elección de arquitectura. El autor mencionado habla de calidad decreciente (*Declining Quality*) para el caso donde la calidad de un sistema de tipo E se percibirá como decreciente a menos que se mantenga rigurosamente y se adapte a los cambios en el entorno operativo.

A partir de las definiciones propuestas por el cliente ISP FTTH, se establecieron como necesarios los siguientes microservicios:

msvc OLT:

- integración con la OLT (integraciones disponibles con las OLT ZTE (Batna24, s. f.) y Huawei) y obtención de datos crudos
- Enviar comandos a la OLT y las ONT
- gestión de placas de la OLT
- gestión vlans de servicios

msvc ONT:

- gestionar tipos de ONT: tipos de PON, tipo de puertos LAN, ACCESS, etc.
- Tipos de DHCP, perfiles de velocidad, tipo de perfil de velocidad
- configuración de ONT en relación con la OLT

- gestionar solicitudes de conexión de nuevas ONT
- procesar datos crudos de la OLT

msvc ODB:

- gestionar zonas
- gestionar las ODB
- ubicación de las ONT

msvc auth:

- gestionar usuarios + credenciales
- roles y permisos

msvc clientes:

- gestionar clientes y empresas
- facturación y suscripciones mensuales con vencimiento
- gestión solicitudes de pago (si se puedo facturar + reintentos)

El desarrollo inicial del sistema se esquematiza en la Figura 1, que muestra una cantidad de OLT[2] a gestionar de manera remota, que fue cambiando de una única unidad a 3 unidades, con el aumento de clientes de la empresa ISP FTTH, de 4000 a 12 000 aproximadamente.

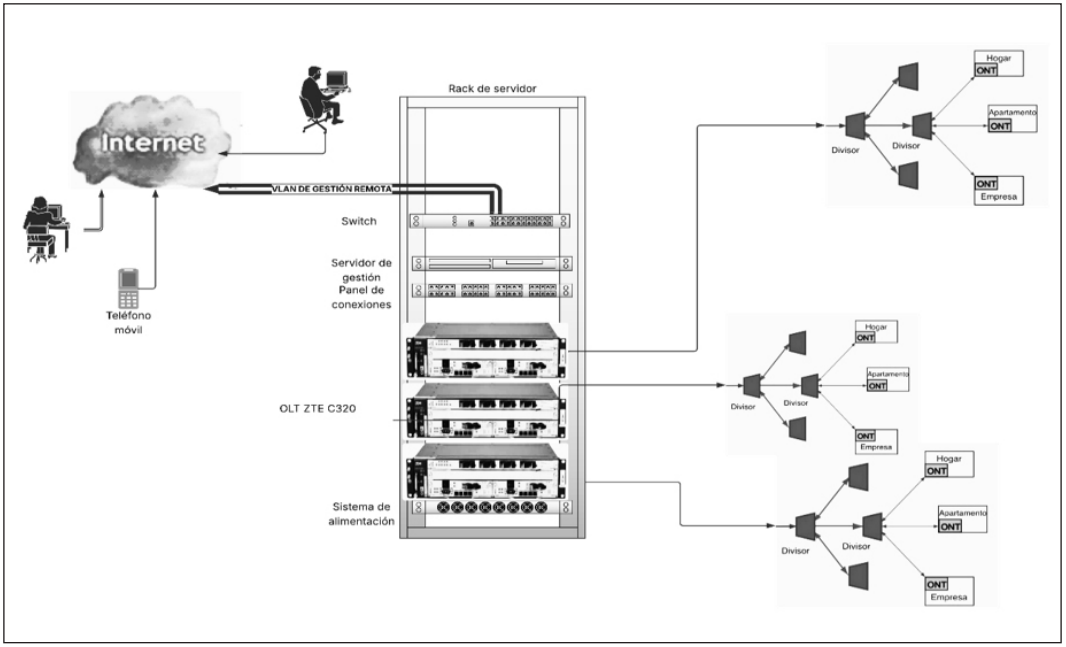

Figura 1. Esquemático físico de la solución con un único servidor de gestión.

[2] Se inició con una OLT ZTE CX320 pero, luego, el crecimiento del ISP sumó al desarrollo del gestor otras marcas de OLT, como ser Huawei.

En el entorno operativo controlado, el incremento de condiciones de operación junto con la necesidad de interacción con otros sistemas (como ser control de *stock*, sistemas de pago y gestión de visitas técnicas) obligó a abandonar la comunicación HTTP (evento sincrónico) por las demoras y la creciente detección de fallas.

