Integración por software de las plataformas Nagios y OTRS para automatizar el registro de incidentes en la empresa STR

Software integration of Nagios and OTRS platforms to automate the recording of incidents in STR

Alvarado Girón Juan Manuel[1]
juan.alvarado03@usc.edu.co

Universidad Santiago de Cali, Facultad de Ingeniería, Programa de Ingeniería de sistemas (1)

Resumen
Las necesidades del negocio y de la vida cotidiana, han demandado que los sistemas de información sean integrables, es decir, que un sistema de información pueda interactuar con otros sistemas de información.
En este artículo, se trabajó un caso particular de la empresa STR (Smart Technology for Receiving), que tiene sede en la ciudad de Santiago de Cali. STR se dedica a operar equipos tecnológicos para el transporte, entre estos, equipos de recaudo.
El caso particular fue el registro manual de incidentes por fallos de equipos, lo cual afectaba el nivel de prestación de servicio.
La solución propuesta fue una automatización del proceso de registro de incidentes, mediante intercambio de datos por servicio web SOAP. Esto implicó el desarrollo de un *componente de software* que requirió la solidez de un modelo relacional y la consistencia de un diagrama de clases. También, fue necesario estudiar el API de Nagios IV, el WSDL del servicio web OTRS.
Para medir el impacto de solución, se definieron una serie de variables, las cuales fueron alimentadas por el *componente de software* mencionado. Así, se pudo cuantificar el problema y ver la mejoría del nivel de prestación de servicio (en cuanto a reducción de tiempo).

Palabras Clave: Servicio web; Integración de plataformas; SOAP; Nagios; OTRS

Abstract
The needs of business and everyday life have asked the information systems are integrable, i.e. an information system can interact with other information systems.
In this paper, a particular case of STR Company (Smart Technology for Receiving) was worked. STR is headquartered in Santiago de Cali. STR operates technological transport equipment, between these, collection equipment.
The particular case was the manual record of incidents due to equipment failure, which affected the level of service provision.
The proposed solution was an automation of the incident registration process by the exchange of data through SOAP Web Service.
This implied the development of a software component, which required the solidity of a relational model and the consistency of a class diagram. Also, studying Nagios API and WSDL of OTRS Web Service was necessary.
For measuring the impact of the solution, several variables were defined, these variables were fed by the above software component. So, the problem could be quantified and the improvement of the level of service provision could be seen (in terms of time reduction).

Keywords: Web Service; platforms integration; SOAP; Nagios; OTRS

1. INTRODUCCIÓN

La cantidad de equipos gestionados y monitoreados por STR es alrededor de 1.200, los cuales tienen sus respectivos servicios. Esta cantidad de equipos crece año tras año, debido a las necesidades y retos de la operación tecnologica de STR.

Para monitorear equipos, STR tiene un servidor con el software Nagios Core IV. Para registrar incidentes, STR tiene un servidor con el software OTRS 5.

La forma de monitorear los equipos y registrar las alertas se realizaba así: las alertas eran detectadas de forma visual por el personal de monitoreo, quien las registraba en OTRS, después, se reportaba al técnico en campo para su solución. Esta forma de operar tiene algunas falencias: alertas detectadas tarde, alertas omitidas y alertas no registradas.

Debido a estas falencias, se generó la siguiente hipotesis: *la tarea manual para registrar alertas afecta los indicadores de servicio porque la tarea queda sujeta a omisiones humanas*, además, la cantidad de equipos-servicios es grande y tiende a aumentar, por ende, la probabilidad de omisión humana y la necesidad de más personal aumentan.

En el proyecto se propuso como solución, *la integración por software de las plataformas OTRS 5 y Nagios IV para automatizar el registro de alertas*. La integración fue desarrollada mediante el lenguaje Perl (codificación del algoritmo) y el lenguaje SQL (modelo de datos), a su vez, se usó el protocolo SOAP para intercambiar datos entre plataformas.

El desarrollo de la integración fue descompuesto en las siguientes fases:

- Definición del protocolo de comunicación para el intercambio de datos.

- Definición del mecanismo de captura de alertas que dispara el proceso automático.
- Diseño del diagrama de clases (Modelo y Controlador) y el modelo relacional del problema.
- Diseño interfaz de administración (Vista).

Después de implementar la integración, se analizaron datos obtenidos durante tres meses. Estos datos fueron capturados desde la propia integración, así, cada registro de alerta y su detalle era almacenado en una base de datos. También, se definieron varios procedimientos almacenados que consolidaban los datos para su analisis.
Los datos generados permitieron cuantificar el problema, encontrar puntos críticos de alertas y medir el progreso de los indicadores de servicio.
Al analizar la información, se evidenció una reducción en el tiempo de respuesta, debido a que, la alerta era registrada automáticamente e informada en tiempo real, así, se brindaron condiciones de operación óptimas para mejorar los indicadores de servicio.
El resto del documento está organizado así:

- La segunda sección es una revisión de la metodología para cuantificar y medir el impacto de la solución, también, se presenta el diseño lógico e implementación de la solución.
- La tercera sección presenta los resultados de la solución.
- La cuarta sección concluye el documento.

2. MATERIALES Y MÉTODOS

2.1 Enfoque y tipo de la investigación

El proyecto tuvo un enfoque cuantitativo, porque, el problema requirió variables numéricas como: tickets generados por día mediante automatización, tickets generados por día y zona, tickets generados por día y servicio, tickets generados por hora y servicio, promedio tiempo de respuesta por día Gral, promedio tiempo de respuesta por día y zona, promedio tiempo de respuesta por día y servicio.
El tipo de la investigación es de carácter aplicado, porque, se buscó la solución de un problema práctico.

2.2 Población

Todos los equipos monitoreados de la empresa (equipos de recarga, servidores, validadores, consultores de saldo, equipos a bordo de buses, cámaras, etc.) son 1200, cada uno con sus respectivos servicios.

2.3 Muestra

El proyecto fue de enfoque cuantitativo y con cantidad de población conocida, así, se tomó una muestra de acuerdo a la ecuación de la *Figura 1*:

$$n = \frac{Z^2 N PQ}{(N-1)E^2 + Z^2 PQ}$$

Figura 1. **Ecuación para la muestra de una población finita.**
Fuente: (Martinez, 2012, pág. 306)

N = Número de población = 1200
Z = Nivel de confianza del 95% = 1,96
d = error muestral del 5% = 0,05
p = 0,95
q = 0,05

La muestra debe ser de 69 servicios con un nivel de confianza de 95% y un margen de error del 5%.
Los 69 servicios están distribuidos en 23 máquinas de recarga automática de estaciones, que tienen 3 servicios cada una.

2.4 Técnica e instrumentos de recolección de datos

2.4.1 Sistemas de información

Todos los datos fueron guardados en bases de datos relacionales, los sistemas de información implicados son:

- Nagios Core IV (MySQL).
- OTRS 5 (Oracle).
- Propio desarrollo de la integración (la aplicación que servirá para integrar las dos plataformas) (MySQL).

El procesamiento de datos está sustentado en datos reales tomados de una fuente primaria.

2.4.2 Variables de estudio y técnica de procesamiento y análisis de datos

Las variables de estudio permitieron cuantificar el problema y medir el impacto de la solución; el propósito específico de cada variable es descrito en la *Tabla 1*:

Tabla 1
Variables de estudio

Variable	Motivo de medida	Método de medida	Medición
Tickets generados por día mediante automatización	Mostrar la eficiencia de la solución.	Procedimiento almacenado SQL programado para ejecución diaria.	Intervalo
Tickets generados por día y zona	Clasificar las alertas por zonas, esto ayudaría a distribuir mejor el personal en campo.	Procedimiento almacenado SQL programado para ejecución diaria.	Ordinal
Tickets generados por día y servicio	Clasificar las alertas por servicio, esto ayudaría en la planeación de la compra de repuestos y focalizar los esfuerzos.	Procedimiento almacenado SQL programado para ejecución diaria.	Ordinal
Tickets generados por hora y servicio	Clasificar las alertas por hora, esto ayudaría a identificar horarios críticos para focalizar los esfuerzos en campo.	Procedimiento almacenado SQL programado para ejecución cada hora.	Intervalo
Promedio tiempo de respuesta por día Gral.	Mostrar la situación actual de respuesta y como, el proyecto contribuye en la disminución del tiempo de respuesta (Solución del problema).	Procedimiento almacenado SQL programado para ejecución diaria.	Razón
Promedio tiempo de respuesta por día y zona	Mostrar el progreso en cuanto a la distribución de esfuerzos en campo.	Procedimiento almacenado SQL programado para ejecución diaria.	Razón
Promedio tiempo de respuesta por día y servicio	Mostrar el progreso en cuanto a la planeación y disponibilidad de repuestos.	Procedimiento almacenado SQL programado para ejecución diaria.	Razón

Fuente: Elaboración propia.

