Implementación de Middleware Publicador/Subscriptor para Aplicaciones Web de Monitoreo

Defossé N., López R. A., Marcelo E. Gómez, Konstantinoff P., Wahler S., Castro L., Harris G.
Departamento de Informática Trelew, Facultad de Ingeniería, Universidad Nacional de la Patagonia
Mitre 655 – CP (9100) Trelew Tel.: (+54280) 4428402
{nahuel.defosse, lopez.ricardoa, gomezmarceloe, pkonstantinoff, sebastian.wahler} @gmail.com

RESUMEN

En la actualidad, millones de clientes se conectan a la nube utilizando el protocolo HTTP (Protocolo de Transferencia de Hipertexto). Históricamente, el estudio de los sistemas distribuidos ha propuesto diversas técnicas de optimización de acceso; como ser Sun RPC, CORBA, SOAP. Sin embargo, en el ámbito de las aplicaciones web una tendencia muy generalizada es la abstención a este tipo de *middlewares,* en favor de principios arquitectónicos propuestos como REST (*Representational State Transfer*) [1].

Uno de los problemas que presenta esta mecánica de comunicación, es la imposibilidad que un cliente reciba actualizaciones de un recurso remoto sin iniciar un requerimiento. Una técnica actual para mitigar el problema consiste en la utilización de WebSockets.

Paralelamente, las bases de datos relacionales han ganado capacidades de notificación a través de canales asincrónicos. Estas pueden ser aprovechadas para recuperar cambios en los datos de tiempo real.

En este trabajo se presenta el desarrollo de una [illegible] actualizaciones de estado en tiempo real. Inicialmente se plantea su diseño exponiendo recursos bajo REST, para luego abordar un enfoque Publicador Subscriptor sobre los mismos recursos utilizando *middleware* basado en WebSockets.

Palabras Clave: Websocket, REST, MQTT, Real Time Web

CONTEXTO

El presente trabajo se encuentra dentro de las líneas de investigación del proyecto "Computadoras Industriales con control WEB", del Departamento de Informática de la Facultad de Ingeniería de la Universidad Nacional de la Patagonia San Juan Bosco.

1. INTRODUCCIÓN

Una API REST se conforma de un conjunto de URIs (Uniform Resource Identifier) , las cuales dependiendo del método HTTP que el cliente utilice tienen una semántica diferente. Un ejemplo de definición de métodos HTTP ante URIs puede apreciarse en Tabla 1.

Por ejemplo, si el sistema en el servidor se tratase de un RDBMS (*Relational DataBase Management System*), una petición GET podría asociarse a una instrucción SQL `SELECT`, con POST a un `INSERT`, con `PUT` a un UPDATE y `DELETE` a su homónimo SQL.

Ciertos encabezados HTTP cobran relevancia como el caso de `Content-Type`, definiendo la codificación o representación del estado del [illegible] siendo popular la utilización de Javascript Object Notation o JSON por su versatilidad de operación del lado del cliente.

URI	GET	PUT	POST	DELETE
Colección, como `/resources`	**Listar** las URIs de colección.	**Reemplazar** toda colección.	**Crear** un nuevo eleme La nueva URI se asi automáticamente.	**Borrar** la colección entera
Elemento, co `/resources/item`	**Recuperar** representación del miem de la colección, expres en el tipo de med apropiado (`Conten Type`)	**Reemplazar** el element actualizar. Crearlo si existe.	No utilizado generalme por ya incluir identificaci Se utiliza para la creación la identificación colección.	**Borrar** el elemento referenci en la colección.

Tabla 1. Descripción de métodos HTTP bajo REST.

En nuestro estudio se desarrolló una aplicación web de monitoreo de estaciones transformadoras eléctricas. La cual está conformada por una base de datos, un módulo de recolección de estados y eventos y un módulo de visualización.

El módulo de visualización está compuesto por gráficos vectoriales redimensionables (SVG) encargados de mostrar el estado del campo en tiempo real al operador. Estos gráficos son almacenados en una tabla de la base de datos llamada `PSVG` (por Pantalla SVG).

Cada uno de los archivos SVG está constituido por varios elementos o nodos -que representa algún estado de la estación-. Estos elementos poseen un identificador (TAG) que los relaciona con una fila de una tabla llamada `EG` (Elemento Gráfico). En cada columna de esta tabla se almacenan el estado de los atributos relleno (*`fill`*), contorno (*`stroke`*), texto (*`text`*) y un *timestamp* que indica la marca temporal de la última actualización.