2. Metodología

Este trabajo adopta un enfoque metodológico de tipo estudio de caso técnico aplicado (Otero, 2012) junto con un caso de prototipado guiado por problemas (Mackay y BeaudouinLafon, 2023), con enfoque de investigación evaluativoexploratoria, basado en la evolución arquitectónica de un sistema real en desarrollo.

El abordaje fue realizado en un entorno controlado, con el objetivo de explorar y validar la implementación de patrones arquitectónicos modernos en una solución FTTH. El proceso metodológico incluyó:

- Un diagnóstico técnico de la arquitectura inicial basada en API Gateway y comunicación HTTP sincrónica.
- La reestructuración del sistema mediante una arquitectura basada en eventos, incorporando los patrones SAGA y Transactional Outbox Pattern (TOP), y empleando Apache Kafka como bus de eventos.
- La definición de flujos críticos del negocio para aplicar variantes de orquestación y coreografía, según el nivel de acoplamiento y criticidad del servicio.

Se trató de una validación parcial bajo condiciones controladas, en vistas a su futura evaluación bajo pruebas de estrés.

3. Problema y justificación

Un protocolo sincrónico como el elegido significa que una solicitud HTTP espera la respuesta del servidor antes de continuar con la siguiente acción. La situación antes mencionada trajo aparejados los siguientes problemas:

- Comunicación sincrónica compleja, ya que cada servicio tiene que conocer la ubicación exacta de los microservicios con los que se comunica.
- Un servicio puede afectar a otro y eso representa una complejidad para migraciones / actualizar referencias.
- No es flexible (no soporta cambios de negocio sin incrementar el número de fallas)
- Todos los microservicios de un flujo tienen que estar OK para funcionar, lo que implica que se pueden perder peticiones por errores 5xx.

La Figura 2 presenta en conjunto dos de las dificultades que pueden ocurrir cuando se usa comunicación sincrónica. La representación simplificada con pocos microservicios expone con claridad cómo el aumento de nodos de microservicios aumenta la demora en respuestas, la complicación y la posibilidad de fallas.

En síntesis, los problemas de acoplamiento, la baja disponibilidad y la poca escalabilidad del sistema basado en API Gateway motivaron la transición hacia una arquitectura guiada por eventos con comunicación asincrónica mediante Apache Kafka. Sin embargo, este proceso de cambio implica desafíos técnicos significativos y requiere una cuidadosa evaluación de decisiones de diseño que se van a enfrentar con una cambiante realidad. Los requerimientos de negocio tienden a evolucionar continuamente, lo cual exige arquitecturas que puedan adaptarse dinámicamente a nuevos escenarios funcionales y, en un caso real, el desconocimiento del grupo de diseño puede comprometer la viabilidad del proyecto debido a errores de diseño no anticipados.

Por lo expuesto, aparece como necesario que siempre se elija un representante del negocio para fijar puntos claros que delimiten el trabajo a encarar por el grupo de diseño.

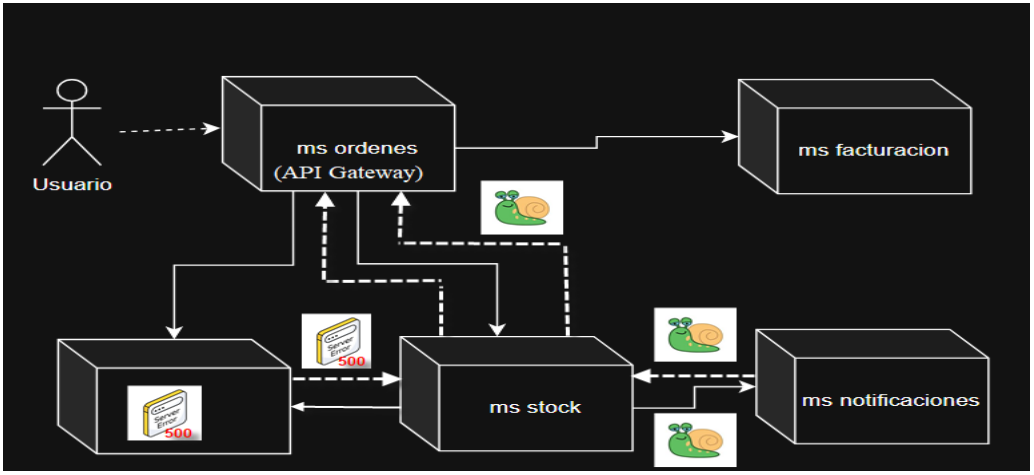

Figura 1. Posibles demoras por el uso de sincronismo en la comunicación de eventos.