2.5 Estrategia para resolver el problema

Para desarrollar la solución fue conveniente dividir cada fase en tareas, la *Tabla 2* muestra la división lógica (en orden descendente):

Tabla 2

Fases y tareas de la integración

Fase	Tarea
Definición del protocolo de comunicación para el intercambio de datos	Configurar el Web Service OTRS como proveedor (SOAP).
Definición del mecanismo de captura de alertas que dispara el proceso automático	Definir los comandos y objetos Nagios.
Diseño del diagrama de clases (Modelo y Controlador del MVC) y modelo relacional del problema	Conceptuar el modelo relacional del problema.
	Traducir el modelo relacional a almacenamiento físico (SQL).
	Diseñar las clases que solucionan el problema (Modelo del MVC).
	Diseñar el algoritmo para crear caso automático (Controlador del MVC).
Diseño interfaz de administración (Vista del MVC)	Diseñar el sistema de configuración del aplicativo (XML).
	Diseñar la interfaz gráfica del aplicativo (HTML, Ajax).
	Configurar servicios y colas OTRS Vs alertas Nagios.

Fuente: Elaboración propia.

2.6 Definición del protocolo de comunicación y el mecanismo de captura de alertas (Fases 1 y 2)

La primera fase fue definir el protocolo de comunicación para intercambiar datos. Esta información muestra el contexto tecnologico de las plataformas, es decir, muestra los estandares y protocolos a usar. Así, se definió que las plataformas trabajan en un modelo cliente-servidor bajo el protocolo TCP/IP.

2.6.1 El protocolo TCP/IP, la estructura para la comunicación

El protocolo TCP/IP define las reglas que rigen la sintaxis (que puede ser comunicado), semántica (como puede ser comunicado) y la sincronización (cuando y a qué velocidad puede ser comunicado).

La tarea de comunicación TCP/IP está dividida en capas. Cada capa tiene una función y propósito diferente durante la comunicación.

¿Cómo se comunican muchas aplicaciones mediante una sola conexión de red? Los host tienen un solo puerto de comunicación activo, que es la interfaz de red. Si una aplicación está usando un puerto, otra aplicación no puede usar el mismo puerto. Por este motivo, surgió la necesidad de tener múltiples vías en la comunicación de una sola interfaz de red (Goralski, 2017, pág. 24), estas vías se denominan puertos, así, una interfaz de red tiene asignado 65535 puertos, de los cuales, se asigna uno por aplicación.

La tarea de comunicación mediante las capas es semejante a la escritura y envío de una carta. La carta es enviada dentro de un sobre, que tiene datos sobre su emisor y receptor. El sobre es transportado dentro de un vehículo (Goralski, 2017, pág. 25).

La *Figura 2* muestra las capas TCP/IP; cada capa tiene protocolos independientes, que son usados para proporcionar funcionalidades. En TCP/IP, cada protocolo de la capa superior es soportado por los protocolos de la capa inferior (Goralski, 2017, pág. 26).

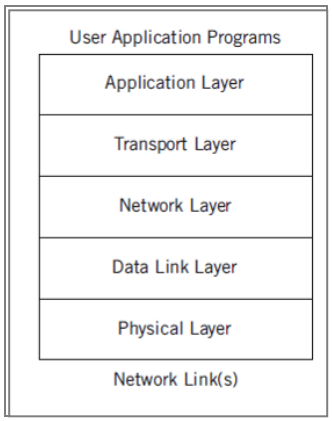

***Figura 2*. Capas de TCP/IP.**
Fuente: (Goralski, 2017, pág. 27)

Cada capa tiene una interface con las capas inmediatamente superior e inferior, la *Figura 2* muestra el orden descendente de las capas. La capa inferior provee servicios a la capa inmediatamente superior y consume servicios de su capa inmediatamente inferior. La capa superior envía los comandos, datos y parámetros a la capa inferior, la cual ejecuta la tarea correspondiente (Goralski, 2017, pág. 27).

2.6.2 XML, el lenguaje para el intercambio de datos

En la capa de aplicación hay un protocolo llamado SOAP, que es un esquema de mensajería basado en XML.
XML ó lenguaje extensible de marcas fue diseñado para guardar y transportar datos. Aun así, XML no hace nada.
En XML, los datos están encerrados dentro de marcas, que son palabras entre <>, ejemplo:

<marca>dato</marca>

Cada marca con dato es llamada elemento. Los documentos XML son formados como arboles de elementos, esto implica que un elemento puede tener elementos hijo:

<padre>
    <hijo></hijo>
</padre>

Los elementos pueden tener o no atributos, que son usados frecuentemente como metadato del elemento:

<padre nombre=”Pérez”></padre>

XML tiene un mecanismo para evitar ambigüedades de nombre de elementos, el mecanismo son *los espacios de nombre*, que son definidos como atributo del elemento raíz:

<padre *xmlns:espacionombre=”http://articulo.com”*>
    <espacionombre:hijo></espacionnombre:hijo>
<padre>

Todo documento XML debe tener un elemento raíz (Refsnes Data, 2018).

2.6.3 SOAP, el protocolo de comunicación

SOAP es un esquema de mensajería basado en XML. SOAP viaja mediante el protocolo HTTP en forma de documento XML (Gurugé, 2004, pág. 12).
La *Figura 3* muestra la estructura XML de un documento SOAP:

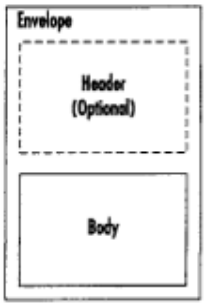

***Figura 3*. Mensaje SOAP.**
Fuente: (Foster, 2002, pág. 206)

Envelope es el elemento raíz del documento, por tanto, debe estar presente en cada mensaje SOAP. Envelope es el padre de Header y Body.
Todo mensaje SOAP requiere un espacio de nombre, que es definido en el elemento Envelope. El espacio de nombre debe ser parametrizado así:

URI: http://schemas.xmlsoap.org/soap/envelope
Prefijo: soap-env

SOAP establece que todos los elementos en un mensaje deben ser identificados por un espacio de nombre (Foster, 2002, pág. 205). El elemento Body es obligatorio y contiene datos de solicitud, datos de respuesta o datos de error.

Una transacción SOAP puede verse así:

Solicitud

```
<soap:Envelope xmlns:soap=”http://www.w3.org/2003/05/soap-envelope/”>
        <soap:Body>
                <m:GetPrice xmlns:m=”https://www.w3schools.com/prices”>
                        <m:Item>Apples</m:Item>
                </m:GetPrice>
        </soap:Body>
</soap:Envelope>
```

Respuesta

```
<soap:Envelope xmlns:soap=”http://www.w3.org/2003/05/soap-envelope/”>
        <soap:Body>
                <m:GetPriceResponse xmlns:m=”https://www.w3schools.com/prices”>
                        <m:Price>1.90</m:Price>
                </m:GetPriceResponse>
        </soap:Body>
</soap:Envelope>
```

2.6.4 Los Web Service, la interface de integración

Los Web Service están basados en XML y HTTP, por tanto, su uso (proveer y consumir) es realizado mediante la web; esta característica hace que los Web Service sean independientes de la plataforma (sistema operativo y lenguaje de programación) (Gurugé, 2004, pág. 9).
Los Web Service son un mecanismo de llamado a procesos remotos. La aplicación que ejecuta un proceso remoto, llama desde su lógica al Web Service, para esto, envía un documento con los parámetros de la solicitud y recibe un documento con la respuesta del proceso remoto. Este intercambio de datos es realizado usando documentos XML (Gurugé, 2004, pág. 11).

2.6.5 Configurar el Web Service OTRS como proveedor

Para configurar el web service, ingrese ala interfaz de administración OTRS, *Figura 4*:

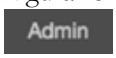

Figura 4. **Módulo administración OTRS**

Fuente: (OTRS AG, 2015, pág. 49)

En administración, abra el módulo para administrar web service, *Figura 5*. Aquí se gestionan todas las conexiones con sistemas remotos:

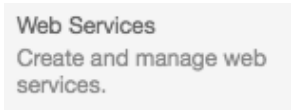

Figura 5. **Módulo creación y gestión Servicios Web OTRS**

Fuente: (OTRS AG, 2015, pág. 60)

Agregue y configure un servicio web de acuerdo a la *Figura 6*, el protocolo debe ser HTTP::SOAP. También, especifique el nombre de los métodos:

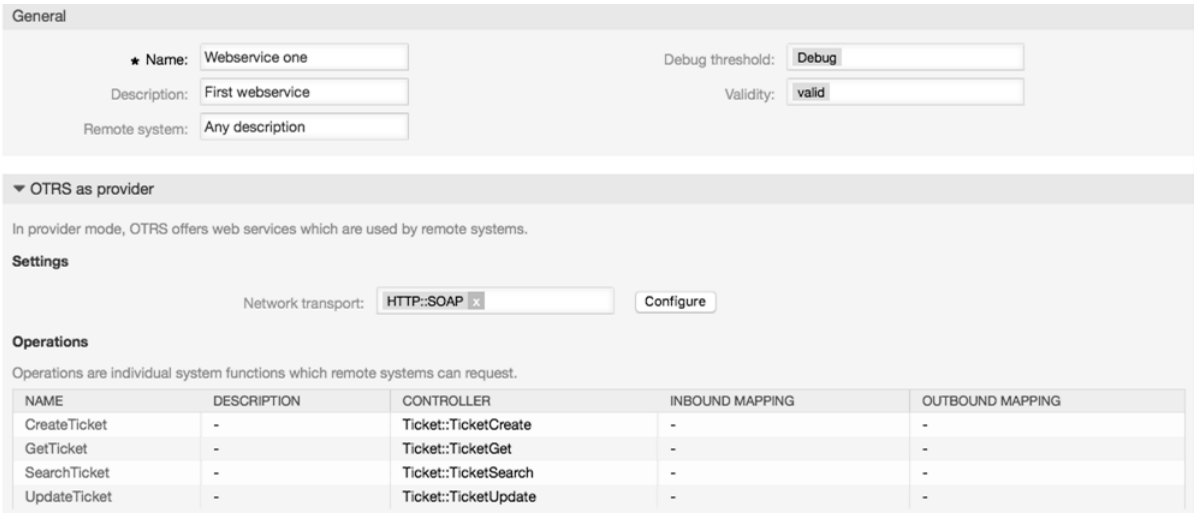

Figura 6. **Configuración Web Service OTRS**

Fuente: (OTRS AG, 2015, pág. 160)

La configuración del protocolo de transporte descrita en la *Figura 7*, asigna un espacio de nombres para reducir ambigüedades en los elementos del documento XML:

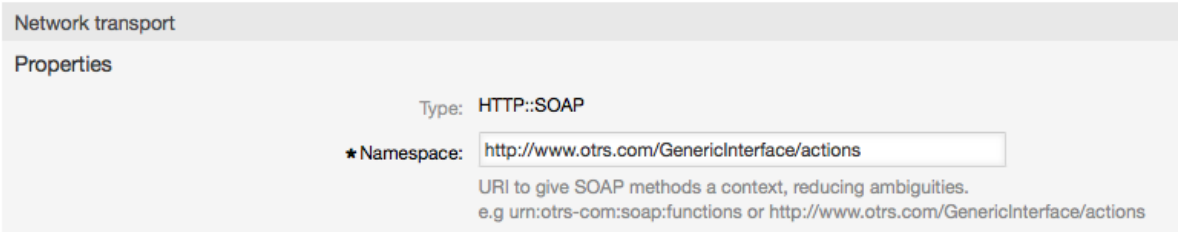

Figura 7. **Espacio de nombre Web Service OTRS**

Fuente: (OTRS AG, 2015, pág. 168)

El cuerpo del mensaje SOAP, debe cumplir el formato:

<UserLogin></UserLogin>
<Password></Password>
<Ticket>
<Title></Title>
<QueueID></QueueID>
<TypeID></TypeID>
<ServiceID></ServiceID>
<SLAID></SLAID>

```
        <StateID></StateID>
        <PriorityID></PriorityID>
        <CustomerUser></CustomerUser>
</Ticket>
<Article>
        <ArticleType>webrequest</ArticleType>
        <Subject></Subject>
        <Body></Body>
        <ContentType>text/plain; charset=utf8</ContentType>
</Article>
```

(OTRS AG, 2015, pág. 185)

2.6.6 La API de Nagios

Nagios es una plataforma que monitorea servicios mediante plugins desarrollados a medida (Schubert, 2008, pág. 23). Un plugin es cualquier ejecutable en el sistema operativo que retorne un código de estado (entero 0, 1 o 2), así, Nagios chequea el estado de los servicio en intervalos de tiempo regulares (Schubert, 2008, pág. 16).
Cuando se presenta una alerta, Nagios puede ejecutar una acción especifica, es decir, puede ejecutar un programa en el sistema operativo. La directiva event_handler indica el comando a ejecutar:

```
define service{
        host_name                       host
        service_description             servicio
        event_handler_enabled           1
        event_handler                   comando_a_ejecutar
}
```

(Galstad & Nagios Core, 2018).

Nagios envía datos sobre host y servicios mediante macros, que son referencias a una variable Nagios en la definición del comando:

```
define command{
        command_name        check_ping
        command_line        /usr/local/nagios/libexec/check_ping –H $HOSTADDRESS$
}
```

Aqui, la macro es $HOSTADDRESS$.

Antes que Nagios ejecute el comando, reemplazará la macro por el valor correspondiente al host y servicio.
También, puede pasar argumentos a los comandos. Los argumentos son especificados en la definición del servicio y son separados por !.

```
define service{
        host_name                       servidor
        service_description             ping
        check_command                   check_ping!200.0,80%!400.0,40%
}
```

La definición del comando sería:

```
define command{
        command_name        check_ping
        command_line        /usr/local/nagios/libexec/check_ping –H $HOSTADDRESS$ -w $ARG1$ -c
$ARG2$
```

} (Galstad & Nagios Core, 2018)

2.6.7 Definir los comandos y objetos Nagios

El monitoreo es ejecutado por Nagios quien permite extender la funcionalidad e integración mediante plugins (Khan & Khan, 2018, pág. 25). La funcionalidad a agregar es crear un ticket automático cuando se presente una alerta, entonces, se debe crear un comando que ejecute el plugin desarrollado.
La configuración de comandos es realizada en el archivo /usr/local/nagios/etc/objects/commands.cfg. Así, la definición del comando es:

```
define command{
        command_name        evt_crea_ticket_por_servicio
        command_line perl $USER1$/eventhandlers/ticket-otrs-nagios/src/crea_ticket_por_servicio '$SERVICESTATE$' '$SERVICESTATETYPE$' $SERVICEATTEMPT$ '$HOSTALIAS$' '$SERVICEDESC$' '$HOSTADDRESS$' '$SERVICEOUTPUT$' '$HOSTGROUPNAMES$' '$HOSTSTATE$' '$LASTSERVICESTATE$' '$SERVICEPROBLEMID$' '$LASTSERVICEPROBLEMID$'
}
```

Las macros son los parámetros del plugin, así, el plugin determinará si el ticket será creado o no.
Debe especificar en los servicios, cual comando será el manejador de eventos. Así, el servicio designado para crear ticket, debe definirse así:

```
define service {
   use                             service
   hostgroup_name        maquinas_recargas
   service_description   Atasco_billetero
   check_command         check_nrpe!chequar_atasco
   event_handler_enabled         1
   event_handler         evt_crea_ticket_por_servicio
}
```

Las directivas *event_handler_enabled* y *event_handler* definen el comando manejador de eventos.

2.7 Diseño del modelo de datos y el componente de software (Fases 3 y 4)

2.7.1 Conceptuar el modelo relacional del problema y traducirlo a modelo físico (SQL)

Al diseñar la base de datos, la normalización ayuda a encontrar un esquema de base de datos que pueda procesar eficientemente los cambios frecuentes (Koehler & Link, 2017, pág. 88). En esta fase, fue necesario describir el problema atraves de un modelo relacional normalizado, que elimina valores duplicados. La ausencia de datos redundantes es un criterio semantico deseable para que, un esquema de base de datos este bien diseñado (Kohler & Link, 2018, pág. 99).

Debido a que las aplicaciones a integrar poseen un esquema relacional, la tarea fue crear un esquema de base de datos que aproveche las entidades y relaciones existentes. Así, se considera que dos bases de datos son similares, si mantienen la distribución de relaciones entre tablas, es decir, entre los pares de claves primarias y foraneas. Como las llaves foraneas son enlaces forzados entre tablas, ellas representan datos invaluables para representar las relaciones entre datos (Buda, Cerqueus, Grava, & Murphy, 2017, pág. 84).
La *Figura 8* describe el modelo entidad-relación resultante:

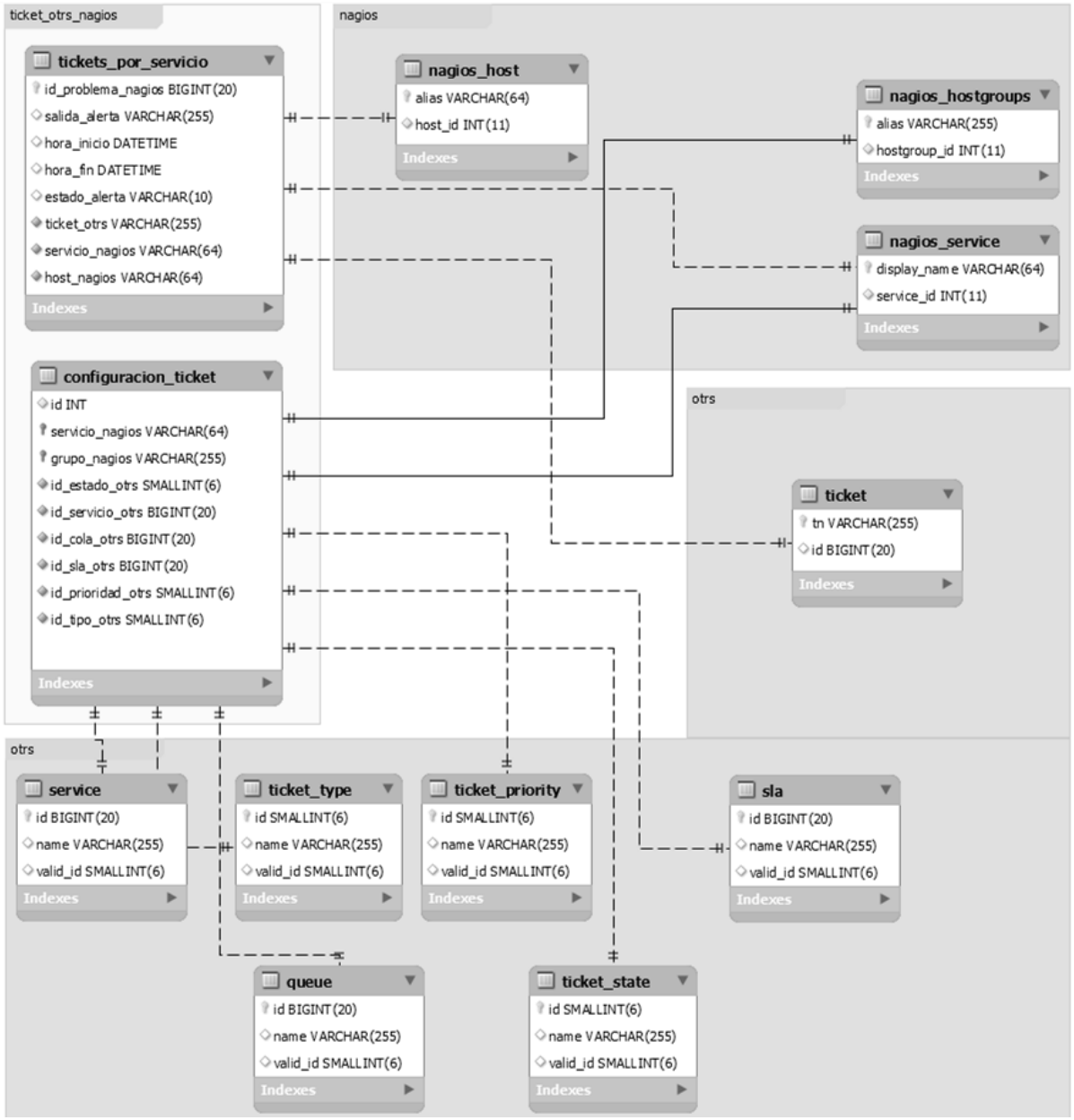

Figura 8. **Modelo relacional de la solución.**

Fuente: Elaboración propia.

El esquema *ticket_otrs_nagios* guarda la configuración de ticket por alerta y los registros de tickets generados automáticamente. Los esquemas *otrs* y *nagios* son las bases de datos existentes.

Para traducir el modelo conceptual a físico, es necesario usar el Lenguaje de Definición de Datos SQL. Las bases de datos deben satisfacer algunas reglas de integridad, así, la forma más básica de restricción es declarar algunos atributos como NOT NULL. En general, cada tabla de base de datos tiene un subconjunto de atributos declarados como clave primaria (Guagliardo & Libkin, 2018, pág. 5) que identifican a los registros de una tabla, por esa razón, los atributos que conforman dicha clave nunca pueden ser NULL.

La *Figura 9* muestra las sentencias versión MySQL para crear el esquema *ticket_otrs_nagios*:

```
CREATE SCHEMA IF NOT EXISTS ticket_otrs_nagios DEFAULT CHARACTER SET utf8;
USE ticket_otrs_nagios;
CREATE TABLE IF NOT EXISTS ticket_otrs_nagios.configuracion_ticket(
  id                INT NOT NULL,
  servicio_nagios   VARCHAR(64) NOT NULL,
  grupo_nagios      VARCHAR(255) NOT NULL,
  id_estado_otrs    SMALLINT(6) NOT NULL,
  id_servicio_otrs  BIGINT(20) NOT NULL,
  id_cola_otrs      BIGINT(20) NOT NULL,
  id_sla_otrs       BIGINT(20) NOT NULL,
  id_prioridad_otrs SMALLINT(6) NOT NULL,
  id_tipo_otrs      SMALLINT(6) NOT NULL,
  PRIMARY KEY (servicio_nagios, grupo_nagios)
) ENGINE = InnoDB;
CREATE TABLE IF NOT EXISTS ticket_otrs_nagios.tickets_por_servicio(
  id_problema_nagios    BIGINT(20) NULL,
  salida_alerta         VARCHAR(255) NULL,
  hora_inicio           DATETIME NULL,
  hora_fin              DATETIME NULL,
  estado_alerta         VARCHAR(10) NULL,
  ticket_otrs           VARCHAR(255) NOT NULL,
  servicio_nagios       VARCHAR(64) NOT NULL,
  host_nagios           VARCHAR(64) NOT NULL,
  PRIMARY KEY (id_problema_nagios)
) ENGINE = InnoDB;
```

***Figura 9*. SQL esquema ticket_otrs_nagios.**

Fuente: Elaboración propia.

2.7.2 Diseñar las clases que solucionan el problema (Modelo y Controlador del MVC)

Modelo-Vista-Controlador o MVC, es un patrón de arquitectura de software para implementar interfaces de usuario. MVC divide una aplicación de software en *tres partes* interconectadas: Modelo, Vista y Controlador (Lee & Wang, 2019, pág. 16).

El *Modelo* es el conjunto de clases identificadas en el dominio del problema, estas clases representan los objetos presentes en el problema.

La *Vista* es la interfaz de usuario, es decir, lo que el usuario observará en la pantalla, impresora u otro dispositivo de salida de la computadora.

El *Controlador* permite al usuario interactuar con la vista para usar el Modelo que representa y soluciona el problema. El *Controlador* es quien lleva el control del funcionamiento del programa, auxiliándose de la *Vista* que seguramente es una sofisticada interfaz gráfica y del *Modelo,* que representa y soluciona el problema (López Román, 2011, pág. 354).

Las *partes* y sus *relaciones* pueden ser representadas mediante UML, que es el lenguaje estandar para modelar software orientado a objetos (Pérez & Porres, 2019, pág. 152). Para representar objetos, UML se apoya en el diagrama de clases, que es el bloque de construcción básico de un sistema orientado a objetos (Bashir, Lee, Rehman Khan, Chang, & Farid, 2016, pág. 887). Así, se definió el diagrama de clases para el Modelo y Controlador, *Figura 10*:

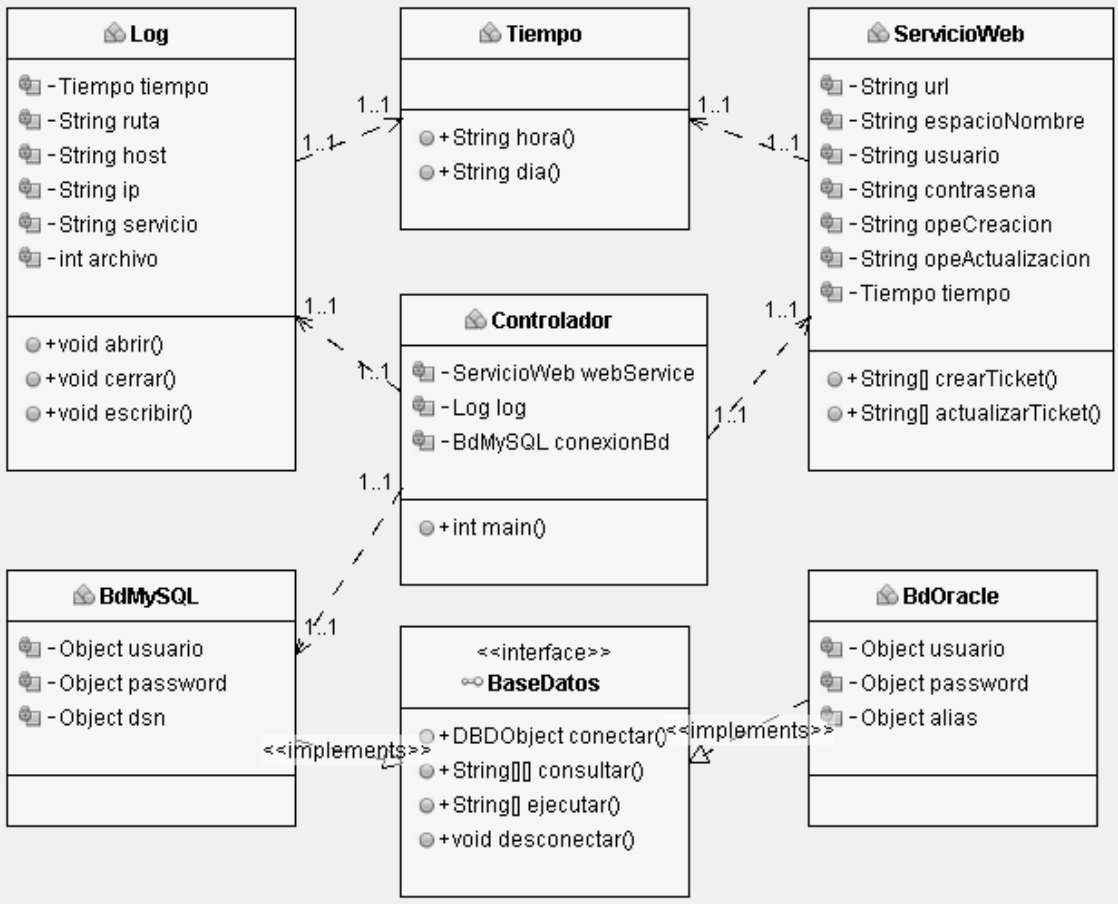

Figura 10. **Diagrama de clases Modelo y Controlador.**

Fuente: Elaboración propia.