La tabla `EG` es actualizada periódicamente por un proceso de recolección de estados y eventos.

La aplicación web descarga un documento HTML con su correspondiente código JavaScript. Este accede a las tablas previamente mencionadas como recursos REST. El diagrama de secuencias de esta operación puede observarse en la Figura 1. En función de la pantalla (o diagrama SVG) seleccionado, se realiza una petición asincrónica GET a `/api/v1/eg/?psvg_id=<identificador_pantalla>`, que recupera el estado inicial de la estación y lo aplica a cada nodo del SVG haciendo *queries* al DOM, utilizando el TAG y modificando atributos (*`fill`*, *`stroke`* y *`text`*). Durante este proceso guarda el *timestamp* mayor.

Luego cada 5 segundos se repite la petición, pero agregando como parámetro a la misma, el último *timestamp* recuperado, para filtrar solo las novedades y aplicando los cambios que sean recuperados a los nodos SVG referenciados.

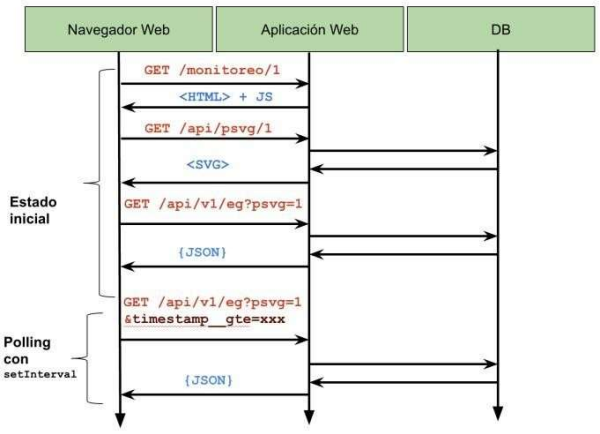

Figura 1: Secuencia de recuperación de estado.

Dado que la carga de 160 elementos iniciales rondaba los 240Kb y su actualización era de 16Kb por petición, detectamos un *overhead* innecesario en el módulo de visualización y consecuentemente en la base de datos. Sumado a esto, el uso de peticiones cronometradas genera desfases y retrasos del estado real.

Para solucionar estos problemas abordamos la detección de cambios dentro del dominio de la base de datos. Para esto, PostgreSQL provee las instrucciones `NOTIFY` y `LISTEN` [6], estas permiten publicar y recibir mensajes a través de un canal. Basándonos en estas funcionalidades, se desarrolló una utilidad de línea de comandos llamada *`pg_notfications`*, que agrega un *stored procedure* (SP) y una serie de *triggers* ante los eventos `INSERT` o `UPDATE` de las tablas que se indiquen como argumento. Para la publicación de datos, el RDBMS provee la información acerca de cuál tabla y fila activaron el *trigger* que invoca el SP encargado de convertir la fila afectada a formato JSON y publicar con `NOTIFY al canal.`

Para la recepción de datos del canal se desarrollaron una serie de *callbacks*, entre los cuales se encuentran el *logging* por salida estándar o invocación de un comando.

Para llevar estas actualizaciones al módulo de visualización se requiere una vía hacia el navegador web dónde el cliente no deba realizar peticiones cada cierto tiempo. Para esto se investigó el protocolo WebSocket, el cual consiste una comunicación bidireccional entre un servidor web y un cliente. A diferencia de HTTP, dónde se transporta texto, WebSocket está orientado a tramas. El tipo de tramas a transmitir se negocia al establecimiento de la conexión. El formato de estas tramas se conoce como sub- protocolos de WebSocket y se encuentran establecidos por la Autoridad de Nombres de Internet (o IANA por su siglas en inglés).

Message Queuing Telemetry Protocolo MQTT[3] fue el sub-protocolo seleccionado para el envío de notificaciones de nuestra aplicación web.

MQTT es un *middleware* orientado a mensajes (MoM)[2] basado en el paradigma Publicador Subscriptor [3]. Se compone de un broker central al cual se conectan los cliente.

Al conectarse un cliente, si éste desea recibir publicaciones bajo un tópico, lo hace saber al broker mediante un mensaje *Subscribe*. Una vez que el *broker* acepta la suscripción, las publicaciones que se correspondan con el tópico serán enviadas al cliente suscrito.