En resumen, la complejidad del sistema obligó a mutar de la arquitectura de microservicios basada en HTTP (comunicación sincrónica) hacia la arquitectura de microservicios guiada por eventos que se basa en comunicación asincrónica (Fowler y Lewis, s. f.).

Para mitigar estos problemas, que se incrementaban con el aumento de clientes y ONT u OLT y situaciones asociadas a gestionar, se exploró la adopción del patrón SAGA para coordinar transacciones distribuidas, y del patrón de bandeja de salida transaccional asincrónico (del inglés *transactional outbox pattern*) que en adelante llamaremos TOP, para garantizar la entrega fiable de eventos.

4. Patrón SAGA

El patrón SAGA divide una transacción distribuida en una serie de transacciones locales, cada una ejecutada por un microservicio. Existen dos variantes principales:

- SAGA orquestada: un orquestador central controla el flujo.
- SAGA coreografiada: cada servicio reacciona a eventos de otros.

5. Patrón de bandeja de salida transaccional (TOP)

Este patrón consiste en escribir eventos en una tabla especial dentro de la misma transacción que afecta al modelo de negocio. Posteriormente, un componente externo publica esos eventos en un sistema de mensajería (p. ej., Kafka), garantizando consistencia sin usar TwoPhase Commit (Compromiso en dos fases o 2PC), un protocolo clásico de las bases de datos distribuidas (Desai y Boutros, 1996).

6. Propuesta de arquitectura

Se propone una arquitectura basada en eventos donde los servicios utilizan el patrón TOP para publicar eventos y se coordinan mediante SAGA. Las decisiones de orquestación o coreografía se toman según la complejidad del flujo de microservicio a satisfacer (Rudrabhatla, 2018):

SAGA orquestada para procesos críticos, como facturación.

SAGA coreografiada para flujos descentralizados, como reclamos técnicos.

Debido a que ambos patrones SAGA son combinables, se propone una arquitectura que los combina en función de las necesidades del negocio, por ello hemos de asignar el patrón SAGA orquestado al sector de cobros y facturación por su crecimiento muy estable (el patrón SAGA orquestado requiere una coordinación compleja con las reglas de negocio). En cambio, para el sector de reclamaciones y operación técnica de respuesta automatizada, el patrón SAGA coreografiado resulta más adecuado por su capacidad de escalabilidad y bajo acoplamiento (los reclamos técnicos suelen enfrentar picos de uso de recursos).

La implementación incluye: uso de Kafka como bus de eventos, TOP por servicio, Poller o CDC (Debezium) (Morling, 2019) para publicar eventos y mecanismos de idempotencia en productores/consumidores.

La combinación de SAGA + TOP mejora la confiabilidad de los flujos distribuidos. La idempotencia en productores y consumidores es crítica. Se destaca la necesidad de depurar flujos en SAGA coreografiada, mediante un correcto registro con logs.

Por otro lado, Change Data Capture (CDC) es una técnica que detecta cambios en una base de datos (por ejemplo, inserciones en una tabla TOP) y los propaga a otros sistemas —como Kafka— en tiempo real o casi en tiempo real. Es comúnmente utilizada para publicar eventos de forma confiable y desacoplada, sin necesidad de implementar un sistema de consultas manual.

En el contexto del trabajo de campo realizado, la adopción de CDC complementa el patrón TOP, lo cual permite que los eventos escritos en la base de datos sean publicados automáticamente (por ejemplo, con Debezium).

7. Definiciones teóricas adoptadas

Ha resultado claro que una estructura de microservicios interconectados entre sí no resultó en una propuesta viable a largo plazo para nuestro desarrollo. Por tal motivo el cambio al llamado modo natural, donde Kafka actúa como un intermediario (bróker) entre productores (quienes generan datos o eventos) y consumidores (quienes procesan esos datos), mediante un modelo de publicación y suscripción (pub/sub).

En este sentido aparecen nuevas funciones que debieron asignarse: la de bróker, o sea, un nodo del clúster Kafka que almacena y distribuye los mensajes. También, las funciones de Productor/Consumidor de eventos que son quienes envían mensajes o leen mensajes desde Kafka. Inclusive habrá nodos que cumplan la doble función de productores y consumidores de eventos.

