UNIVERSIDAD NACIONAL DE EDUCACION A DISTANCIA

UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO FIN DE MASTER

MASTER EN INGENIERÍA DE SISTEMAS Y CONTROL

Incorporación de métodos de muestreo periódicos basados en eventos en el desarrollo de Laboratorios Remotos para Control Automático

Autor: Jorge Hernández Medina

Directores: Luis de la Torre Cubillo

Jesús Chacón Sombría

Curso académico 2019/2020

Convocatoria de Septiembre

UNIVERSIDAD NACIONAL DE EDUCACION A DISTANCIA

UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO FIN DE MASTER

MASTER EN INGENIERÍA DE SISTEMAS Y CONTROL

Incorporación de métodos de muestreo periódicos basados en eventos en el desarrollo de Laboratorios Remotos para Control Automático

Autor: Jorge Hernández Medina

Directores: Luis de la Torre Cubillo

Jesús Chacón Sombría

Proyecto Tipo A

Autorización

Autorizamos a la Universidad Complutense y a la UNED a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado:

Jorge Hernández Medina

RESUMEN

Históricamente los sistemas de control automático han empleado principalmente la técnica del muestreo periódico de señales, debido especialmente a la gran base teórica que existe al respecto de los mismos. Sin embargo, en las últimas décadas los sistemas basados en la aparición de eventos para la toma de decisiones están experimentando un gran auge, debido a que sus características los hacen muy interesantes para los sistemas controlados en red.

Teniendo en cuenta lo anterior resulta claro que los sistemas basados en eventos constituyen un campo muy importante en la actualidad, de lo que se deriva lo interesante de proporcionar una formación al respecto de los mismos a todas aquellas personas que, o bien trabajan en el desarrollo de sistemas de control, o bien están desarrollando sus estudios en el campo de la automatización y control.

Este Trabajo de Fin de Master tiene como objetivo el proporcionar las herramientas que permitan la incorporación de estos métodos de control basado en eventos dentro del marco de desarrollo de Laboratorios Remotos para enseñanza de Control Automático que emplea la UNED, de forma que los alumnos puedan familiarizarse con este tipo de sistemas mediante la práctica con ellos.

Para llevar a cabo esta tarea nos hemos apoyado en un desarrollo prexistente de un servidor que emplea un protocolo de comunicaciones desarrollado por la UNED. Partiendo de este punto hemos desarrollado toda la parte de detección de eventos, así como la recuperación de eventos perdidos en caso de desconexión. También se ha implementado la programación necesaria para que se puedan conectar varios usuarios a la vez al mismo laboratorio.

Palabras clave: sistemas basados en eventos, sistemas controlados en red, laboratorios remotos, protocolo de comunicaciones

ABSTRACT

Historically, automatic control systems have mainly employed periodic signals sampling techniques, due to the great amount of theorical basis existing about them. Nevertheless, based on events systems have experimented a big growth in the last decades, because their characteristics make them an engaging option for Networked Control Systems.

Event-Based Systems constitute a very important field nowadays. Therefore, it is interesting to provide, not only Automation and Control students but also control systems developers with knowledge and training in this area.

The aim of this master's project is to provide the tools to incorporate these event-based control methods into the framework of Remote Laboratories for the teaching of Automatic Control developed by the UNED, so that students can practice with this kind of systems and familiarize with them.

To carry out this task we have based our work on a preexisting development of a server that employs a communication protocol developed by UNED. From that starting point, we have developed all the event detection part, as well as the lost events recovering part in cases of disconnection. We have also implemented all the necessary programming to allow multiple users to access the same laboratory at the same time.

Key Words: based on events systems, Networked Control Systems, remote laboratories, communications protocol

ÍNDICE

ÍNDICE DE FIGURAS

1.- PREÁMBULO

1.1.- INTRODUCCIÓN

Si bien los sistemas de control automático se llevan empleando en la industria desde hace ya varias décadas, actualmente estamos asistiendo a un auge en la implementación de los mismos. Esto es debido a múltiples factores, entre los que caben destacar el ahorro que suponen en los costes de producción, así como las mejoras experimentadas tanto en sensores y actuadores como en posibilidades de comunicación. Este último aspecto unido al gran desarrollo que han experimentado las redes de comunicación es muy interesante, ya que permite controlar dichos sistemas desde lugares muy alejados de los mismos.

Tradicionalmente los sistemas de control automático se han basado mayoritariamente en el muestreo periódico de las señales del sistema a controlar para, en base a ellas, decidir qué acciones se han de tomar. Se trata de sistemas con un comportamiento muy sólido y que han sido estudiados en profundidad, contando con una gran base teórica y literatura al respecto [1] (Åström y Wittenmark, 1997), siendo este uno de los principales motivos de su uso mayoritario.

Sin embargo, este tipo de sistemas, basados en intervalos periódicos de tiempo, presentan algunos inconvenientes a tener en cuenta. En primer lugar, el uso extensivo de la CPU, la cual se ve muy condicionada por tener que comprobar de forma periódica el estado de las entradas, aun cuando puede no ser necesario si no se ha producido un cambio significativo. Lo anterior supone que la CPU tenga que desatender otras tareas para atender la tarea periódica, así como un aumento del consumo de energía.

Otro aspecto a tener en cuenta es la gran cantidad de flujo de información que se genera, con el consiguiente consumo de ancho de banda. Se trata de una característica muy a tener en cuenta cuando se emplean dentro de sistemas distribuidos en los que el medio es compartido y se hace necesario el aprovechamiento del mismo.

Es importante destacar el empleo cada vez más habitual de sistemas de control embebidos, los cuales hacen uso de microcontroladores. Estos controladores presentan limitaciones computacionales, por lo que el empleo de técnicas de muestreo periódicas supone un hándicap importante para ellos. Además, estos sistemas son muy compactos y

apenas cuentan con espacio para fuentes de energía, por lo que también necesitan que el consumo energético sea muy eficiente. En este tipo de sistemas las comunicaciones inalámbricas suponen una parte importante de su consumo energético, por lo que conseguir reducir las comunicaciones tiene un gran impacto en la autonomía de los mismos.

Teniendo en cuenta lo anterior, es conveniente explorar otros enfoques. Por ejemplo, existe una rama dentro de los sistemas de control la cuál basa la toma de decisiones en la aparición de eventos en el sistema. Se trata de una idea cuyo germen aparece ya en las décadas de los sesenta y los setenta del siglo pasado como sistemas con una frecuencia de toma de muestras adaptativa [2] (R. Dorf, M. Farren and C. Phillips, 1962) [3] (T. Hsia 1974).

Sin embargo, no es hasta el año 1999 y la aparición de los artículos de Karl-Erik Årzen [4] y Karl Johan Åström y Bo Bernhardsson [5] que esta tecnología capta el interés de la comunidad. Esto puede apreciarse en el aumento casi exponencial en la cantidad de literatura publicada al respecto desde entonces, como se recoge en el artículo de E. Aranda-Escolástico et al. (2020) [6].

Entre las ventajas que presenta el control por eventos, destaca la mayor eficiencia de uso de la CPU, ya que no necesita interrumpir periódicamente las tareas que esté haciendo para comprobar las señales que le llegan y decidir si ha de llevar a cabo alguna nueva acción, es sólo cuando ocurre un evento que ha de prestarle atención y actuar en consecuencia.

Es justamente este comportamiento el que, tal y como señalan Dormido, Sánchez y Kofman (2008) [7], aporta una de las principales mejoras en los sistemas de control. Se trata de la reducción del tiempo de respuesta ante la modificación de uno de los parámetros a controlar.

Otra ventaja de este tipo de sistemas es la reducción del tráfico de información en el bus de comunicaciones, lo que conlleva una reducción del ancho de banda necesario. Se trata de una característica muy interesante y que hace que esta tecnología sea muy atractiva para los sistemas de control basados en red o NCS (Networked Control Systems) [8] (L. de la Torre et al., 2019).

Podemos, por tanto, señalar que el despliegue y la cada vez mayor penetración de las redes de comunicaciones y las diferentes tecnologías asociadas a las mismas que se ha

experimentado en las últimas décadas, es uno de los principales motivos del auge de este tipo de sistemas de control.

Otro aspecto a favor estos sistemas es la mayor eficiencia energética de los mismos. Como indican Liu, Wang, He y Zhou (2014) [9], en los sistemas del control periódicos cada vez que se cumple el tiempo de muestreo se realizan ajustes sobre los actuadores con el consiguiente gasto energético y desgaste de los materiales. Mientras que en los sistemas de control por eventos se reduce el gasto energético únicamente a cuando se producen eventos.

Esta es una característica muy interesante especialmente para equipos autónomos y móviles tales como vehículos o robots, los cuales no disponen de una fuente constante de energía. La reducción del consumo les permite disponer un mayor tiempo de autonomía y con ello extender su tiempo de trabajo. Además, el comportamiento de este tipo de dispositivos se adapta mejor a un control basado en eventos más que a uno periódico, ya que normalmente deben responder únicamente cuando se producen determinados cambios en su entorno y no de forma constante. De igual manera, esta cualidad es muy interesante en aquellos sistemas en los que los sensores y actuadores se encuentran alejados o en lugares remotos, sin conexión cableada y lejos de fuentes de energía, no sólo por el ahorro energético sino también por la reducción de las necesidades de mantenimiento.

1.2.- OBJETIVO

El objetivo de este trabajo fin de master es añadir la capacidad de dar soporte a métodos de control basado en eventos a la herramienta RIP-LabVIEW server, dentro del marco de desarrollo de Laboratorios Remotos para enseñanza de Control Automático. Para ello, nos basaremos en el Protocolo Remoto de Interoperabilidad desarrollado por la UNED para laboratorios online y al que, en adelante, nos referiremos por las siglas RIP, acrónimo del inglés Remote Interoperability Protocol.

1.3.- ANTECEDENTES

1.3.1.- PROTOCOLO RIP

El auge que han experimentado las redes de comunicaciones, en especial Internet, ha dado lugar a que se desarrollen multitud de equipos dotados de conectividad que posibilitan usar las redes para obtener datos de los mismos o controlarlos. Este alto grado

de conectividad ha permitido el desarrollo de los sistemas NCS, aunque para ello es necesario el desarrollo de herramientas que permitan establecer comunicaciones con dichos equipos.

A día de hoy existen en el mercado un buen número de programas enfocados a la ingeniería que permiten el modelado de sistemas o que disponen de drivers que les permiten comunicarse con todo tipo de dispositivos físicos. Si bien el desarrollo de todo el modelado y uso de drivers, se lleva a cabo con estos programas, lo normal es que la interfaz que se le presenta al usuario para poder comunicarse e interactuar con los dispositivos se desarrolle con otro software que permita un entorno más amigable. Dicha interfaz no tiene que encontrarse necesariamente en el equipo del usuario, sino que puede estar en otro equipo con el que el usuario puede establecer una conexión.

De lo explicado anteriormente se deduce la necesidad de disponer de una forma de establecer una comunicación entre ambos tipos de software. Es aquí donde surge la idea de la elaboración de un Protocolo Remoto de Interoperabilidad (RIP) que permita dicha interconexión. La intención era que fuese un protocolo sencillo y efectivo que permitiese un alto grado de independencia.

Finalmente se desarrolló un protocolo de código abierto, con un modelo de cliente-servidor, donde el cliente se ejecuta desde un navegador web. Es por ello que se basa en el protocolo HTTP estándar, el cuál es compatible con los principales navegadores del mercado. El uso de la tecnología Server-Sent Events (SSE) habilita para realizar la implementación de los mecanismos basados en eventos. Es importante señalar que tanto el navegador Microsoft Explorer como el Microsoft Edge de versiones anteriores a la setenta y cinco no soportan de forma nativa esta tecnología, por lo que es necesario recurrir a soluciones que les añaden dicha capacidad.

Este protocolo ha sido diseñado para que pueda comunicarse tanto con laboratorios virtuales (VI), los cuales consisten en simulaciones de entornos reales basadas en modelos matemáticos, o laboratorios remotos, en los que se controla de forma remota equipamiento real. A cada una de las actividades que se pueden desarrollar con un laboratorio online, ya sea virtual o remoto, se ha dado en llamarla experiencia. Los clientes pueden acceder de forma sencilla tanto a la información acerca de cada una de las experiencias que existan definidas en el servidor, como a los valores de sus entradas y salidas.

Las posibilidades de comunicación que proporciona este protocolo, tanto con equipamiento real como con laboratorios virtuales, así como el hecho de que pueda funcionar sobre HTTP/HTTPS dependiendo de las necesidades, permite que pueda emplearse en aplicaciones web y funcionar desde cualquier dispositivo con acceso a un cliente web, hace que sea especialmente interesante en el campo de la enseñanza a distancia de Ingenierías [10] (L. de la Torre et al., 2019). En este modelo de enseñanza resulta complicado el poder llevar a cabo una formación práctica, debido entre otras cosas, al coste de los equipamientos, lo que impide que la mayoría de alumnos puedan disponer de ellos o tengan acceso a los mismos en algún lugar cercano. Este protocolo permite implementar un servidor que comunique con los equipos de laboratorio, con lo que los estudiantes sólo necesitan disponer de un ordenador con acceso a internet para poder tener acceso a los mismos, eliminando así las limitaciones tanto espaciales como temporales.

Las principales características del RIP son las siguientes:

- Permite crear experiencias en el laboratorio online.
- Proporciona los meta datos de cada una de las experiencias.
- Proporciona la lista de variables de lectura y de escritura de cada experiencia.
- Proporciona la lista de métodos de lectura y de escritura de variables para cada experiencia.
- Dispone de métodos de activación para la lectura y escritura de variables en cada experiencia.
- Permite definir servidores de eventos (SSE), los cuales pueden enviar información de la experiencia bien periódicamente o al producirse un evento definido en la propia experiencia.
- Permite de que un cliente se suscriba a cualquier servidor de eventos definido en una experiencia.