Finalmente, se vincularon los mensajes recuperados por la utilidad `pg_notifications` y se los envió al broker como publicaciones.

Del lado del código HTML, se procedió a adaptar el código Javascript -que utilizaba un *timer* para disparar la llamada AJAX-, sustituyéndolo por la utilización de una biblioteca para realizar comunicaciones MQTT.

Una característica de los mensajes *Publish*, es que aceptan una bandera llamada *retain*, que le indica al *broker* que debe guardar el mensaje. Cualquier cliente que se suscriba posteriormente al tópico de un mensaje retenido, recibirá éste último. La utilización de este flag permitió completar en la subscripción, gran parte del estado inicial de la estación, pero se debió agregar además un proceso que publique el estado de los elementos que no hubieren cambiado.

Finalmente, las llamadas a `/api/v1/svgelement/` se reemplazaron por suscripciones al tópico `/api/v1/EETT/+`, donde EETT es la `PSVG` y el + representa el comodín mono-nivel.

El Topic Mapper se encarga de publicar las actualizaciones de `fill`, `stroke` y `text` de los EG con el tópico correspondiente.

En la figura 3 pueden observarse los componentes del sistema y su interacción. La línea punteada identifica el flujo de los eventos asincrónicos que son enviados al cliente, atravesando sus diferentes fases: detección en el trigger, publicación en el canal de la base de datos, recolección por el Topic Mapper, asignación de tópico y publicación al broker

MQTT y por último, recepción por parte de los clientes suscritos.

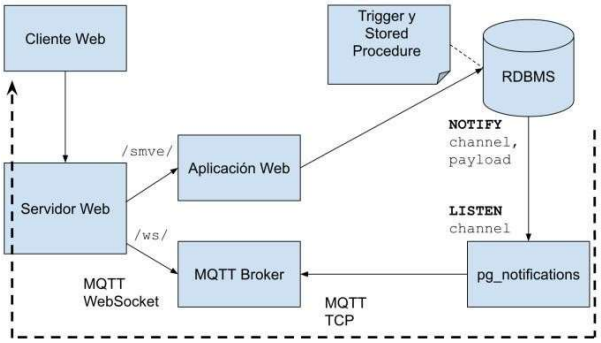

Figura 2: Aplicación adaptada a MQTT-WC.

2. LÍNEAS DE INVESTIGACIÓN, DESARROLLO E INNOVACIÓN

El estudio de aplicaciones para monitoreo de centrales eléctricas involucra una serie de elementos, cada uno de ellos con diferentes desafíos específicos. Actualmente contamos con las siguientes líneas de investigación dentro del grupo.

- Sistemas SCADA [7].
- Protocolos de comunicación industrial.
- Actualizaciones asincrónicas con MQTT-WS.
- Microcontroladores DSP y aplicaciones WEB.
- Protocolos IoT.

3. RESULTADOS OBTENIDOS

En este trabajo se muestra una mejora en el desempeño y la experiencia de usuario de una aplicación de monitoreo de centrales eléctricas, puntualmente un proceso de *polling* de 5 segundos pasó a requerir fracciones de segundo para observar cambios en un diagrama SVG. La eliminación del *polling* redujo la cantidad de recursos requeridos por la aplicación web en estado de reposo. Además hemos aprovechado características reactivas del RDBMS PostgreSQL, haciendo innecesarias las consultas basadas en ventana de *timestamp*.

Una captura de pantalla del sistema en funcionamiento puede observarse en la Figura 3.

4. FORMACIÓN DE RECURSOS HUMANOS

El equipo de trabajo está formado por docentes y alumnos del grupo de investigación relacionado con el proyecto enunciado, dirigido por el Mg. Ing. Ricardo López y el Lic. Marcelo Gómez.

El Lic. Nahuel Defossé se encuentra desarrollando su trabajo de tesis de Maestría dirigido por el Dr. Fernando Tinetti y co-dirigido por el Mg. Ing. Ricardo López. Los alumnos Lucas Luis Castro y Gonzalo Harris realizaron su tesina de licenciatura implementando actualizaciones asincrónicas con MQTT-WS.

El alumno Pedro Konstantinoff se encuentra desarrollando su tesina de grado en relación a temáticas de IoT, reutilizando estos resultados.