Adicionalmente, se definieron tópicos, que son categorías o canales al que los mensajes se publican y es una manera de organizar datos en el entorno distribuido Kafka. En este punto, hemos mencionado reiteradas veces "clúster" y se hace necesario explicar que es un conjunto de computadoras llamadas nodos que, interconectadas, trabajan juntas como si fueran un solo sistema con un objetivo en común. Esta interconexión debe ser coordinada.

En versiones previas de Kafka la coordinación se hacía mediante ZooKeeper, que se encargaba de almacenar metadatos del clúster, coordinar elecciones de líderes y registrar el estado de los bróker, tópicos y particiones. Actualmente, Kraft es el reemplazo de ZooKeeper y usa un algoritmo de consenso llamado Raft. Con este algoritmo, varios nodos pueden acordar el mismo estado sin conflictos.

KRaft usa el algoritmo distribuido Raft para elección de bróker líderes, replicación de registros (logs) de metadatos y mantener la consistencia del estado de la estructura coordinada.

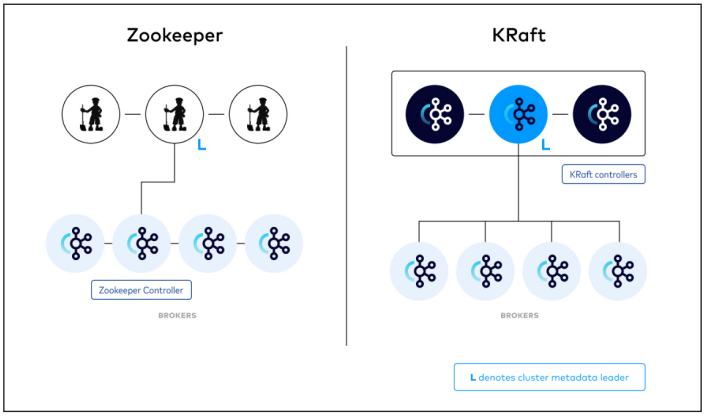

Figura 2. Diferencias entre ZooKeeper y KRaft (obtenido de Confluent Developer).

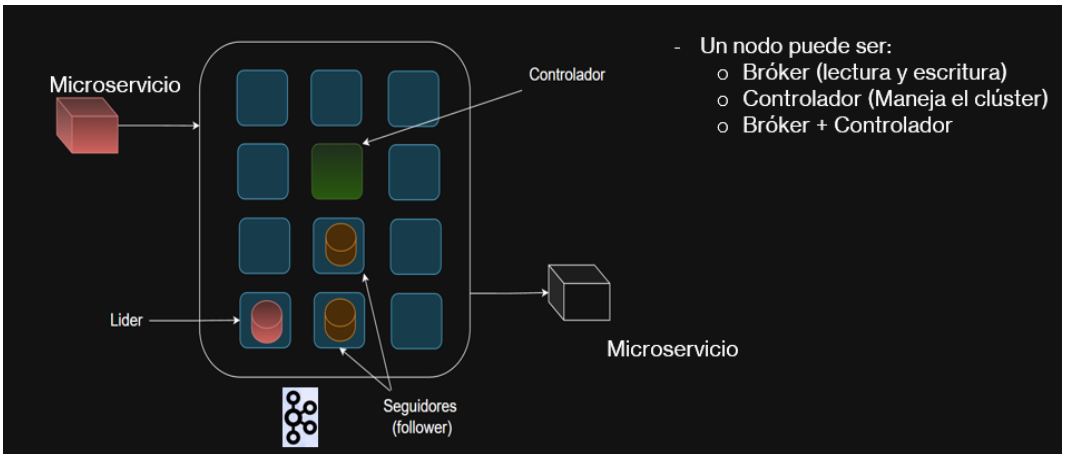

Figura 3. Esquema de clúster de nodos con controlador de bróker usando Kraft.

La Figura 4 plantea el mecanismo mediante el cual un microservicio determina el bróker adecuado para acceder al tópico correspondiente como publicador o consumidor, dentro del clúster y cuál es el bróker que contiene el tópico que necesita. Para ello se define el término Bootstrap server que se refiere a la lista inicial de bróker (nodos Kafka) que un cliente (productor o consumidor) necesita conocer para conectarse al clúster Kafka.

Este parámetro corresponde a la dirección IP o al nombre de *host* y puerto de uno o más bróker Kafka, que el cliente utiliza para descubrir la infraestructura distribuida; por lo tanto, no se necesita incluir todos los bróker de un nodo Kafka dentro de la red distribuida. Solo uno válido es suficiente para que el cliente descubra el resto.