La estructura de un servidor que implementa el protocolo RIP puede descomponerse en tres partes: por un lado, el servidor web, el cual se encarga de gestionar las conexiones de los clientes, las sesiones de los usuarios y demás; por otro lado, el intérprete de comandos, que entiende RIP e interpreta las órdenes recibidas; en tercer lugar, tenemos al ejecutor, que es el programa encargado de llevar a cabo el laboratorio aplicando las órdenes recibidas. Una representación de dicha estructura y sus comunicaciones con los clientes y programas de control puede verse en la Figura 1.

Figura 1. Estructura de un servidor RIP

2.- DESARROLLO

2.1.- BASE CONCEPTUAL PARA LA DETECCIÓN DE EVENTOS

Atendiendo a la definición de lo que es un evento vemos que es algo que sucede, sin embargo, esta definición resulta algo genérica. Por tanto, lo primero que vamos a hacer es definir lo que en este trabajo vamos a considerar como un evento. En nuestro caso, consideraremos que se ha producido un evento cuando una variable de estado del sistema experimente un cambio en su medida superior a un valor prefijado, al que se suele llamar delta. Este tipo de detección de eventos se conoce por su nombre inglés Send-On-Delta. La aparición de un evento hace que el SSE lleve a cabo un intercambio de información con el cliente.

De la propia definición se deduce la necesidad de comparar la señal consigo misma en distintos instantes para poder determinar si se ha producido una variación que de lugar a un evento. Surge por tanto la obligación de obtener el valor de la señal cada cierto tiempo para poder llevar a cabo dicha comparación. En nuestro caso estableceremos una base de tiempo periódica, denominando a la frecuencia con la que se observa el valor de la señal como frecuencia de muestreo. Dependiendo del sistema que se quiera controlar habrá que ajustar el valor de dicha frecuencia para que detecten correctamente los eventos que se puedan producir.

2.2.- DESARROLLO PREEXISTENTE

El trabajo que se lleva a cabo en este TFM no parte de cero, sino que se apoya en un servidor RIP creado con anterioridad con el software LabVIEW y que ha servido de punto de partida [11] (L. de la Torre, 2019). Sin querer entrar en profundidad en los diferentes elementos de la estructura que compone el servidor RIP, sí que nos parece necesario explicar aquellas partes necesarias para facilitar la comprensión del trabajo posteriormente desarrollado.

2.2.1.- CONFIGURACIÓN DEL SERVIDOR

En primer lugar, el servidor cuenta con un VI denominado Global_Configurations.vi el cual se emplea para configurar diferentes características del servidor. En la Figura 2 se presenta una captura de pantalla de este fichero. La parte superior del VI contiene una serie

de variables que están relacionadas con la configuración de la conexión del servidor y que no forman parte del alcance de este proyecto, por lo que no vamos a entrar a detallarlas. La parte que resulta de interés para este TFM es la lista que se encuentra en la parte inferior y que se denomina Experiments. Es en esta parte donde se han de declarar las experiencias que se creen y que se quiera que estén disponibles en el servidor RIP.

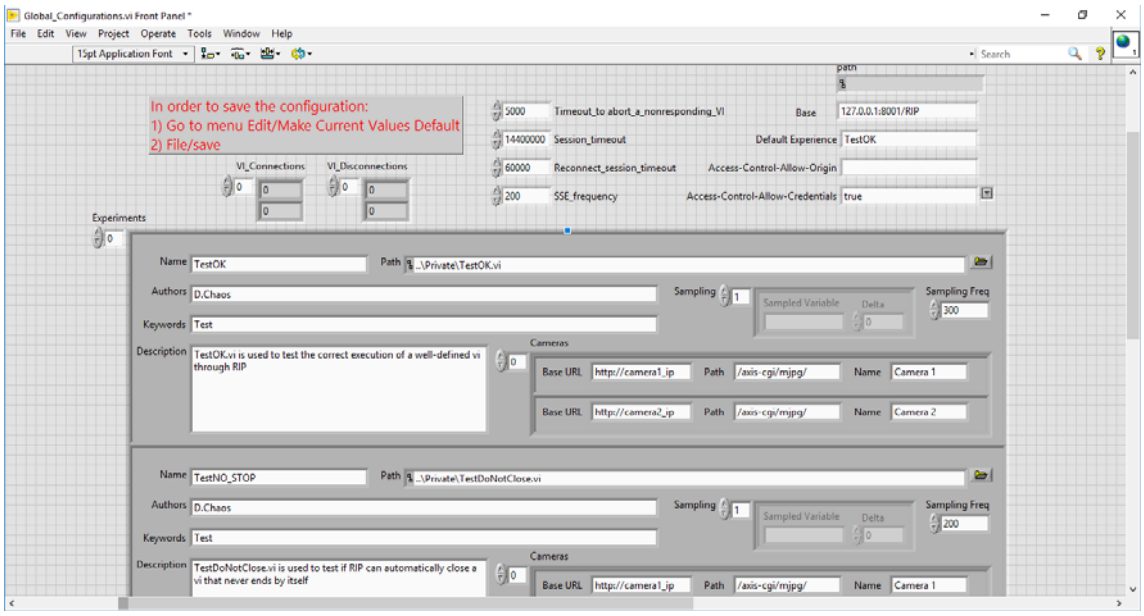

Figura 2. Aspecto del fichero Global_Configurations

Si nos fijamos en la lista pueden verse las dos experiencias que vienen por defecto con el servidor *TestOK* y *TestDoNotClose*, la primera en su totalidad y la segunda parcialmente. A continuación, explicaremos la utilidad de cada uno de los distintos campos que aparecen:

- Name: Nombre que se asigna a la experiencia y que el cliente deberá usar para acceder a ella.
- Path: Dirección en la que se encuentra el VI que arranca la experiencia.
- Authors: El autor o autores de la experiencia
- Keywords: Palabras clave que definen la experiencia
- Description: Breve descripción de la experiencia
- Sampling: Es una lista en la que se declaran aquellas variables de la experiencia de las cuales queremos controlar por eventos. Dentro de la lista disponemos de dos campos, *Sampled Variable* en el que se introduce el nombre de la variable a controlar, y *Delta* donde se introduce la desviación que ha de producirse entre dos valores de la señal para que se produzca el evento.

- Sampling Freq: frecuencia de muestreo con la que se comprobarán los valores de las variables de la experiencia.
- Cameras: listado de las cámaras usadas por la experiencia y en el que pueden definirse su nombre, dirección y URL.

No todos los campos anteriores son necesarios para que la experiencia funcione, los estrictamente necesarios son *Name* y *Path*. Los otros proporcionan información adicional o permiten activar la parte de control por eventos si se desea, ya que si en el listado *Sampling* no se define nada actuará como un servidor de datos periódicos.

2.2.2.- FUNCIONAMIENTO DEL SERVIDOR

El siguiente VI que vamos a comentar es el denominado SSE.vi. Este bloque es el que lleva a cabo la comunicación con los clientes y cuenta en su interior con diferentes secciones las cuales son responsables de distintas funciones, empleando para ello llamadas a otros VIs con los que cuenta el servidor. La primera parte del código se ejecuta cuando un cliente se conecta al servidor, y se encarga de comprobar si el cliente ya tiene una sesión iniciada, lo que implica que se está ante una reconexión, o bien si se trata de una conexión nueva. En el caso de tratarse de una reconexión se deberían de proporcionar al cliente los eventos que se han producido desde su desconexión hasta que se produce la reconexión antes de pasar a volver a proporcionar los eventos que se producen en tiempo real, mientras que en el caso de una nueva conexión se pasa directamente a proporcionar los eventos que se producen en tiempo real.

Una vez se termina la ejecución de la primera parte del código se pasa a la siguiente sección que es la que se ejecuta de forma constante mientras haya un cliente conectado al servidor para entregar la información de los eventos que se producen en tiempo real. Esta sección se encarga de obtener periódicamente los valores de las variables, mediante el VI InvokeForJSONResult.vi, para posteriormente determinar si se ha producido un evento. El VI existente para detectar eventos es el denominado Events_Composer.vi, si bien en este punto inicial sólo genera respuestas periódicas. Todo lo explicado hasta ahora respecto al SSE.vi se corresponde con lo que se puede ven en la Figura 3.

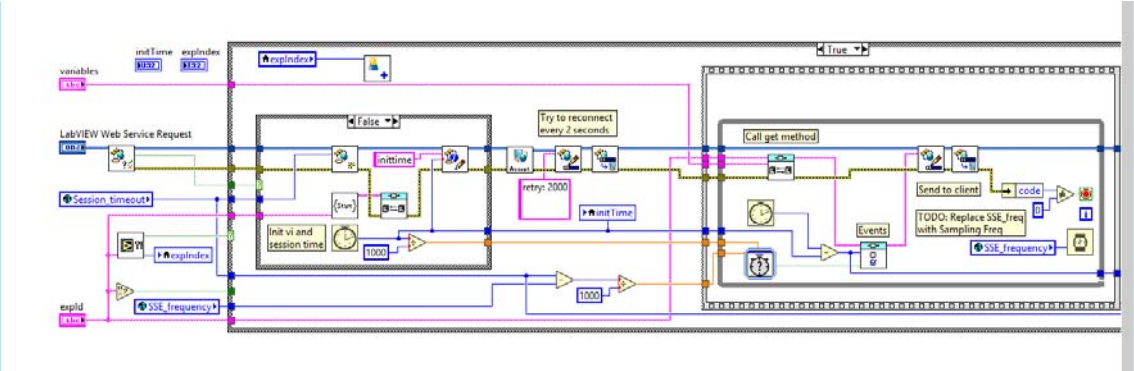

Figura 3. Primera parte de la estructura del fichero SSE

La tercera sección del código es la que se muestra en la Figura 4. En esta sección se define el comportamiento del servidor SSE cuando una sesión de un cliente se desconecta del mismo. Básicamente consta de tres partes, en la primera lo que se hace es esperar un tiempo por si el cliente se reconecta. Durante este tiempo se sigue ejecutando la experiencia y los eventos que se producen se almacenan en un fichero para poder entregárselos al cliente en caso de reconexión. De todo esto se encarga el VI Store_Missed_Events.vi. La segunda parte se ejecuta una vez se ha cumplido el tiempo de espera de reconexión o si se produce la reconexión, aquí lo que se hace es eliminar el fichero creado en la primera parte con los eventos almacenados. La tercera parte consiste en el cierre de sesión del cliente una vez ha superado el tiempo máximo de espera.

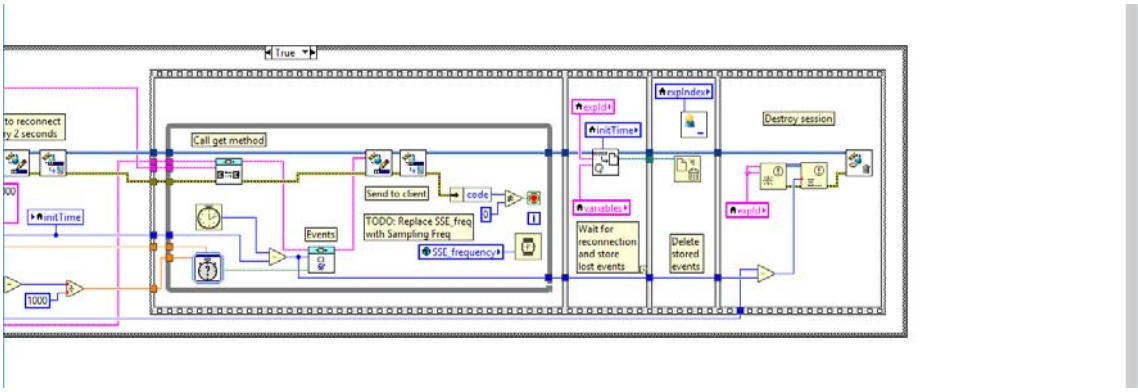

Figura 4. Segunda parte de la estructura del fichero SSE

Otro aspecto relevante de este servidor es su modo de funcionamiento, ya que se puede ejecutar de dos maneras. La primera es el llamado modo depuración, pensado para la realización de las pruebas necesarias de la lógica desarrollada. En este modo el servidor permite acceder a los distintos VI y comprobar los valores de sus distintas variables. La segunda opción se denomina modo producción y es el modo en el que el servidor va a

trabajar normalmente. En este caso no es posible abrir los VI que forman parte del proyecto y ver los valores de sus variables.

2.2.3.- RESPUESTA ENTREGADA AL CLIENTE

Respecto a la respuesta que se entrega al cliente, se obtiene de una composición que se hace dentro del Events_Composer.vi de la respuesta que entrega el VI InvokeForJSONResult.vi con la variable de tiempo transcurrido en la experiencia. La respuesta del InvokeForJSONResult.vi está formada por dos vectores, uno con el nombre de todas las variables de la experiencia y otro con el valor de dichas variables.

2.3.- TRABAJO DESARROLLADO

2.3.1.- FRECUENCIA DE MUESTREO

En esta parte pasaremos a explicar las modificaciones realizadas al servidor preexistente para mejorar su comportamiento. La primera tarea que se llevó a cabo tiene que ver con la frecuencia de muestreo. Anteriormente hemos señalado que el valor de dicha variable se ha de adaptar a las necesidades del sistema en el que se está trabajando. Si nos fijamos en la Figura 5, donde se muestra la parte del SSE.vi encargada de comprobar periódicamente los valores de las variables, observamos que está empleando como base de tiempo la variable global SSE_frequency. Por tanto, con esta configuración, todas las experiencias realizan la comprobación de eventos con la misma frecuencia. Esto supone una limitación, ya que no permite adaptar cada laboratorio a la frecuencia que mejor le viene.