Las clases que componen el Modelo *son usadas* por la clase Controlador, que contiene el algoritmo (*Figura 11)* que valida y crea el caso. Las clases fueron escritas en Perl, porque, se procesaron y combinaron datos de fuentes dispares (Andress & Linn, 2017, pág. 97); así, Nagios enviaba parámetros en tiempo de ejecución y los datos de configuración eran extraidos de MySQL y Oracle.

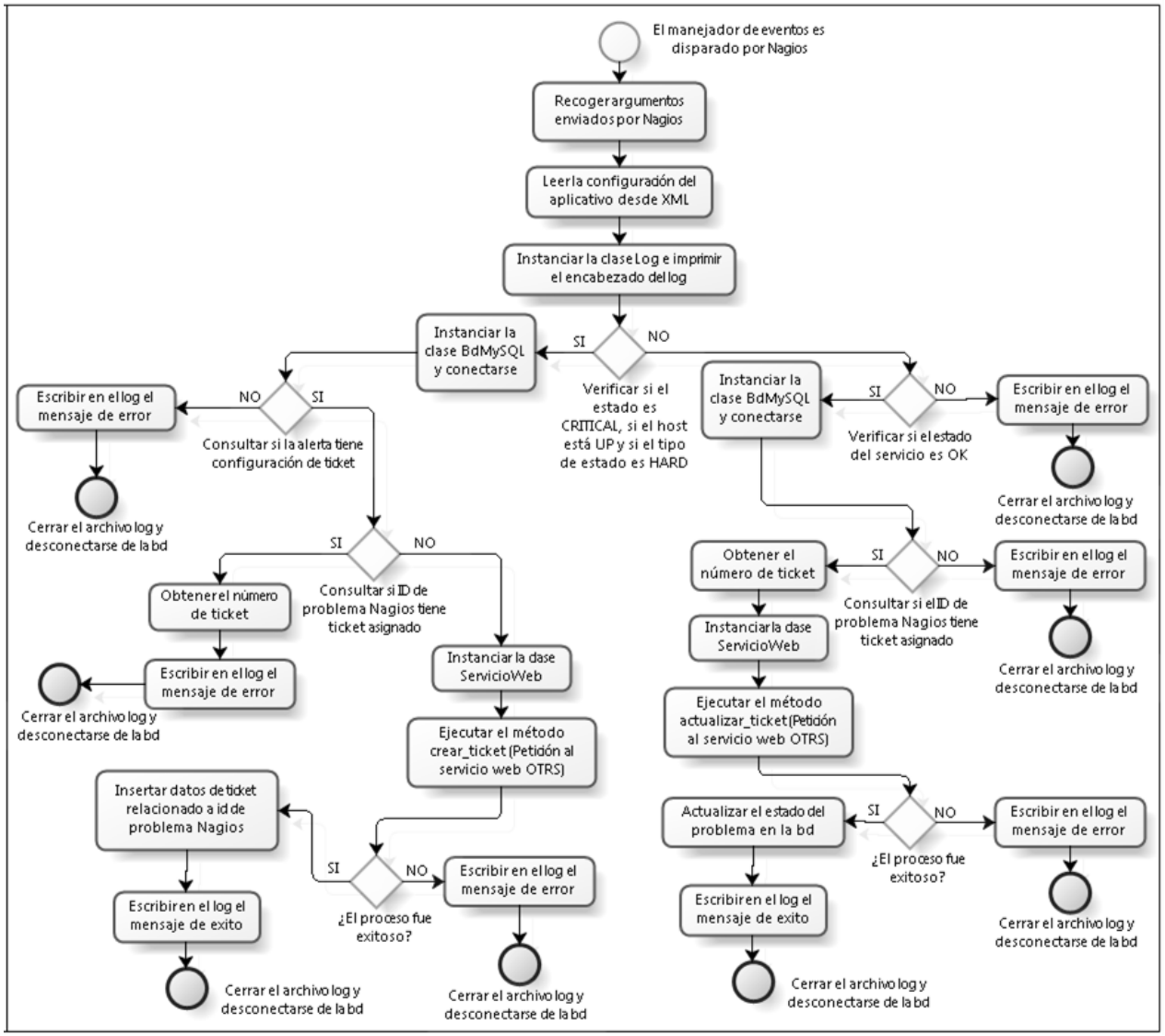

Figura 11. **Algoritmo del Controlador.**
Fuente: Elaboración propia.

2.7.3 Diseñar interfaz de administración (Vista)

El *Controlador* y la *Vista* son independientes, porque, estas pueden cambiarse, sin afectar la lógica del sistema o la manipulación de datos (Neven, Ross, Kim, & Bruce, 2018, pág. 5). Así, la conexión entre el *Controlador* y la *Vista* es realizada mediante los datos guardados en base de datos.

La *Vista* fue diseñada para configurar atributos del ticket a crear, de acuerdo a una alerta y grupo de host; esta tiene acceso a las tres bases de datos (otrs, nagios, ticket_otrs_nagios) y genera los formularios para gestionar la configuración ticket-alerta.

La *Figura 12* describe el diagrama de la *Vista*:

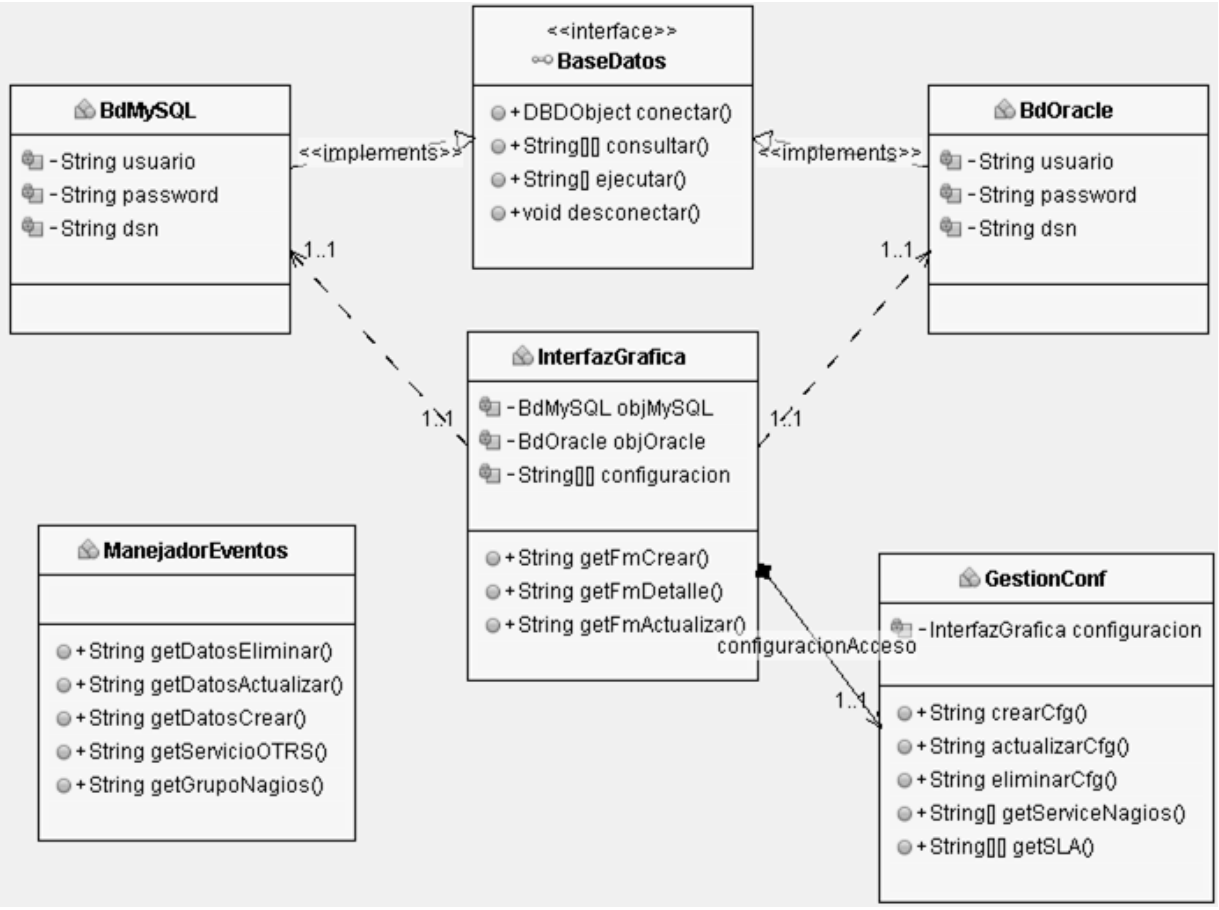

***Figura 12*. Diagrama de clase Vista.**

Fuente: Elaboración propia.