Este mecanismo funciona igual en Kafka tradicional (con ZooKeeper) y en Kafka con Kraft, pero la diferencia es que en modo KRaft, en el plano de control también existe un componente especial llamado el "KRaft Controller", que gestiona los metadatos del clúster.

Entonces, un cliente que en ZooKeeper debía conocer Bootstrap server y tópico para acceder al nodo, en Kraft solo necesita saber el Bootstrap server y de manera interna se lo redirige al bróker que le brindará servicios otorgándole la metadata inicial del tópico buscado. Esta forma de trabajo permite que la red de nodos sea dinámica, mientras que en ZooKeeper debían permanecer estáticos. De esta manera solucionamos el problema del error 500 de la Figura 2, porque cualquier bróker que se halle fuera de servicio va a salir del clúster y el plano de control de Kraft designará un reemplazo. No obstante, este enfoque no soluciona por completo la duplicación de mensajes, lo cual exige la implementación de mecanismos complementarios, por lo cual han de recordarse los conceptos referidos a productor/consumidor idempotente.

Un productor idempotente es aquel que asegura que un mismo mensaje no se publique más de una vez, incluso si el envío se repite debido a errores de red o reintentos (esto lo soluciona el propio Kafka desde la versión 0.11 y es un parámetro configurable; mientras que un consu-

midor idempotente es aquel que puede procesar el mismo mensaje varias veces sin producir efectos secundarios adicionales. Esta situación la solucionan las habilidades del desarrollador y las soluciones posibles son verificar si una operación ya se ejecutó antes (por ejemplo, usando un message_id), utilizar bases de datos con claves únicas o transacciones o registrar eventos procesados en una tabla de duplicación.

8. Conclusiones

El uso combinado de los patrones SAGA y Outbox ofrece una solución robusta para transacciones distribuidas. La selección entre coreografía u orquestación debe basarse en el contexto de negocio. La adopción de CDC y mecanismos de idempotencia complementan la solución. No debe descartarse la adopción de patrones híbridos donde se combinen los beneficios de cada uno de ellos para cada conjunto de microservicios que se pretende desarrollar.

Los próximos pasos del equipo de desarrollo consistirán en efectuar un testeo en condiciones de estrés controlado, para poder conocer la eficiencia de la solución elegida.

Referencias

Batna24. (s. f.). *ZTE C320 | OLT | GPON ZXA10, 1x GTGO, 2x SMXA*. www.batna24.com/pl. https://www.batna24.com/es/p/zte-c320-olt-rmmhh

Desai, B. C., y Boutros, B. S. (1996). Performance of a two-phase commit protocol. *Information and Software Technology*, *38*(9), 581-599. https://doi.org/10.1016/0950-5849(95)01096-3

Fowler, M., y Lewis, J. (s. f.). *Microservices*. martinfowler.com. https://martinfowler.com/articles/microservices.html

Lehman, M. M., Belady, L. A., y Lehman, M. M. (Eds.). (1985). *Program evolution: Processes of software change*. Academic Press.

Mackay, W. E., y Beaudouin-Lafon, M. (2023). Participatory Design and Prototyping. En J. Vanderdonckt, P. Palanque, y M. Winckler (Eds.), *Handbook of Human Computer Interaction* (pp. 1-33). Springer International Publishing. https://doi.org/10.1007/978-3-319-27648-9_31-1

Morling, G. (19 de febrero 2019). *Reliable Microservices Data Exchange With the Outbox Pattern*. Debezium. https://debezium.io/blog/2019/02/19/reliable-microservices-data-exchange-with-the-outbox-pattern/

Otero, C. E. (2012). *Software engineering design: Theory and practice*. CRC Press.

Richardson, C. (2019). *Microservices patterns: With examples in Java*. Manning Publications.

Rudrabhatla, C. K. (2018). Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture. *International Journal of Advanced Computer Science and Applications*, *9*(8). https://doi.org/10.14569/IJACSA.2018.090804

Carlos Peliza
Perfil académico y profesional: Ingeniero electrónico. Magister en telecomunicaciones y en Gestión de la Educación Superior. Docente investigador (UNLAM).
Correo electrónico: cpeliza@unlam.edu.ar
https://orcid.org/0000-0002-2901-185X

Leandro Alfaro
Perfil académico y profesional : Ingeniero en Informática (Universidad Nacional de La Matanza - UNLaM).
Correo electrónico: ealfaro@alumno.unlam.edu.ar