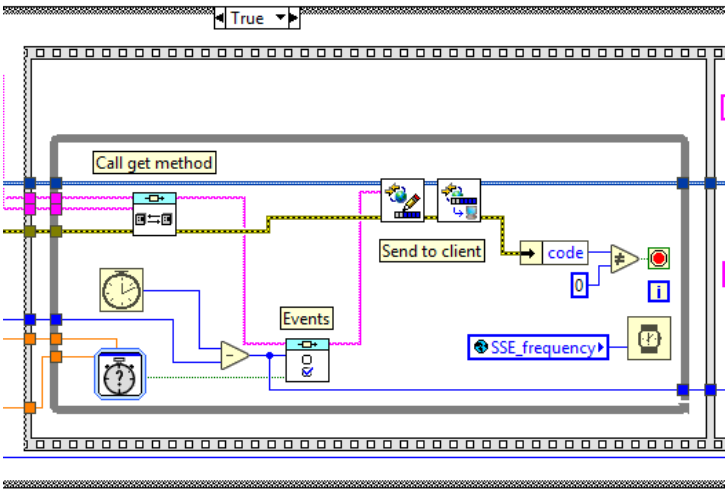

Figura 5. Detalle del uso de la SSE_frequency

Al explicar el Global_Configurations.vi vimos que cada experiencia cuenta con una variable llamada *Sampling Freq* que sirve para definir la frecuencia de muestreo. Es por ello que se realizó la modificación que se muestra en la Figura 6 de forma que la frecuencia que se emplee al comprobar los valores se corresponda con la establecida en la *Sampling Freq* de cada experiencia.

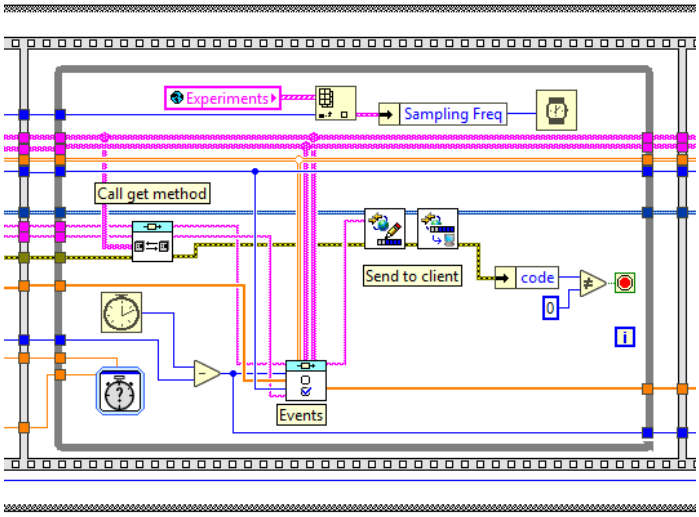

Figura 6. Modificación para el empleo de la *Sampling Freq*

2.3.2.- DETECCIÓN DE EVENTOS

El siguiente paso, fue trabajar en la detección de eventos en sí. Para detectar un evento hemos de obtener el valor de las variables que estamos controlando para luego compararlas con el valor obtenido en el anterior ciclo de muestreo, de forma que podamos determinar si su variación es mayor que el valor de delta definido para las mismas.

Cada experiencia tiene sus propias variables definidas por eventos, cada una con su propio valor de delta. Todos estos datos se introducen en el VI Global_Configurations.vi, por lo que primero que el servidor ha de hacer cada vez que un cliente le solicita conectarse a una experiencia es obtener dicho listado de variables y deltas para poder empezar a controlarlas. Originalmente el servidor no disponía de ningún VI que llevara a cabo esta labor, así que se creó el Obtain_Variables_and_Deltas.vi, cuya estructura interna se muestra en la Figura 7.

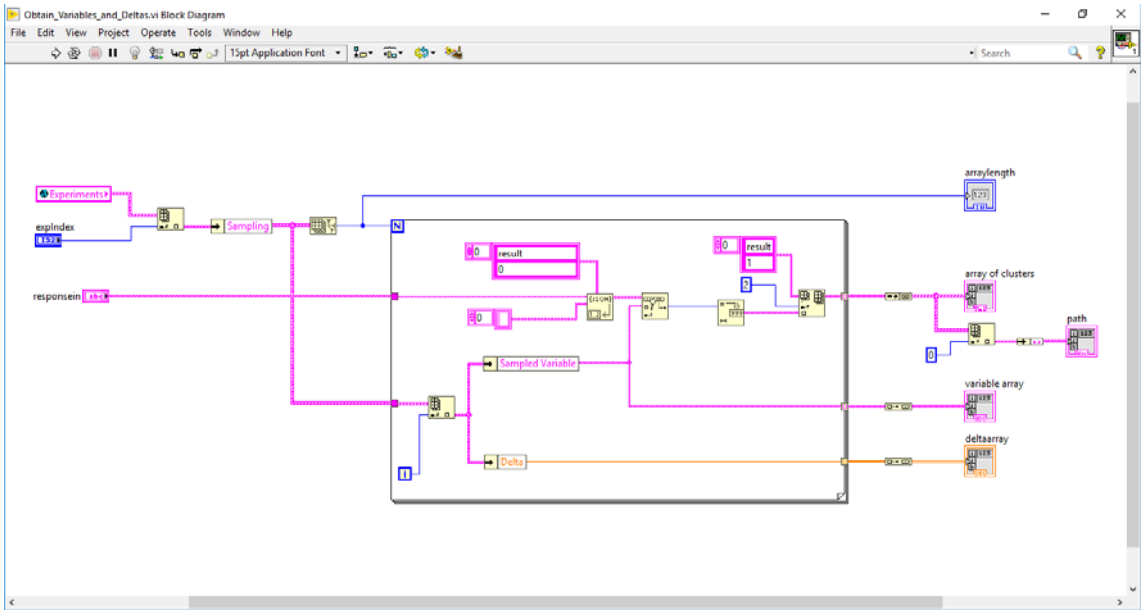

Figura 7. Estructura interna del Obtain_Variables_and_Deltas.vi

La función de este VI es sencilla, a partir de los datos de las variables globales del Global_Configurations.vi y de la referencia de la experiencia que se está ejecutando (*expIndex*) proporciona a la salida un vector con el nombre de las variables de la experiencia definidas para ser controladas por eventos (*variable array*), un vector con las deltas de dichas variables (*deltaarray*), un vector con la posición de los valores de cada variable dentro de la respuesta del servidor (*path*) y el número total de variables controladas por eventos con el que cuenta la experiencia (*arraylength*).

El siguiente paso es obtener los valores de las variables en cada ciclo de muestreo, para esto ya contábamos con el VI InvokeForJSONResult.vi que se encarga de proporcionárnoslos.

Se pasa entonces al paso definitivo dentro de la detección de eventos, el cuál se ejecuta dentro del VI Events_Composer.vi. Una imagen de su programación original se muestra en la Figura 8. Tal y como está diseñado inicialmente este VI proporciona una respuesta periódica con el nombre y valor de todas las variables con las que cuenta la experiencia, así como una respuesta única la primera vez que se ejecuta el mismo. También cuenta con una sección preparada para proporcionar una respuesta en caso de que se produzca un evento, pero no tiene desarrollada la parte que se ha de encargar de detectar el evento.

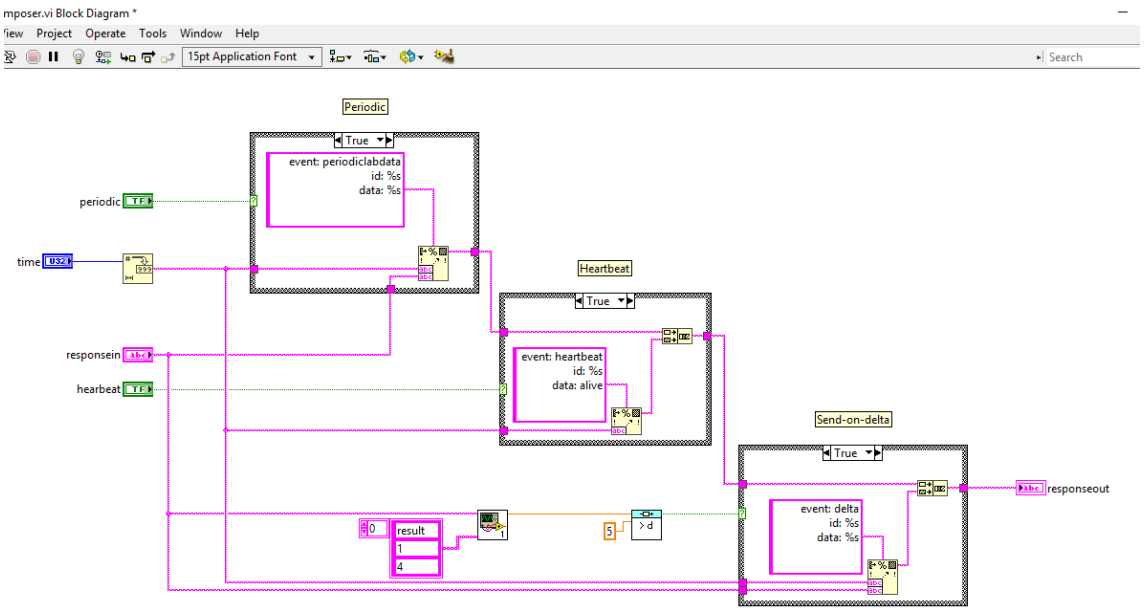

Figura 8. Estructura interna original del Events_Composer.vi

En su estado inicial la parte encargada de detectar un evento está compuesta por dos VIs. El primero de ellos, de nombre Obtain_Variable_Value.vi, está diseñado para, a partir de la respuesta proporcionada por el vi InvokeForJSONResult.vi, extraer el valor de una variable cada vez que se ejecuta. Para hacerlo necesita conocer la posición que ocupa dicho valor en el vector de respuesta que recibe. En esta configuración inicial la declaración de dicha posición es fija, por lo que sólo se puede obtener eventos de una sola variable de la experiencia, independientemente de cuantas variables estén configuradas para ser detectadas como eventos.

Una vez obtenido el valor de la variable de este modo se lleva al segundo VI, denominado Delta.vi. Este bloque es el encargado de comparar el valor actual de la variable con el de la iteración anterior y determinar si su variación es mayor que el valor de delta para, en caso afirmativo, indicarnos que se ha producido un evento y almacenar dicho valor, que pasa a ser el de referencia con el que se han de comparar los valores que se obtengan en las siguientes iteraciones. Sin embargo, al igual que pasaba en el caso anterior, el diseño original de este bloque está pensado para una única variable y el valor de delta no lo toma del que se le ha otorgado al definir la experiencia en el Global_Configurations.vi sino que se le introduce un valor fijo que sería el mismo para cualquier variable de cualquier experiencia.

Teniendo en cuenta lo anterior se procedió a diseñar una detección de eventos apropiada, de forma que el servidor sea capaz de poder proporcionar eventos de todas las variables definidas como *Sampled Variable* en la configuración de una experiencia y para

que cada una de ellas se emplee la delta que le haya hubiese asignado. Esto se ha conseguido modificando los VIs anteriormente mencionados. Empezaremos explicando el trabajo desarrollado con el Delta.vi, cuya estructura original puede verse en la Figura 9.

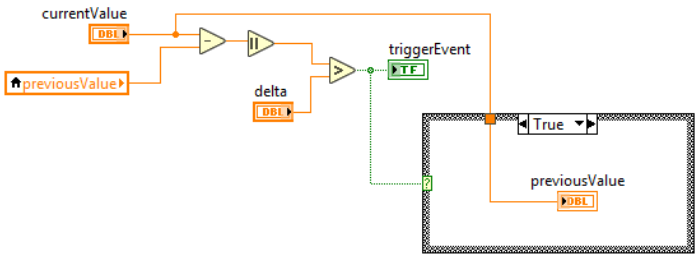

Figura 9. Estructura interna original del Delta.vi

En este caso tuvimos que sustituir la entrada *delta* y la variable interna previousValue por sendos vectores a los que se nombraron como deltaArray y outputArray. El primero contiene ahora todos los valores de delta de las variables definidas en la experiencia, mientras que el segundo es el que se emplea para almacenar el valor de la variable cuando se produce un evento de la misma y poder compararla con los valores que lleguen en siguientes iteraciones. En función del valor de una nueva señal de entrada denominada *index* se determina que posición del array se quiere revisar en cada iteración. Se ha añadido también un vector de entrada denominado inputArray, el cual tiene una doble funcionalidad. En el caso de que se trate de la primera iteración del programa para un cliente este array sirve para inicializar a cero el valor de outputArray, y en caso de que se trate de la primera iteración tras una reconexión del cliente sirve para entregar los valores que tenían las variables en los momentos previos a la reconexión. Este VI modificado se renombró como Delta_Array.vi y la parte del mismo que hemos explicado puede verse en la Figura 10.