La *Vista* queda definida por una interfaz web compuesta de formularios, que intercambian datos con el servidor mediante AJAX.

AJAX reduce el tiempo de respuesta, la carga de servidor y el uso de ancho de banda de aplicaciones web (Kereshmeh & Charles, 2017, pág. 26). La Clase aislada "*ManejadorEventos*" es el código ejecutado del lado del cliente. La Clase InterfazGrafica es el código ejecutado del lado del servidor, así, genera los formularios e interactúa con las bases de datos.

3. RESULTADOS Y DISCUSIÓN

3.1. Flexibilidad de la solución propuesta

Usar el modelo relacional para almacenar datos y el patrón MVC para diseñar el software, permitió obtener una integración que puede aplicar a cualquier cantidad de dispositivos y servicios, porque la *Vista* habilita la creación y edición de n cantidad de relaciones *Alerta_Nagios-Ticket_OTRS*. Así, el formulario más importante permite crear y editar dicha relación, *Figura 13*:

Figura 13. **Formulario relación Alerta-Ticket.**
Fuente: Elaboración propia.

3.2. Cuantificando el problema y la solución

La solución propuesta fue implementada, así, se obtuvieron datos de 3 meses para analizar las variables de estudio. La *Figura 14* muestra cuantos tickets automáticos fueron generados por día:

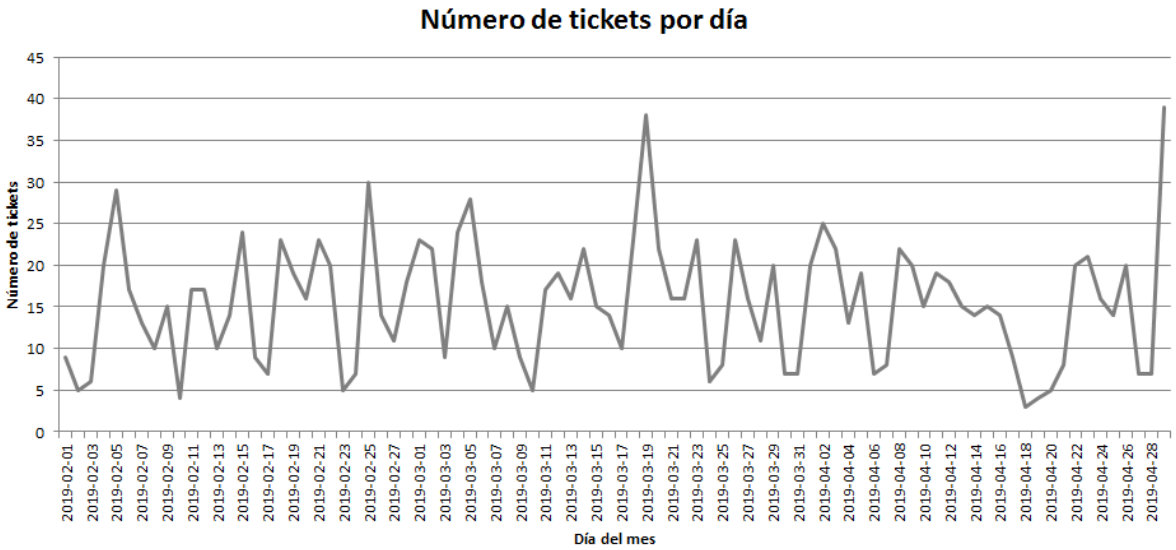

Figura 14. **Tickets por día.**
Fuente: Elaboración propia.

Después de tener una vista global, se analizó la cantidad de tickets por día semanal; la *Figura 15* muestra el acumulado por día semanal durante los 3 meses y determina los días críticos:

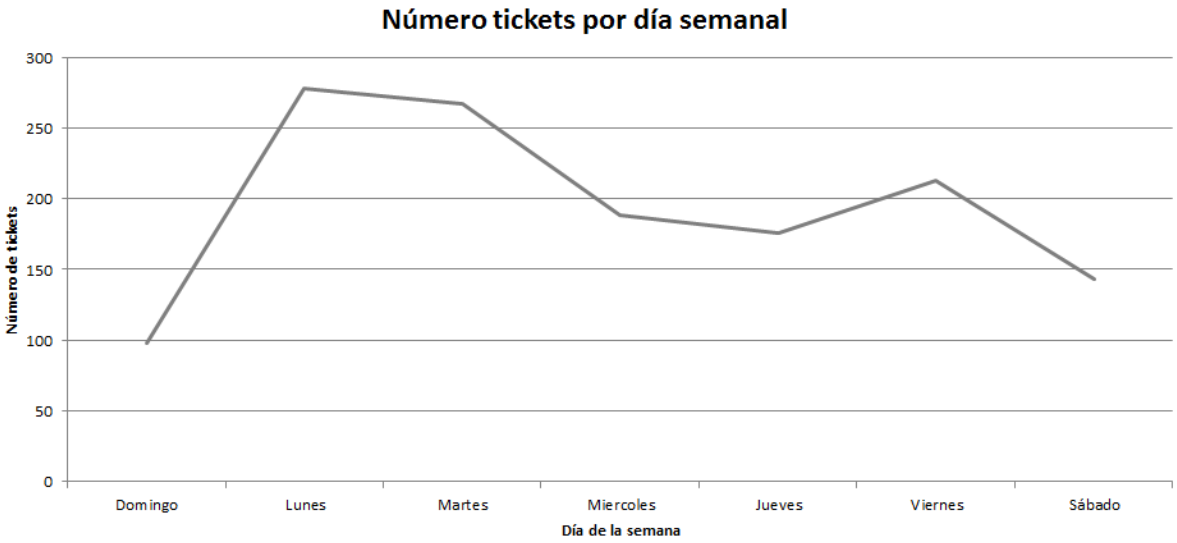

Figura 15. **Tickets por día semanal.**

Fuente: Elaboración propia.

El segundo enfoque fue el servicio, que fue determinado por las *Figuras 16, 17 y 18* (variable tres); así, se generaron gráficos por cada servicio:

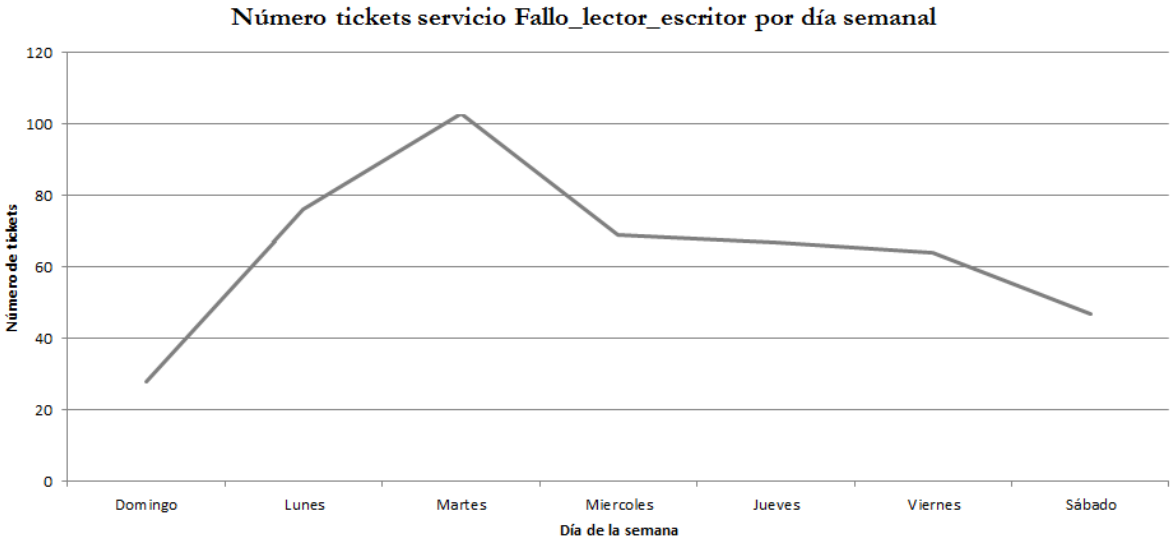

Figura 16. **Tickets servicio Fallo_lector_escritor por día semanal.**

Fuente: Elaboración propia.