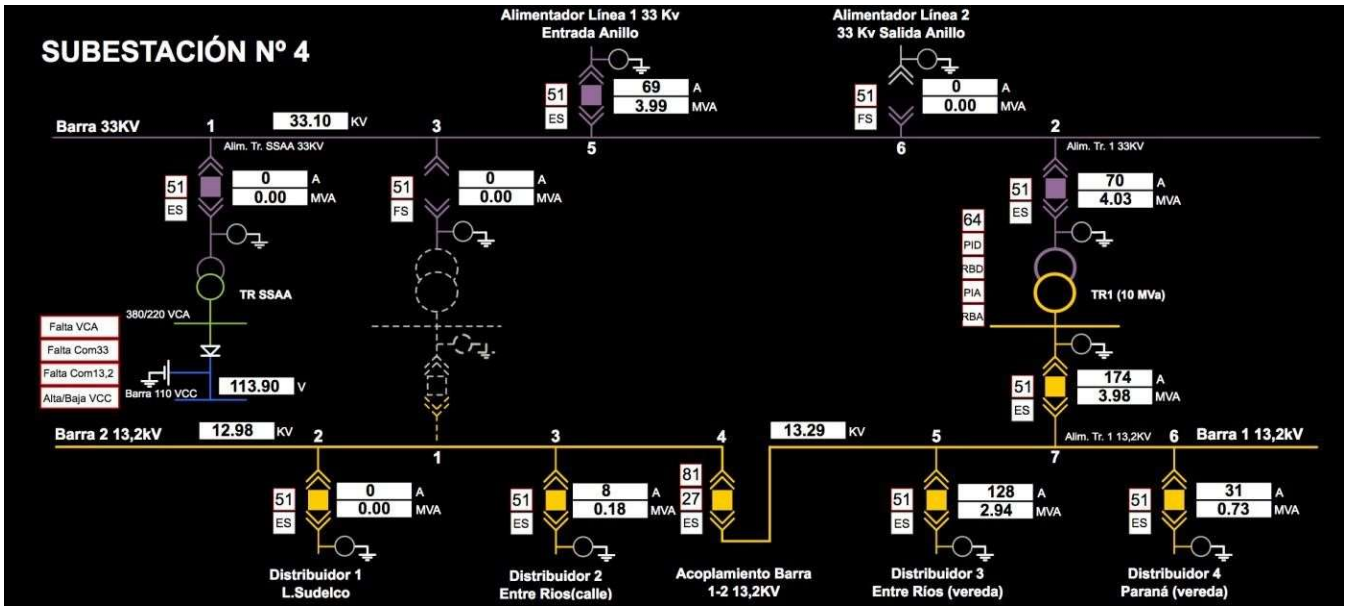

Figura 3: Pantalla SVG en funcionamiento

5. BIBLIOGRAFÍA

[1] "Fielding Dissertation: CHAPTER 5: Representational State Transfer ...." https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Fecha de acceso 25 feb.. 2017.

[2] Coulouris, George F. Distributed Systems: Concepts and Design. Boston, MA: Addison-Wesley, 2011.

[3] Especificación MQTT de OASIS. Sitio Web: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html (fecha de acceso)

[4] Architectural impact of the SVG-based graphical components in web applications. Moreno, Olivera. 2008.

[5] Real Time Water Supply System Hydraulic and Quality Modeling – A Case Study. Sadok, Gomes , Eisenhauer , Kelner.

[6] Listen/Notify en PostgreSQL. Sitio Web: https://www.postgresql.org/docs/9.1/static/sql-notify.html (fecha de acceso)

[7] Defossé, N. , van Haaster, D. M. , Pecile, L., Tinetti, F. G. , & Lopez, R. A. (2014). Implementación de Sistemas SCADA Utilizando Lenguajes de Alto Nivel. *WICC 2014 XVI Workshop de Investigadores en Ciencias de la Computación* (Vol. XVI, pp. 779–784). Ushuaia. [8]López, R. A., Pinciroli, E., & Tinetti, F. G. (2014). Microcontroladores asociados a medición y comunicaciones en sistemas SCADA de energía. *Workshop de Investigadores En Ciencias de La Computación*, *XVI*, 767–773. [9] Gómez, M. E., Wahler, S. P., Tinetti, F. G., & López, R. A. (2014). Implementación de mensajes rápidos y valores de muestreo IEC61850 sobre Ethernet con microcontroladores. *Workshop de Investigadores En Ciencias de La Computación*, 774–778.