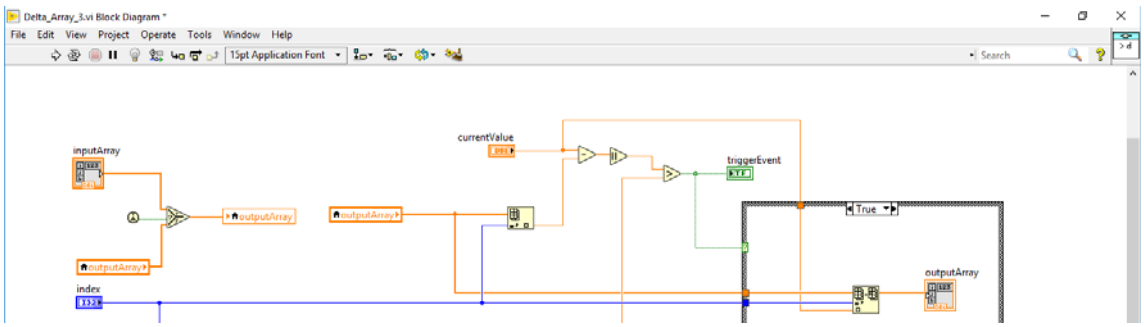

Figura 10. Estructura interna parcial del Delta_Array.vi

Al igual que pasaba con Delta.vi el VI Obtain_Variable_Value.vi estaba diseñado para obtener únicamente el valor de una sola variable, por lo que resultaba necesario modificarlo para que pudiese obtener el valor de todas las variables de la experiencia. La Figura 11 presenta la estructura original de este VI.

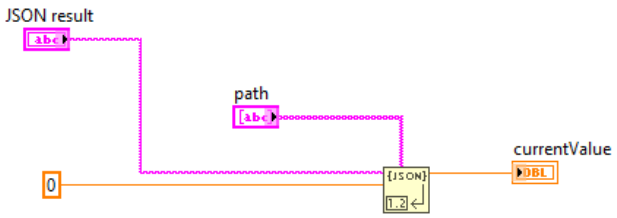

Figura 11. Estructura interna original del Obtain_Variable_Value.vi

En este caso la modificación necesaria es más sencilla que en el anterior. Únicamente se ha tenido que sustituir la entrada simple *path* por un array que contiene la ubicación de cada una de las variables de la experiencia dentro de la cadena JSON. Al igual que en el caso interior se añade una entrada *index* que determina que posición del array se ha de leer en cada iteración del bloque.

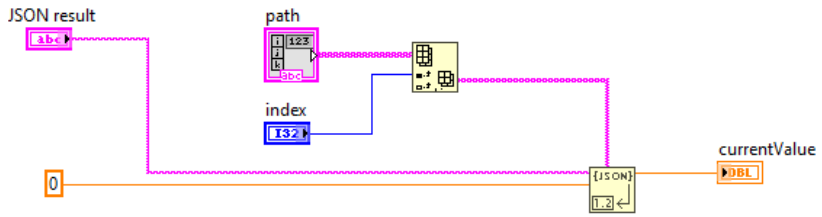

Figura 12. Estructura interna modificada del Obtain_Variable_Value.vi

El siguiente paso, una vez se hubo adaptado los dos sub VIs a las nuevas necesidades, fue modificar el propio Events_Composer.vi para integrarlos. Anteriormente hemos hablado de la creación de una señal *index* en los dos VIs modificados para recorrer los nuevos vectores que hemos creado, esto sirve para que podamos verificar si se ha producido un evento en cualquiera de todas las variables configuradas por eventos en una experiencia. Por tanto, necesitamos que en cada iteración del Events_Composer.vi dicha señal *index* tome todos los valores necesarios para que se recorra el total de estos vectores. Esto lo conseguimos añadiendo un bucle que se encarga de esto.

Otro aspecto a tener en cuenta es que ahora existe la posibilidad de que en una iteración del Events_Composer.vi haya más de una variable en la que se haya producido un evento, por lo que si tomamos tal cual la salida triggerEvent procedente del Delta_Array.vi y la llevamos a la parte del código que se encarga de generar la respuesta ante un evento, podemos encontrarnos con que en una iteración nos encontramos con múltiples respuestas de evento entregadas al cliente, ya que código se ejecuta de forma simultánea. Si bien este comportamiento en si no es incorrecto, no lo consideramos el más adecuado ya que la emisión de tantos mensajes conlleva mayor consumo de ancho de banda, así como un exceso de mensajes presentados al cliente lo que puede hacer que éste no llegue a poder asimilar correctamente toda la información.

Para evitar esto hemos introducido una estructura de secuencia de dos pasos, en el primer paso de la secuencia hemos incluido el nuevo bucle explicado anteriormente y en el segundo paso se encuentra la parte encargada de generar la respuesta ante un evento. De este modo nos aseguramos que primero recorremos todas las variables y una vez que hemos terminado y sabemos que se ha producido un evento en una o más de una, sólo entonces generaremos un único mensaje con la información de todos los eventos producidos en dicha iteración. Todo esto puede verse en la Figura 13.

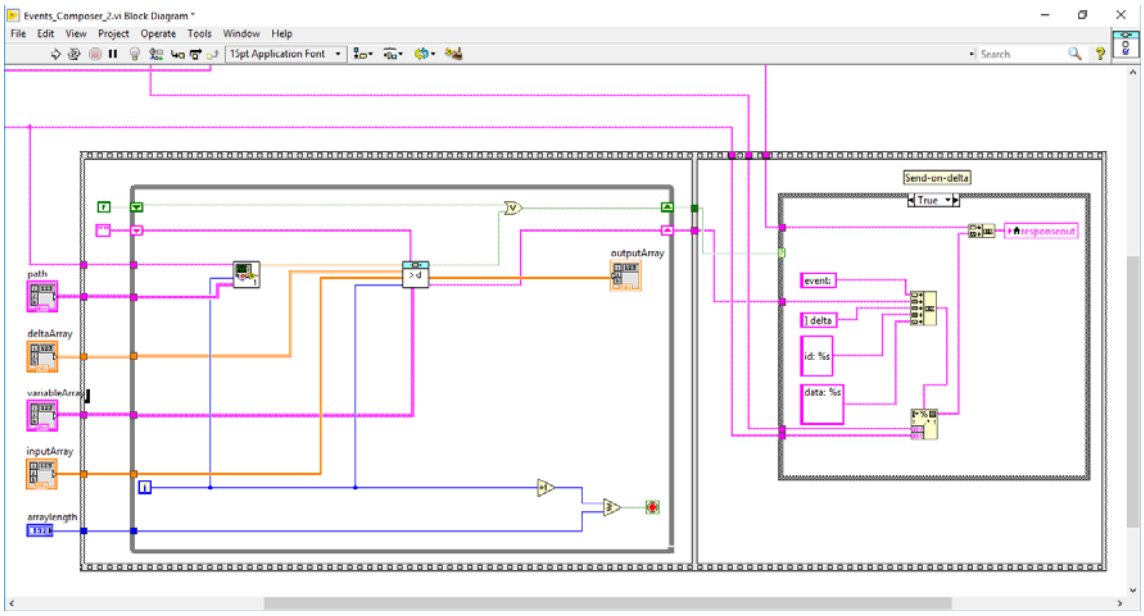

Figura 13. Estructura interna parcial del Events_Composer.vi modificado

Para que el mensaje único explicado anteriormente pudiera contar con el nombre de todas las variables que habían experimentado un evento durante una iteración hubo que

añadir una parte de código en el VI Delta_Array.vi que se encargara de almacenar dichos nombres. Esta parte del código es la que se presenta en la Figura 14.

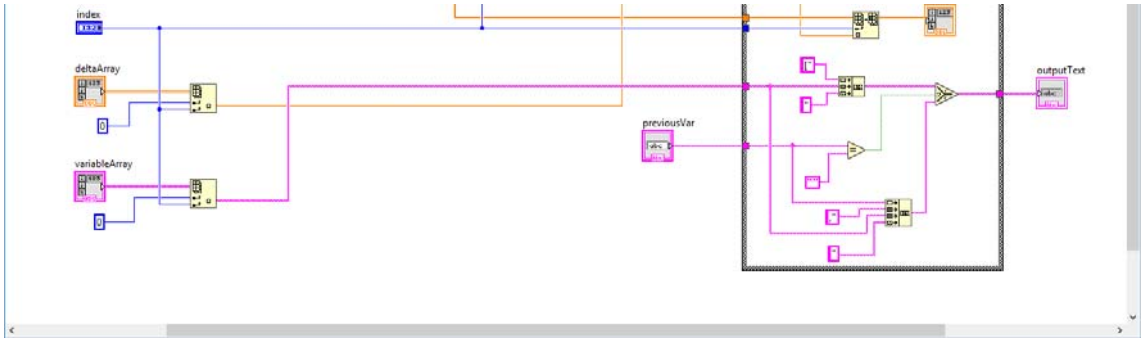

Figura 14. Estructura interna parcial del Delta_Array.vi

Adicionalmente se ha eliminado la parte del código que enviaba un mensaje la primera vez que se ejecuta el VI, esta parte del código tenia utilidad en etapas iniciales del desarrollo del VI, como forma de comprobación de que se estaba accediendo a él, pero en este punto ya no era necesaria, es por ello que se optó por su eliminación. Lo que sí se mantuvo fue la respuesta periódica, ya que sirve para que los usuarios puedan comparar entre la cantidad de información que se manda en el caso de comprobaciones periódicas y en el caso de detección de eventos. En la Figura 15 se muestra como quedó esta parte del VI.

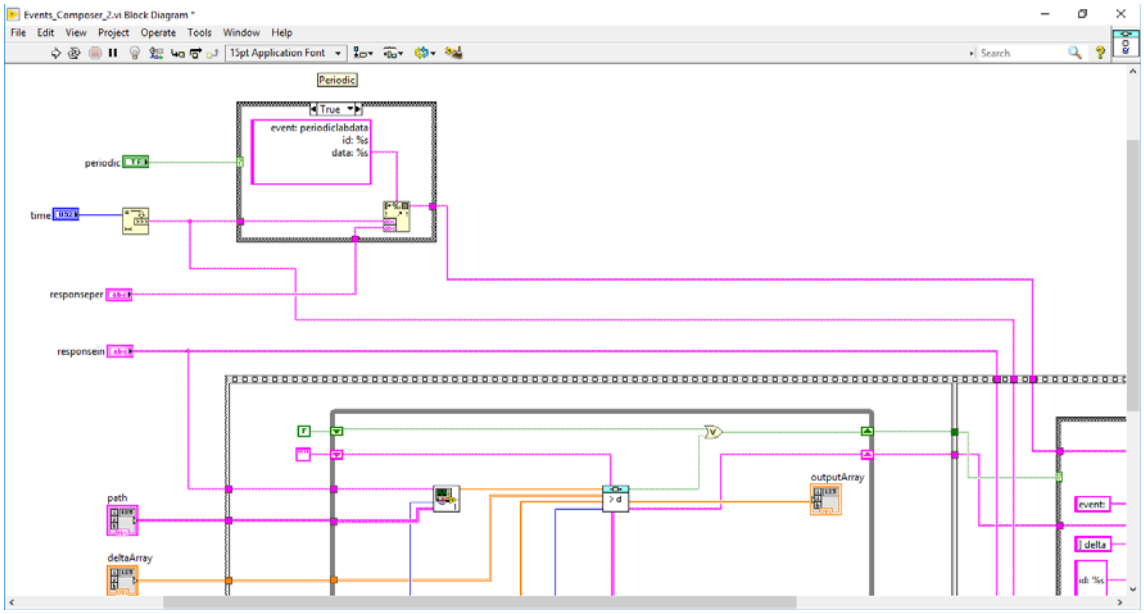

Figura 15. Estructura interna parcial del Events_Composer.vi modificado

2.3.3.- RESPUESTA ENTREGADA AL CLIENTE

Otro aspecto que se ha cambiado respecto al servidor original es la forma de presentar los datos. En su estado original el servidor proporciona al cliente los datos del estado de todas las variables de la experiencia independientemente de que se tratara de una respuesta periódica o de una respuesta debida a un evento. Además, en caso de un evento no indicaba que evento en concreto se había producido, sino que se identificaba de forma genérica. Esto generaba una cantidad de información excesiva y que no ayudaba a identificar que estaba pasando en cada momento. Es por ello que se decidió dividir en dos la respuesta que se envía al cliente, una respuesta periódica y otra para los eventos. En el caso de la respuesta periódica se proporciona sólo el nombre y el valor de aquellas variables que no han sido declaradas en la lista de variables por eventos. En el caso de una respuesta por eventos se identifica la variable o variables que han causado el evento, pero además se entregan el nombre y el valor de todas las variables de la experiencia.

El VI encargado de obtener la parte de la respuesta enviada al cliente que contiene el nombre y el valor de las variables de una experiencia es el InvokeForJSONResult.vi, cuya estructura original puede apreciarse en la Figura 16. Se trata de un programa sencillo que utiliza el identificador de la experiencia (*expid*) junto con las variables de la misma y un sub VI denominado InvokeMethod.vi para proporcionar una respuesta (*response*) formada por dos cadenas de datos unidas, la primera con el nombre de las variables y la segunda con sus valores.

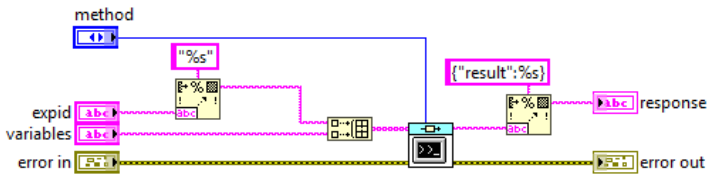

Figura 16. Estructura interna original del InvokeForJSONResult.vi

Atendiendo a la figura anterior es el VI InvokeMethod.vi el que entrega las dos cadenas de datos unidas. Se trata de un VI que en función del valor que recibe en su variable de entrada *method* ejecuta un código u otro. Lo que está recibiendo en la variable de entrada es un *get*, por lo que la estructura de código que está empleando es la que vemos en la Figura 17.