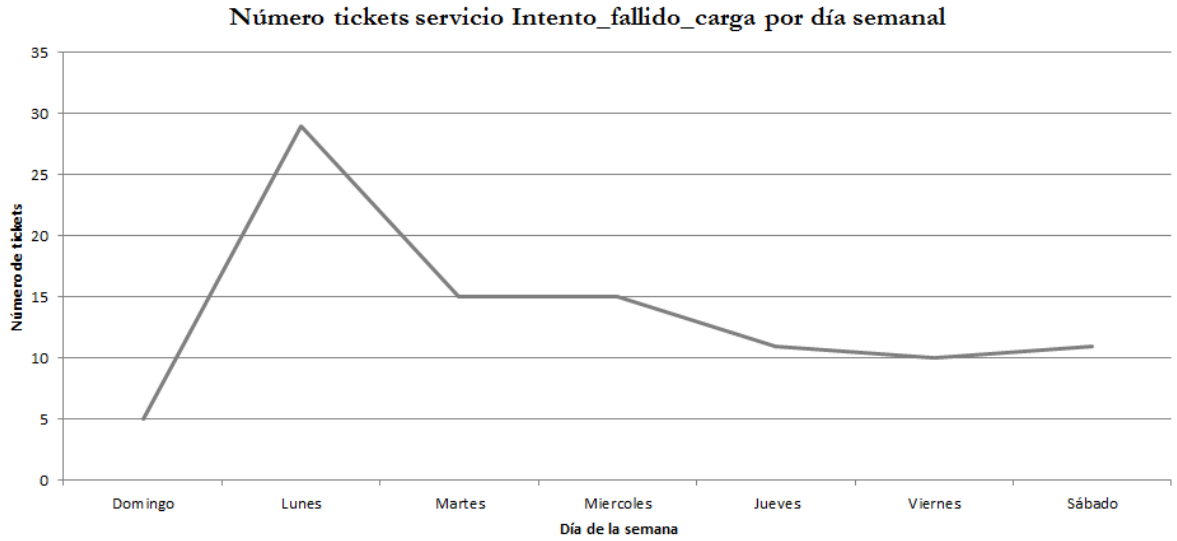

Figura 17. **Tickets servicio Intento_fallido_carga por día semanal.**
Fuente: Elaboración propia.

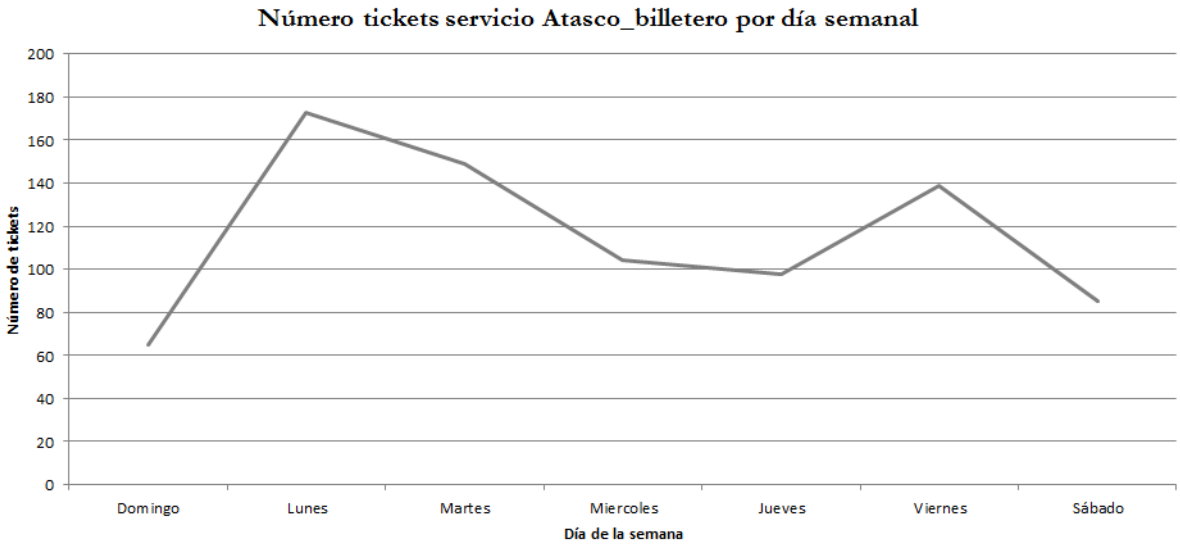

Figura 18. **Tickets servicio Atasco_billetero por día semanal.**
Fuente: Elaboración propia.

Las *Figuras 17 y 18* muestran los servicios críticos, destacando a *Atasco_billetero* por la cantidad de tickets. Estos resultados llevaron a enfocar el estudio en las horas críticas (variable cuatro), que fueron determinadas por la *Figura 19:*

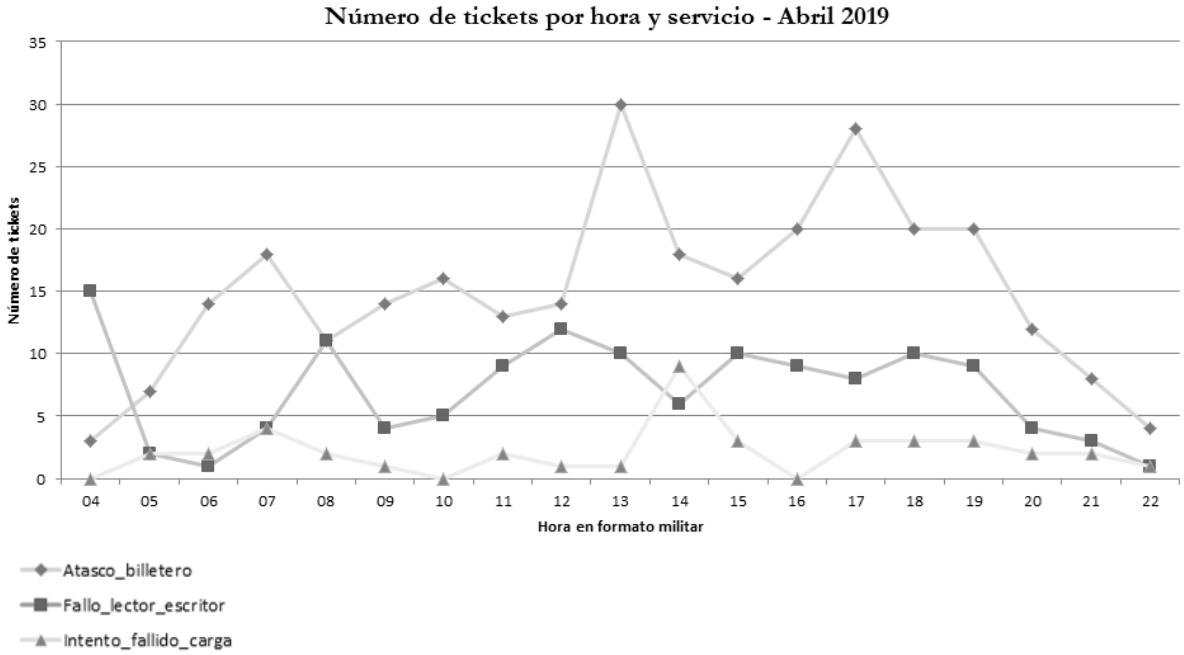

***Figura 19*. Tickets por hora y servicio Abril 2019.**

Fuente: Elaboración propia.

Los datos obtenidos por las variables uno, tres y cuatro permitieron ver puntos neurálgicos en cuanto a causa y horarios, adicionalmente, la variable dos representada por *la Figura 20*, mostró la distribución por zonas:

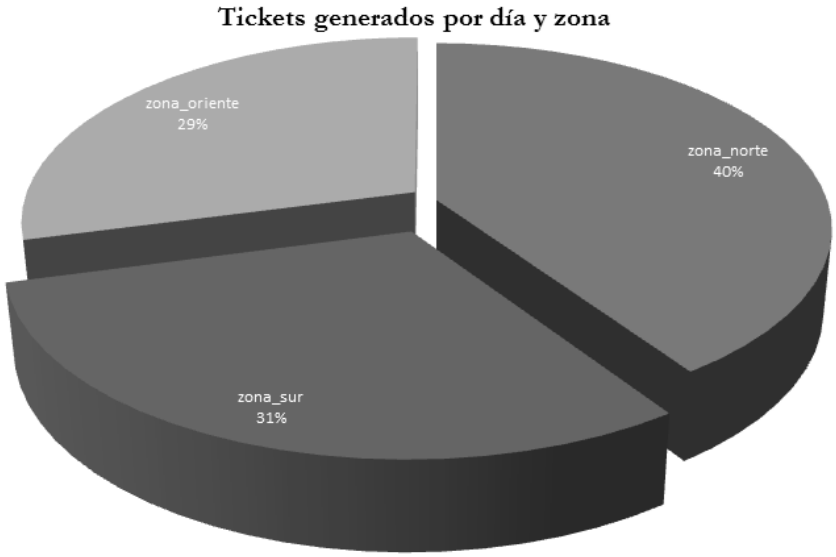

***Figura 20*. Tickets por zona.**

Fuente: Elaboración propia.

Los variables de estudio entregaron información a las áreas implicadas, quienes obtuvieron insumos para la toma de decisiones tanto correctivas como preventivas.

Al analizar la variable siete mediante la *Figura 21* (enfocada en el servicio Atasco_billetero), se evidenció una reducción en los tiempos de respuesta:

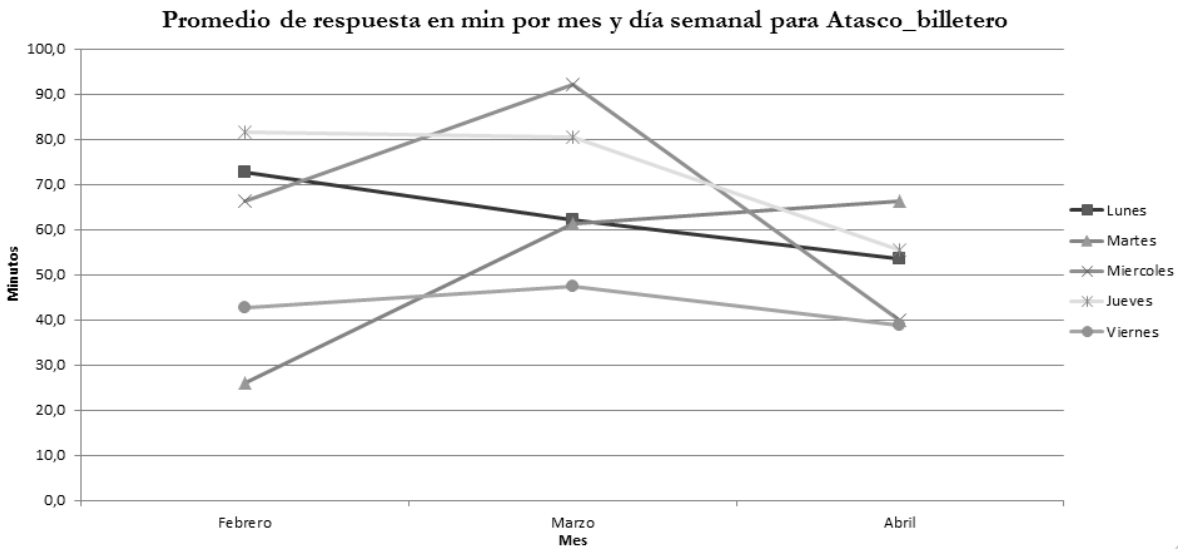

Figura 21. **Promedio respuesta en min por mes y día semanal para Atasco_billetero.**
Fuente: Elaboración propia.

3.3. Discusión

La hipótesis del problema "*la tarea manual para registrar alertas afecta los indicadores de servicio porque la tarea queda sujeta a omisiones humanas*" fue confirmada; la *Figura 21* muestra una disminución del promedio de respuesta, debido a que la integración creó tickets y notificó al personal en campo en tiempo real, así, se redujó el tiempo de notificación del incidente.

La integración reemplazó la tarea manual de crear tickets, cuantificó el problema y obtuvo datos útiles; los cuales quedaron como insumo para la toma de decisiones; estos resultados confirman la siguiente tendencia: *en las industrias automatizadas, las TI reemplazan las labores humanas. Así, las industrias informáticas son clasificadas por el uso de las TI para crear flujos eficientes y efectivos de información para la toma de decisiones* (Joshi, Bollen, Hassink, De Haes, & Van Grembergen, 2018, pág. 369).

4. CONCLUSIONES

La integración de plataformas de software no debe limitarse al simple intercambio de datos, sino que, previamente debe establecerse un modelo de datos que haga útil los datos generados por las transacciones.

El software producido por una integración de plataformas debe ser escalable, esta característica es fundamental para que la solución sea mantenible y responda a nuevas necesidades operativas; esto se logra mediante interfaces de administración y modelos de datos que unifiquen (sin redundar) las diversas fuentes de información.

La integración de plataformas contribuye a la eficiencia y medición de procesos operacionales. ¿Qué pasaría si Nagios no solo se limita a crear el ticket, sino, a realizar una intervención correctiva? El proceso sería más eficiente y las TI estarían reemplazando la tarea humana en todo el proceso.

5. REFERENCIAS

Andress, J., & Linn, R. (2017). Chapter - Introduction a Perl. En *Coding for Penetration Testers Second Edition* (págs. 81-110).

Bashir, R. S., Lee, S. P., Rehman Khan, S. U., Chang, V., & Farid, S. (2016). UML models consistency management: Guidelines for software quality manager. *International Journal of Information Management*, 883-899.

Baskaran, K., Khalid Rai, S., & Arumuga, V. (2017). IMPLEMENTATION OF EFFECTIVE AND LOW-COST BUILDING MONITORING SYSTEM(BMS) USING RASPBERRY PI. *Energy Procedia*, 179-185.

Buda, T. S., Cerqueus, T., Grava, C., & Murphy, J. (2017). Rex: Representative extrapolating relational databases. *Elsevier, 67*, 83-99.

Eito-Brun, R. (2017). Chapter 1 - XML: The Basis of the Language. En *XML-based Content Management* (págs. 1-30).

Faircloth, J. (2016). Chapter 5 - Web applications and services. En *Penetration Tester's Open Source Toolkit (Fourth Edition)* (págs. 179-214).

Foster, J. (2002). *Developing web services with Java APIs for XML using WSDP*. Rockland, Mass: Syngress Media.

Galstad, E., & Nagios Core, D. (2018). *Nagios Core Documentation.* Obtenido de https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/objectdefinitions.html

Goralski, W. (2017). Chapter 1 - Protocols and Layers. En W. Goralski, *The Illustrated Network* (Second Edition ed., págs. 3-46). Burlington, Massachusetts, Estados Unidos: Elsevier.

Guagliardo, P., & Libkin, L. (2018). On the Codd semantics of SQL nulls. *Elsevier*, 1-14.

Gurugé, A. (2004). *Web service: theory and practice.* Burlington, MA, US: Digital Press.

Joshi, A., Bollen, L., Hassink, H., De Haes, S., & Van Grembergen, W. (2018). Explaining IT governance disclosure through the constructs of IT governance maturity and IT strategic role. *Information & Management*, 368-380.

Kereshmeh, A., & Charles, E. (2017). JavaScript Object Notation (JSON) data serialization for IFC schema in web-based BIM data exchange. *Automation in Construction*, 24-51.

Khan, R., & Khan, S. U. (2018). Design and implementation of an automated network monitoring and reporting back system. *Journal of Industrial Information Integration*, 24-34.

Koehler, H., & Link, S. (2017). Inclusion dependencies and their interaction with functional dependencies in SQL. *Journal of Computer and System Sciences*, 104-131.

Kohler, H., & Link, S. (2018). SQL schema design: foundations, normal forms, and normalization. *Elsevier*, 88-113.

Lee, H.-Y., & Wang, N.-J. (2019). Cloud-based enterprise resource planning with elastic model–view–controller architecture for Internet realization. *Computer Standars & Interfaces, 64*, 11-23.

Libkin, L. (2016). Certain Answer as Objects and Knowledge. *Artificial Intelligence*, 1-19.

López Román, L. (2011). *Programación estructurada y orientada a objetos.* Alfaomega Grupo Editor.

Martinez, C. (2012). *Estadística y Muestreo* (Décima tercera ed.). Bogotá: ECOE ediciones Ltda.

Neven, E., Ross, S., Kim, M., & Bruce, T. (2018). Context-aware design pattern for situated analytics Blended Model View Controller. *Journal of Visual Languages and Computing*, 1-12.

OTRS AG. (26 de 05 de 2015). *OTRS 5 - Admin Manual.* Obtenido de https://ftp.otrs.org/pub/otrs/doc/doc-admin/5.0/en/pdf/otrs_admin_book.pdf

Pereira, R., Dupont, I., Carvalho, P., & Jucá, S. (2018). IoT embedded linux system based on Raspberry Pi applied to real-time cloud monitoring of a decentralized photovoltaic plant. *Measurement*, 286-297.

Pérez, B., & Porres, I. (2019). Reasoning about UML/OCL class diagrams using constraint logic programming and formula. *Information Systems*, 152-177.

Putra, L., & Kanigoro, B. (2015). Design and Implementation of Web Based Home Electrical Appliance Monitoring, Diagnosing, and Controlling System. *Procedia Computer Science*, 34-44.

Refsnes Data. (2018). *XML Tutorial.* Obtenido de https://www.w3schools.com/xml/

Schubert, M. (2008). *Nagios 3 enterprise network monitoring: including plug-ins and hardware devices.* Burlington: Syngress Pub.

Shichkina, Y. (2019). Approaches to speed up data processing in relational databases. *Procedia Computer Science*, 131-139.

Smiatek, G. (2005). SOAP-based web services in GIS/RDBMS environment. *Environmetal Modelling & Software, 20*(6), 775-782.

Sterling, T., Anderson, M., & Brodowicz, M. (2018). Chapter 11 - Operating Systems. En *High Performance Computing* (págs. 347-362).