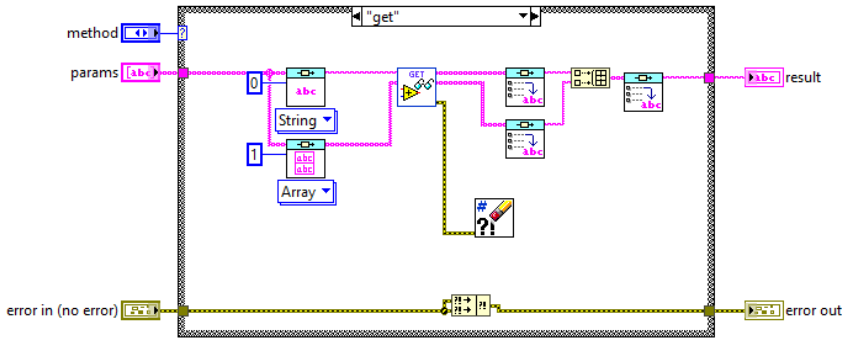

Figura 17. Estructura interna original del InvokeMethod.vi

Revisando esta estructura vemos como cuenta con un VI llamado Get.vi que es el que proporciona las dos cadenas de datos, variables y valores por separado. Por tanto, será este VI el que debamos modificar para obtener los nuevos tipos de respuesta que hemos indicado al principio de este apartado. La estructura original interna de este VI se presenta en las Figuras 18 y 19.

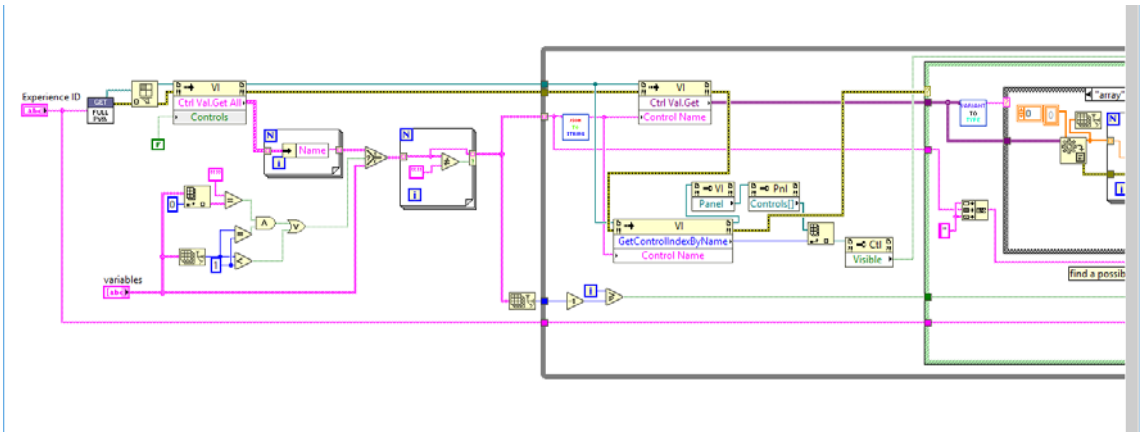

Figura 18. Estructura interna original del get.vi

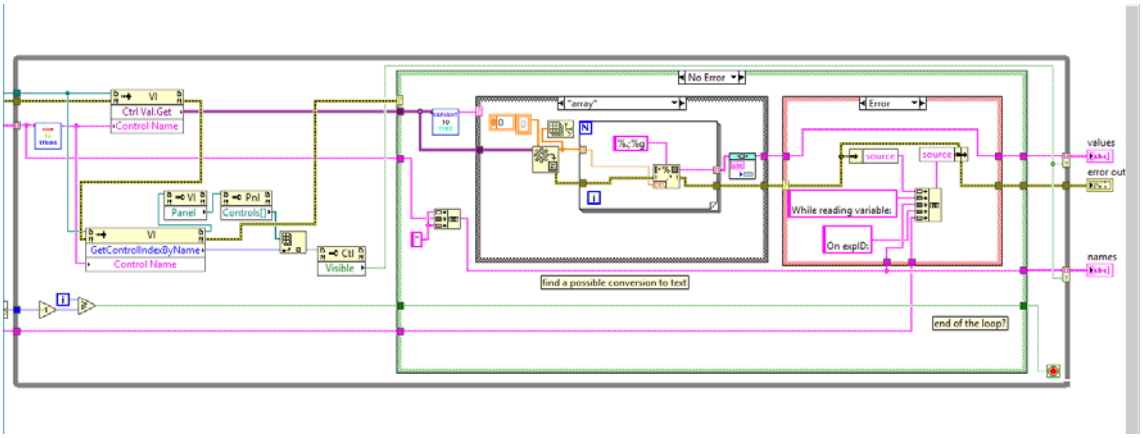

Figura 19. Estructura interna original del get.vi

Revisando la Figura 19 comprobamos que las cadenas *names* y *values* contienen los nombres y los valores de todas las variables de la experiencia, por lo que resultan válidas para generar la respuesta que se quiere presentar a un cliente ante un evento. Por lo tanto, lo que necesitamos es crear las cadenas que nos permitan crear la respuesta periódica, que solo contiene aquellas variables no configuradas como por eventos. Esto se ha conseguido creando un nuevo vector de entrada llamado variableArray, el cuál recibe el nombre de las variables que han sido configuradas para trabajar por eventos. En cada iteración se compara el nombre de la variable de la que se va a obtener el valor con los nombres almacenados en dicho vector, en caso de no coincidir con ninguno de ellos entonces se trata de una variable periódica y su nombre y valor se almacenan de forma concatenada en las nuevas variables de salida creadas para ello denominadas *periodic values* y *periodic names*. La estructura del Get.vi modificado queda como se ve en las Figuras 20 y 21.

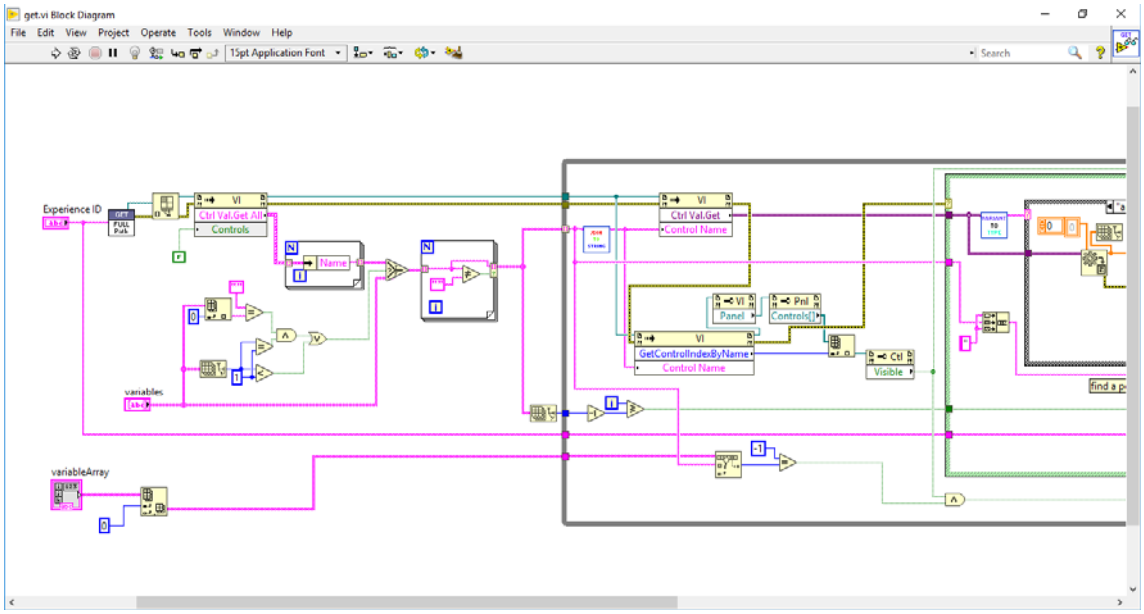

Figura 20. Estructura interna modificada del get.vi

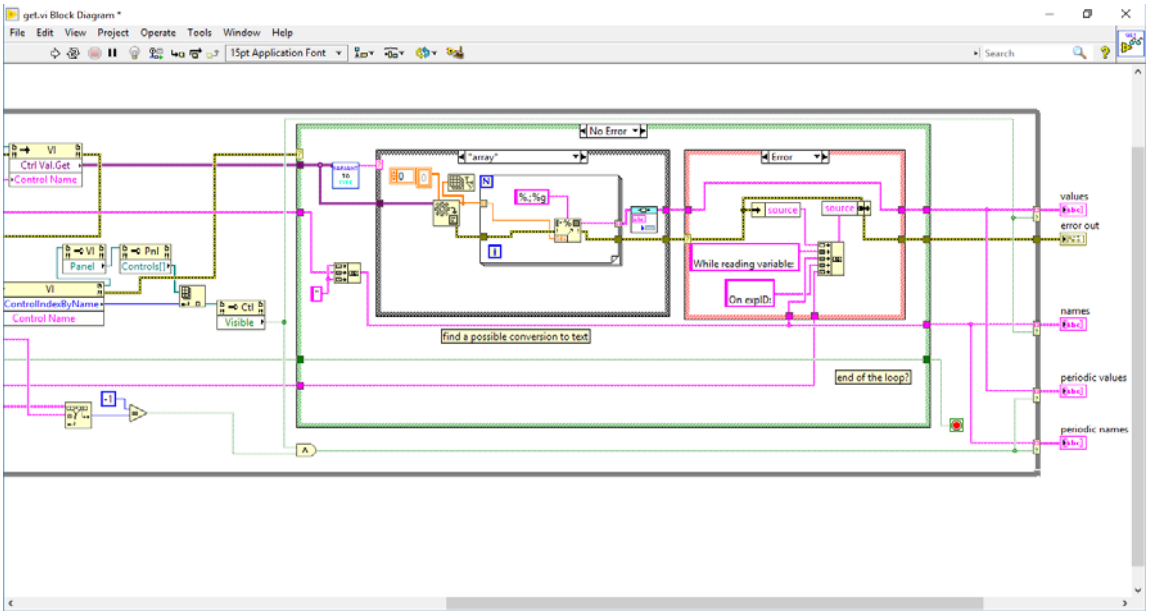

Figura 21. Estructura interna modificada del get.vi

Una vez modificado el Get.vi necesitamos modificar el InvokeMethod.vi, que es el que hace la llamada al Get, para que disponga de la nueva señal de entrada *variableArray* y para que combine las dos nuevas señales de salida provenientes del Get.vi en una nueva señal de salida llamada *resultper* que constituye la respuesta periódica. Estos cambios quedan reflejados en la Figura 22.

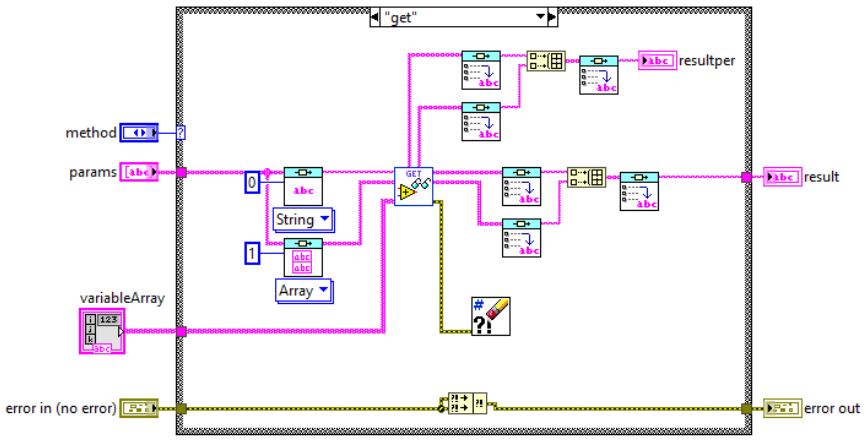

Figura 22. Estructura interna modificada del InvokeMethod.vi

El siguiente paso fue crear en el InvokeForJSONResult.vi un nuevo vector de entrada llamado variableArray, el cual se lleva al InvokeMethod. Este vector recibe únicamente el nombre de las variables que han sido configuradas para trabajar por eventos. Se añade también una salida de respuesta periódica (*reponseper*) ya con el formato que se quiere entregar al cliente. La estructura modificada se presenta en la Figura 23.

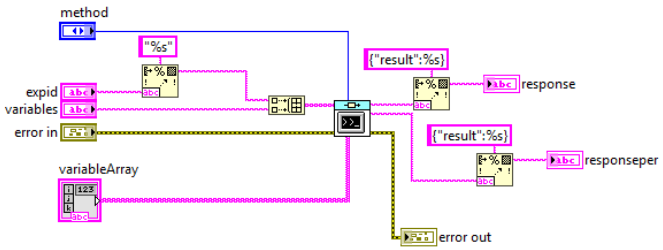

Figura 23. Estructura interna modificada del InvokeForJSONResult.vi

2.3.4.- ALMACENAMIENTO Y RECUPERACIÓN DE EVENTOS

El siguiente punto sobre el que se ha trabajado es el de la pérdida de eventos. En el caso de que un cliente sufriera una desconexión de la sesión, perdería la información relativa a los eventos que se han producido desde su desconexión hasta su reconexión. Con la intención de evitar esto, el servidor original cuenta con un VI llamado Store_Missed_Events.vi diseñado para seguir ejecutando la experiencia y guardar los eventos que se generan en un fichero, de forma que el cliente disponga de ellos en caso de que se reconecte antes de que transcurra el tiempo de reconexión de sesión. La Figura 24 muestra el contenido del VI comentado.

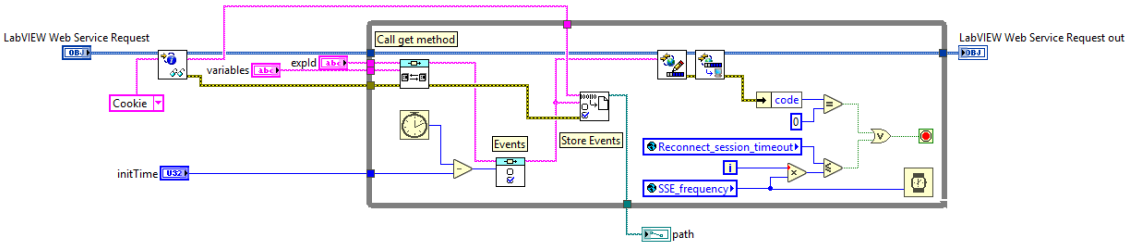

Figura 24. Estructura interna original del Store_Missed_Events.vi

Observando la figura nos encontramos con que se emplea un VI llamado Write_To_File.vi, el cuál se encarga de generar el fichero necesario y almacenar en él los eventos que se produzcan. El programa de este VI se puede ver en la Figura 25.

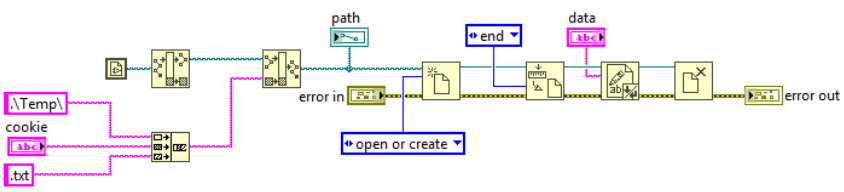

Figura 25. Estructura interna original del Write_To_File.vi

Al estudiar el comportamiento original de esta parte del código nos encontramos con un problema, el programa era incapaz de generar el archivo porque no encontraba la ruta del directorio. Fijándonos en la parte del programa donde se indica la ruta vemos que se emplea enrutamiento relativo (.\Nombre de Carpeta\), es decir, no se proporciona una ruta completamente definida, sino que la ruta va a estar en función de la ruta en la que se encuentra el VI que ejecuta el código. Sin embargo, dicha carpeta no existía en el proyecto original y de ahí el problema. Se optó entonces por modificar el principio del código añadiendo una parte en la que nos aseguramos de que la carpeta a la que apunta el enrutamiento se cree en caso de no existir previamente.

Una vez resuelto lo anterior nos encontramos con otro problema. Si bien el fichero se genera correctamente y su ruta se encuentra disponible en la variable de salida *path*, cuando el cliente se reconecta después de una desconexión lo que se ejecuta en el servidor cuando se hace la llamada al VI SSE es un clon de ese VI pero con sus propios valores para las variables locales, es decir, no comparte los valores de esas variables con el SSE que se estaba ejecutando anteriormente. Por tanto, en la reentrada no se dispone del valor de la variable *path* y resulta imposible abrir el archivo de eventos perdidos para leerlo.

De lo anterior se deduce que resulta necesario guardar dicha ruta en una variable externa que pueda ser compartida por diferentes ejecuciones en paralelo del servidor. Esto se consigue empleando a tal efecto un tipo de VI ya existente y que se conoce como Functional Global Variable (FGV). Se trata de un VI que permite o bien modificar el valor de la variable que almacena o bien leerlo. Al emplear FGV es necesario establecer que tipo de variable se va a almacenar, en nuestro se trata de vectores de tipo *path*, el motivo de emplear vectores es que pretende que el servidor pueda atender a varios usuarios a la vez, es por ello que se necesitará almacenar la ruta para cada uno de los usuarios que estén trabajando con el servidor. El contenido de este VI se muestra en la Figura 26.

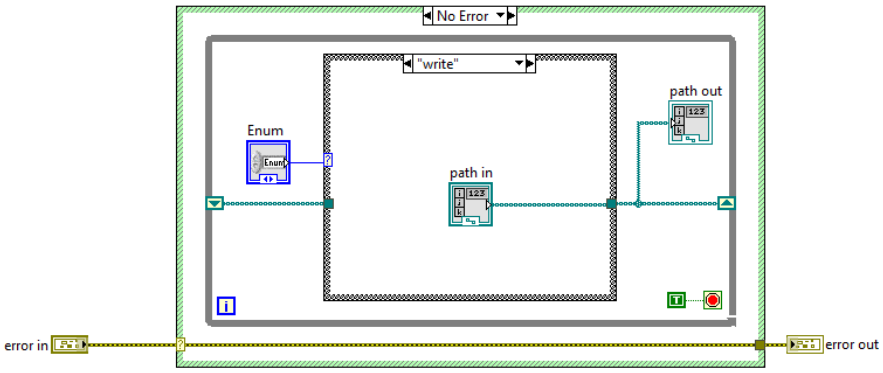

Figura 26. Estructura interna del FGV_Path_Array.vi

Un aspecto importante a destacar al respecto de este tipo de VIs es que se debe seleccionar en sus propiedades la opción de no reentrante, así evitamos que si dos ejecuciones paralelas quieren acceder a la vez al mismo VI se generen clones de dicho VI con valores de variables erróneas. Con esta opción seleccionada lo que sucederá es que primero entrará una de las ejecuciones y cunado esta termine entrará la siguiente, con esto nos aseguramos que el valor de las variables se mantiene correctamente.

Una vez introducidas las modificaciones que acabamos de explicar, el Write_To_File.vi queda como se ve en la Figura 27.

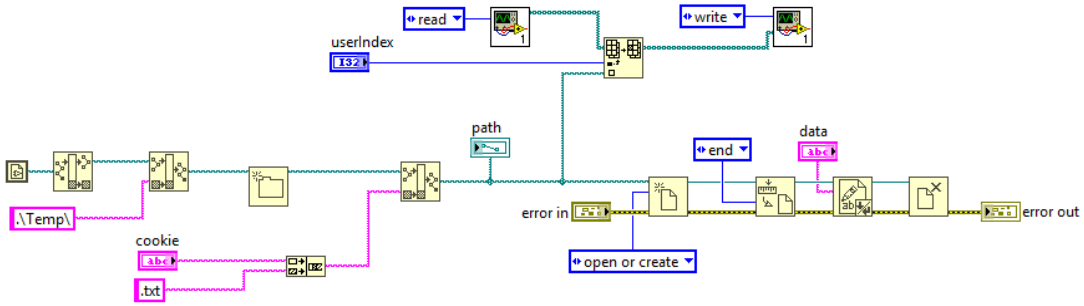

Figura 27. Estructura interna modificada del Write_To_File.vi

Al igual que se almacenan los eventos que se producen durante una desconexión por si se produce una reconexión, también es necesario almacenar los valores de las variables cuando se produjeron estos eventos. De no hacer esto, cuando un usuario se reconectase recibiría en primer lugar todos los eventos guardados en el fichero, pero cuando empezase tomar los valores en tiempo real, lo haría tomando como valor anterior de referencia un cero y no el último valor que generó un evento mientras se esperaba a la reconexión.

Es por ello que se tuvo que crear otro VI del tipo FGV para poder pasar estos valores de una ejecución del servidor a otra, siendo en este caso las variables vectores de vectores de tipo numérico, el nombre de este VI es FGV_Delta_Array.vi. El motivo de emplear vectores de vectores en este caso es, por un lado, el de poder almacenar los valores de tantas variables configuradas para trabajar por eventos como las que hayan sido declaradas como tal en la configuración de la experiencia y por otro poder hacer este guardado para todos los usuarios que estén conectados. Cada usuario puede conectarse a la experiencia en un momento diferente de la misma, con unos valores de variables distintos y, por tanto, desde su percepción, los momentos en los que los valores superan el rango que se ha fijado son distintos.

Otro aspecto del Store_Missed_Events.vi que hubo que modificar fue la condición que hace que deje de ejecutarse. Originalmente incluía únicamente que se produjese un error durante su propia ejecución o que se cumpliese el tiempo de espera de reconexión. No incluía el propio hecho de que se produjese una reconexión, por lo que, aunque esta se produjera se continúa ejecutando el Store_Missed_Events.vi y guardando los eventos en un fichero. Hubo que crear un nuevo VI del tipo FGV que se denominó FGV_Bool_Array.vi que se activa cuando se produce la reconexión de un usuario, esta variable se añadió a las condiciones de parada del Store_Missed_Events.vi consiguiendo así forzar su cierre.

Por último, hay que indicar que de forma esporádica aparecía un problema que relacionado con el nombre del fichero que tenía que generarse. El nombre se basa en una cookie que el servidor manda al usuario. Ocasionalmente la longitud del texto de la cookie era demasiado larga, lo que chocaba con la limitación de caracteres para el nombre de un fichero en Windows. Para resolverlo se optó por truncar la longitud del texto de la cookie a un total de 50 caracteres

El resultado una vez realizadas todas las modificaciones del VI Store_Missed_Events.vi puede verse en la Figura 28.

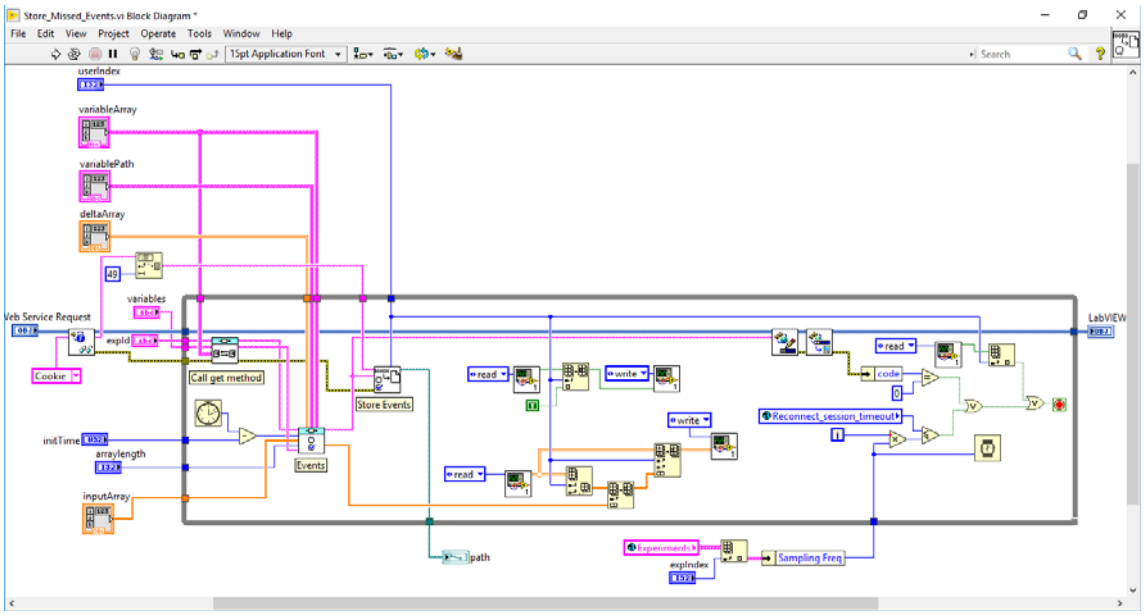

Figura 28. Estructura interna modificada del Store_Missed_Events.vi

2.3.5.- MODIFICACIÓN DEL SERVIDOR SSE

Hasta ahora se han explicado las modificaciones que se han realizado a los diferentes VIs que forman parte del servidor SSE.vi tanto para mejorar las prestaciones de éste, así como para corregir aquellos defectos que se encontraron. Todos estos cambios han hecho necesario reformar el SSE.vi para adaptarlo a las mismas. A continuación, vamos a pasar a explicarlos.

Primero trataremos las variaciones que se han realizado a la parte inicial del código. En el caso de que se trate de una nueva sesión se ha añadido un sub VI llamado Initialization_Of_FGV.vi cuya función, como su nombre indica, es inicializar los valores de las distintas variables FGV que se van a emplear posteriormente. La estructura interna del mismo se puede apreciar en la Figura 29.

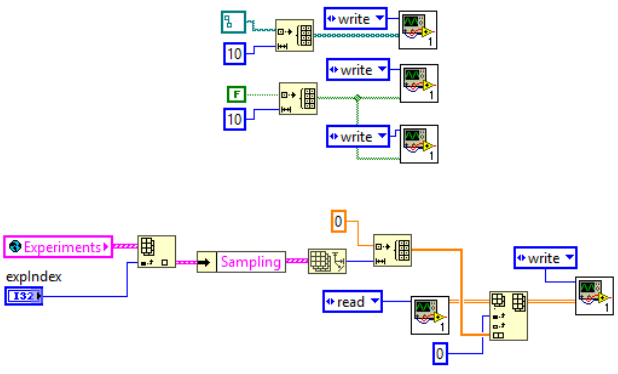

Figura 29. Estructura interna del Initialization_Of_FGV.vi

Muchos de los VIs que hemos creado o modificado hacen uso de una variable de tipo entera denominada *user* o *userIndex*, la cual identifica de forma única a cada uno de los usuarios. Por tanto, cada vez que se establece una nueva sesión hay que asignarle dicho identificador al cliente. Para llevar esto a cabo lo que se hace es crear una nueva variable de sesión llamada *user* y un nuevo VI del tipo FGV llamado FGV_Integer.vi. El valor de este FGV indica el número de usuarios ya conectados a la experiencia, cuando un nuevo cliente accede al servidor comprueba este valor y lo incrementa en una unidad, una vez hecho esto asigna dicho valor a su identificador *user*.

Las modificaciones del SSE.vi comentadas hasta ahora pueden observarse en la Figura 30.

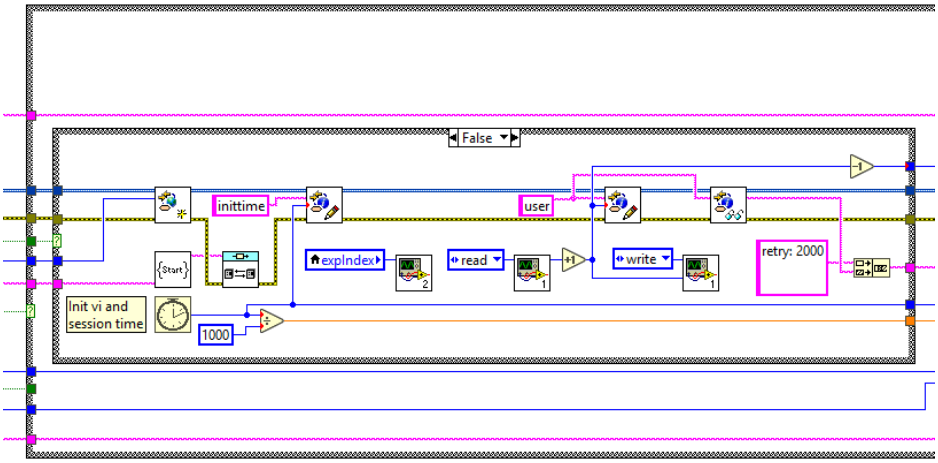

Figura 30. Estructura interna modificada del bucle condicional inicial de SSE en caso de nuevas sesiones.

Siguiendo en la parte inicial del código del SSE.vi, vamos a explicar las modificaciones introducidas para el caso de que se trate de una petición de conexión proveniente de una sesión ya existente. Lo primero que hemos añadido es una parte que se encarga de obtener el valor de su variable de sesión *user*. Además, se han creado dos nuevos VIs.

El primero es el llamado Reconnection_Update_Of_FGV.vi, cuya función es poner a verdadero el correspondiente valor del FGV_Bool_Array. La Figura 31 presenta la estructura de este VI.

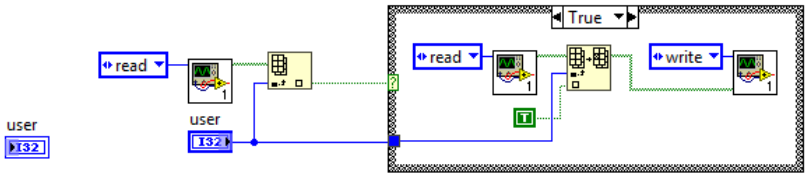

Figura 31. Estructura interna del Reconnection_Update_Of_FGV.vi

El segundo VI que se creó fue el Read_From_File.vi, cuyo objetivo es leer el fichero de eventos que ha estado guardando el Store_Missed_Events y presentárselo al cliente. Su estructura se muestra en la Figura 32.

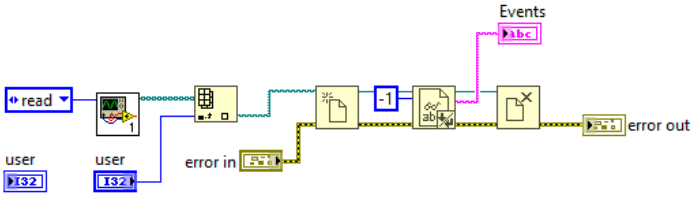

Figura 32. Estructura interna del Read_From_File.vi

La programación del bucle condicional del SSE.vi en el caso de tratarse de una conexión ya existente queda como puede verse en la Figura 33.

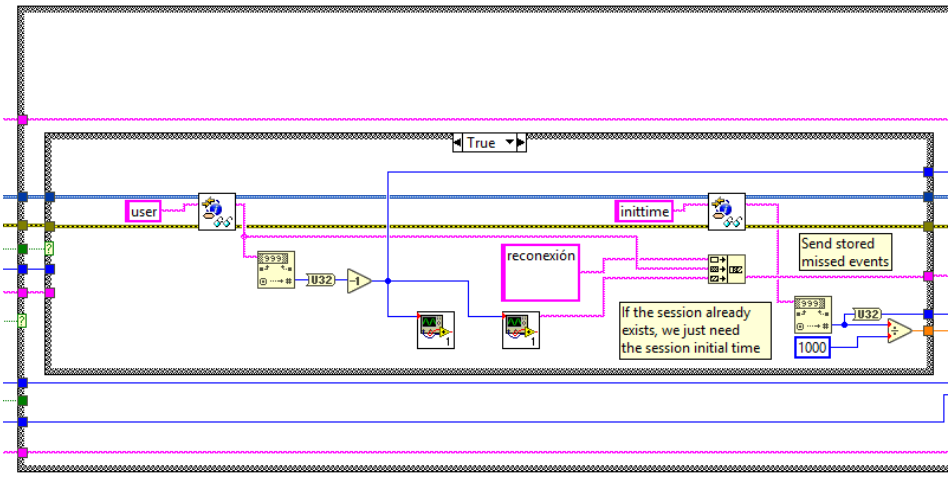

Figura 33. Estructura interna modificada del bucle condicional inicial de SSE en caso de sesiones ya existentes.

Pasamos ahora a explicar los cambios introducidos en la segunda sección del código. Originalmente se pasaba directamente a presentar los datos los eventos que se

produjeran en tiempo real, sin embargo, ahora debemos realizar algunas tareas previas antes de llegar a este punto.

En primer lugar, nos aseguramos de eliminar el fichero de eventos que pudiera existir, ya que estos ya habrán sido entregados al cliente. Es en este mismo momento cuando se llama al VI Obtain_Variables_and_Deltas.vi, cuya función ya ha sido explicada anteriormente.

Seguidamente se hace una llamada a un nuevo VI denominado Delta_Array_Update.vi. Como su nombre indica, el objetivo de este VI es actualizar los valores que deben tener las variables del subvector del FGV_Delta_Array.vi correspondiente al usuario. Su comportamiento varía en función de que se trate de una reconexión, donde se deben mantener los valores provenientes de la anterior conexión, o una conexión nueva, donde todos los valores se deben poner a cero inicialmente. Para determinar si estamos ante una reconexión o no se hace uso del FGV_Bool_Array.vi. La estructura de este nuevo VI puede observarse en la Figura 34.

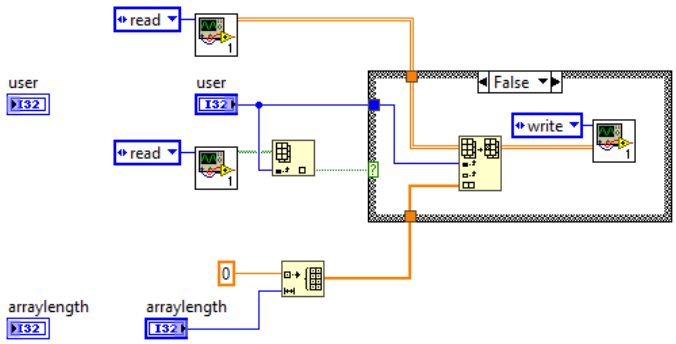

Figura 34. Estructura interna del Delta_Array_Update.vi

Una vez realizada la actualización de los valores del FGV_Delta_Array.vi únicamente tenemos que tomar dichos valores para pasárselos al Events_Composer.vi, de forma que pueda empezar a detectar eventos en tiempo real a partir del último valor que tuviera anteriormente. El conjunto de modificaciones que acabamos de describir se muestra en la Figura 35.

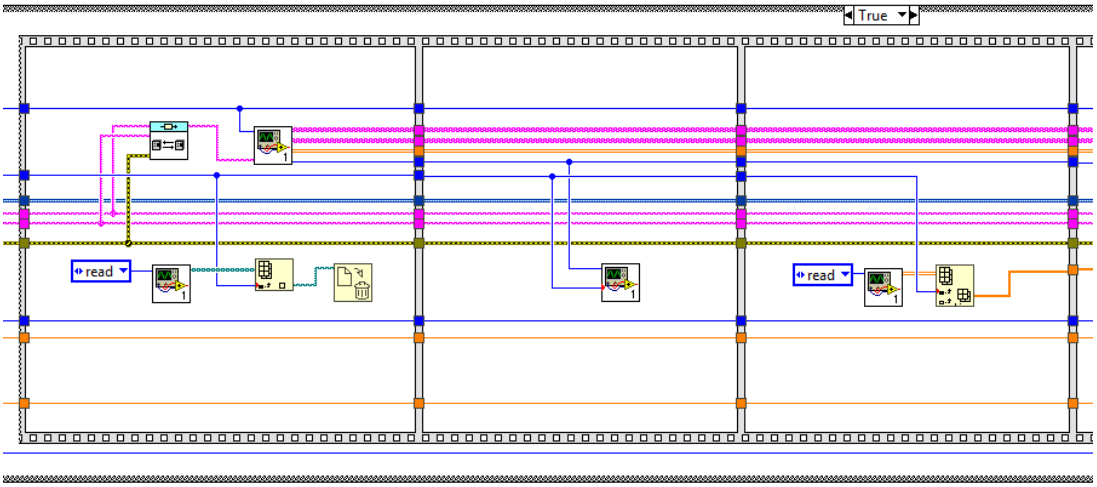

Figura 35. Nuevos pasos añadidos a la estructura secuencial de SSE

Los últimos cambios introducidos se realizaron en la tercera sección del código, concretamente en la parte se ejecuta una vez se ha cumplido el tiempo de espera de reconexión o si se produce la reconexión. Aquí se ha insertado un nuevo VI al que se ha llamado Delete_Stored_Events.vi. Cuando nos encontramos en este paso es porque se está a punto de cerrar la llamada al servidor SSE. Por tanto, este VI reinicia los valores de aquellas variables FGV que se han empleado durante la ejecución para que puedan ser reutilizadas con posterioridad. La estructura de este VI se presenta en la Figura 36.

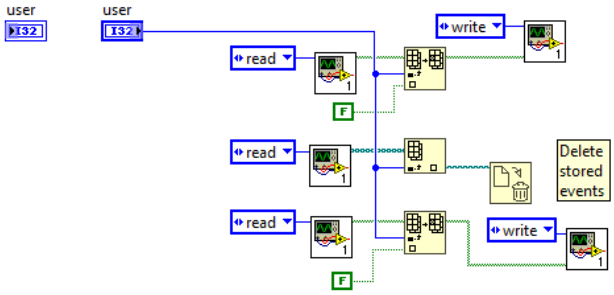

Figura 36. Estructura interna del Delete_Stored_Events.vi

Finalmente, la Figura 37 muestra las modificaciones de esta tercera sección del código del SSE.vi.

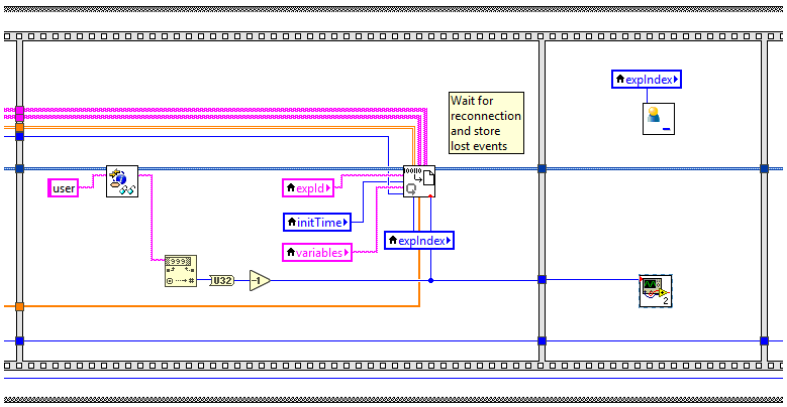

Figura 37. Estructura de la tercera sección de código del SSE modificado

3.- PRUEBAS Y VALIDACIÓN

En esta sección vamos a comprobar el funcionamiento de nuestro servidor de eventos. Para ello, emplearemos un laboratorio existente que proporciona el propio LabVIEW como ejemplo y que se llama *SimEx DC Motor Position Control with PID.vi*. Se trata de un VI en el que se simula un motor DC que se introduce en un lazo de realimentación junto con un controlador PID para hacer un control de posición con una consigna de la posición en la que se quiere que esté, el VI incluye la posibilidad de introducir perturbaciones que afecten al motor.

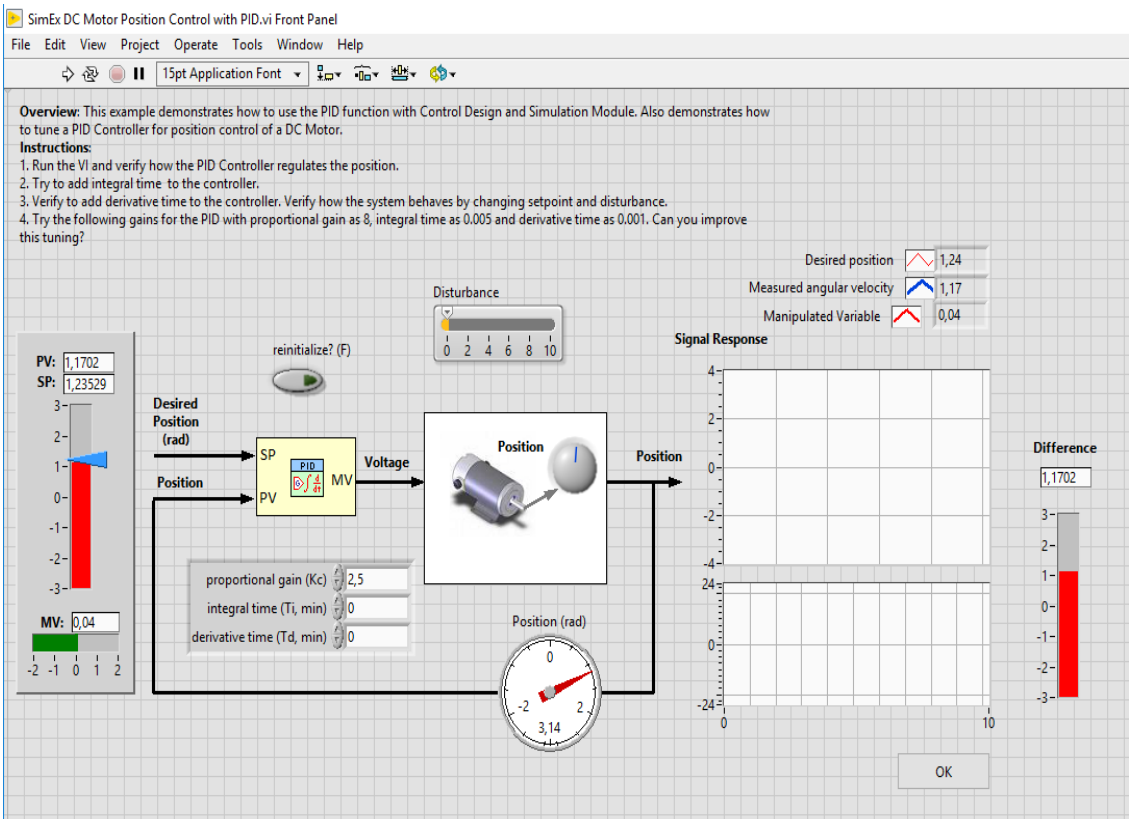

Figura 38. Panel Frontal del SimEx DC Motor Position Control with PID.vi

Como se puede observar en la Figura 38, el laboratorio dispone de cuatro entradas de señal. La primera de ellas (PV) indica la posición actual del motor, mientras que la segunda (SP) es la consigna de la posición que se quiere que alcance el motor. La entrada reinitialize permite reiniciar los valores del laboratorio para volver a empezar desde el punto inicial. Por último, la señal Disturbance permite simular perturbaciones externas que puedan afectar al comportamiento del motor DC. Desde el panel frontal de la experiencia también se tiene acceso a los valores proporcional, integral y derivativo del controlador PID.

Empezaremos realizando simulaciones empleando el servidor RIP original y que sirvió de base para el desarrollo de este trabajo, de forma que podamos ver cuál es su respuesta y poder contrastarla con la respuesta del nuevo servidor.

3.1.- PRUEBAS DEL SERVIDOR ORIGINAL

En esta sección vamos a estudiar las capacidades con las que cuenta el servidor del que partimos, de modo que se pueda comparar con las capacidades del servidor modificado.

3.1.1.- FRECUENCIA DE MUESTREO

3.1.1.1.- PROPÓSITO DE LA PRUEBA

El propósito de esta prueba es comprobar que el servidor original no es capaz de usar distintas frecuencias de muestreo para distintas experiencias, empleando siempre una misma frecuencia para todas.

3.1.1.2.- ACCIÓN A REALIZAR

En el Global_Configurations.vi se configurarán dos experiencias, cada una con un valor de frecuencia de muestreo distinto y a su vez distinto del valor de la frecuencia general que emplea el SSE, cuyo valor se fija en la variable global SSE_frequency y que por defecto tiene un valor de 200 ms.

En la Figura 39 se muestran las dos experiencias que vamos a utilizar, cuyos nombres son TestNO_STOP y DCMotor. A la primera se le ha asignado una frecuencia de muestreo de 500 ms, para la segunda el valor es de 300 ms.

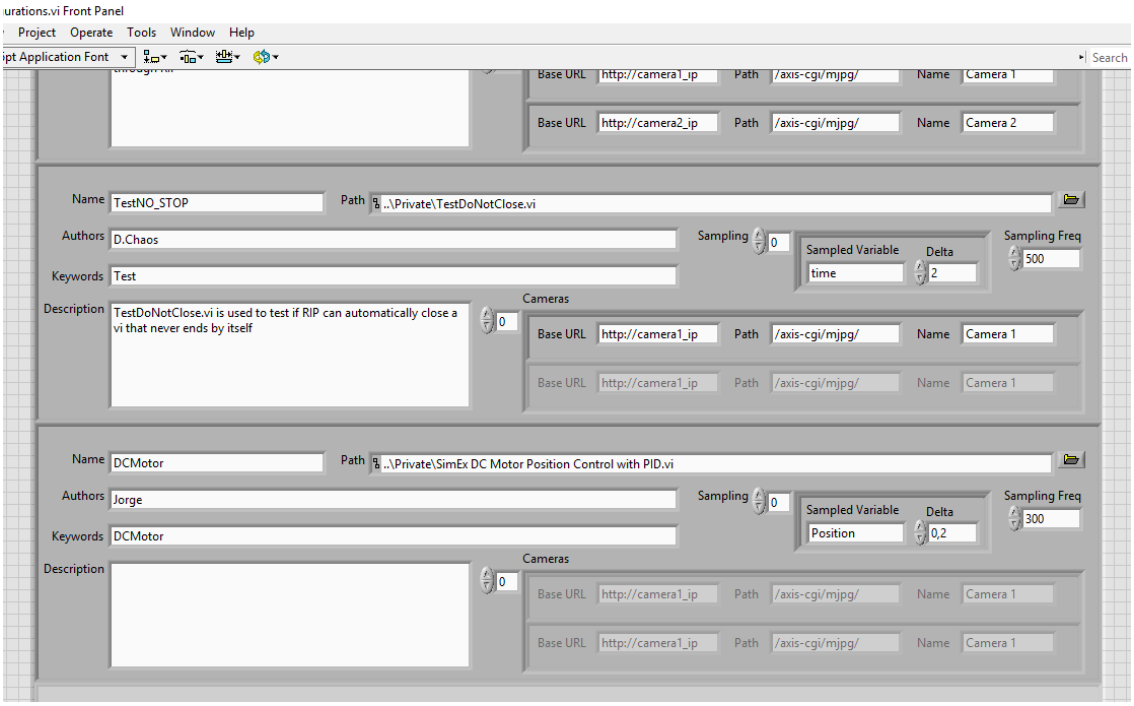

Figura 39. Aspecto del Global_Configurations.vi para las pruebas de frecuencia de muestreo

3.1.1.3.- RESULTADO ESPERADO

Lo siguiente es ejecutar el servidor y conectarnos como cliente a ambas experiencias. Lo que se espera es que la frecuencia entre los mensajes enviados por el servidor no se corresponda con los valores que hemos definido para cada una de las experiencias, sino con el valor de 200 ms de la SSE_frequency.

Las Figuras 40 y 41 nos proporcionan las respuestas del servidor, primero cuando se llama a la experiencia TestNO_STOP, y segundo a la experiencia DCMotor. Como se puede apreciar, en ambos casos la frecuencia con la que se están entregando los eventos al cliente es de 200 ms, por lo se confirma que el servidor emplea la misma frecuencia de muestreo para todas las experiencias independientemente de que éstas tengan definidos valores distintos.

Por tanto, la única forma en la que el servidor en su estado inicial pueda emplear diferentes frecuencias de muestreo, es variando el valor de la SSE_frequency pero dichas frecuencias se aplican por igual a todas las experiencias que encuentren definidas en el mismo.

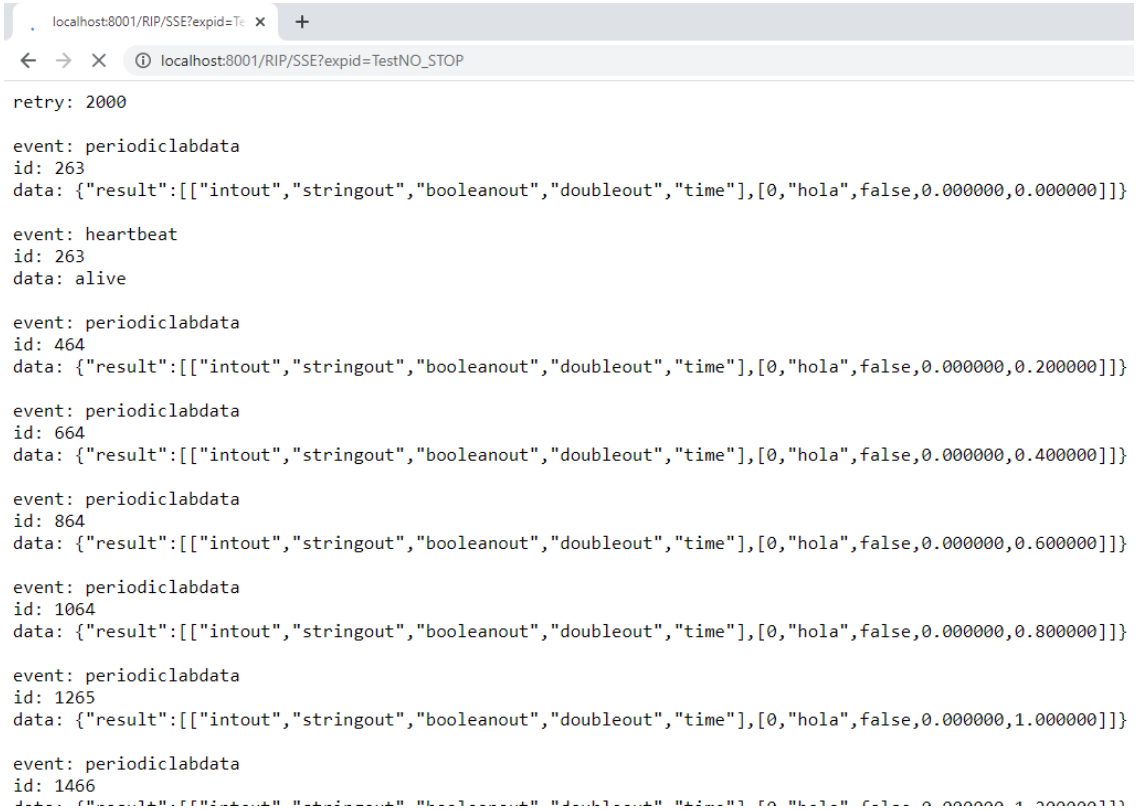

Figura 40. Respuesta entregada al cliente para la experiencia TestNO_STOP

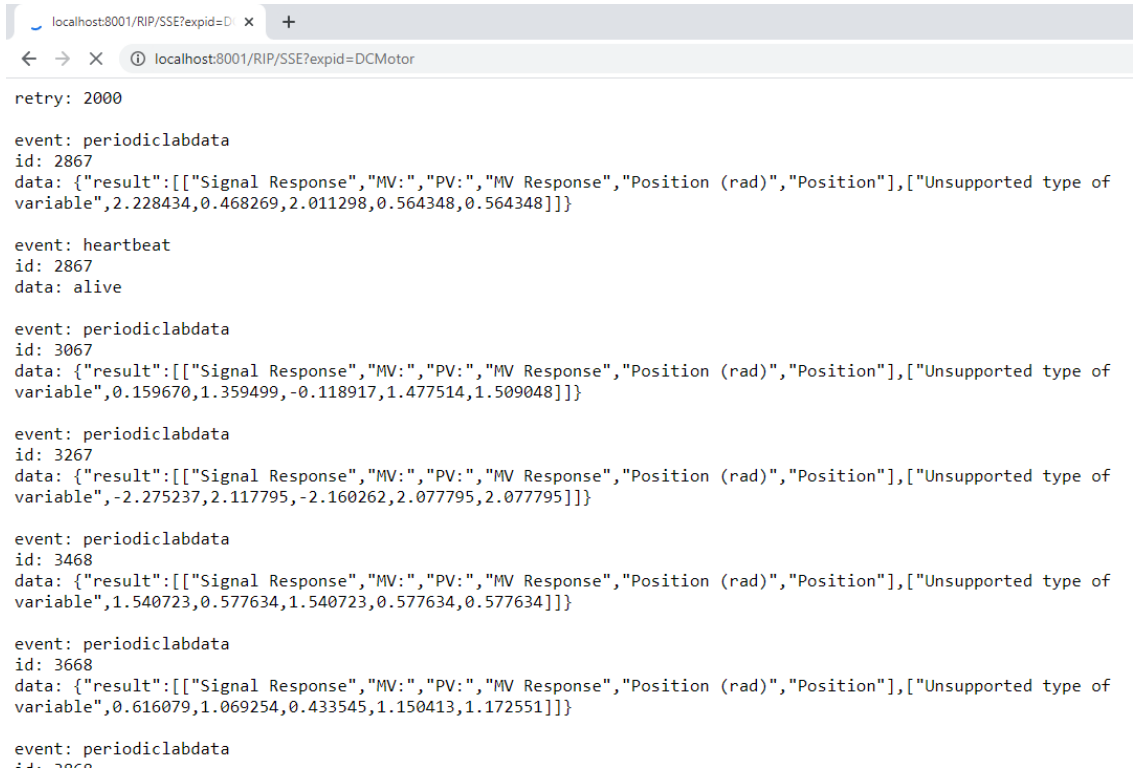

Figura 41. Respuesta entregada al cliente para la experiencia DCMotor

3.1.2.- DETECCIÓN DE EVENTOS

3.1.2.1.- PROPÓSITO DE LA PRUEBA

El propósito de esta prueba es comprobar que el servidor original no es capaz de detectar eventos de una variable de una experiencia configurada como tal.

3.1.2.2.- ACCIÓN A REALIZAR

En el Global_Configurations.vi vamos a la parte en la que se ha configurado la experiencia DCMotor y configuramos la variable *Position*, la cual nos proporciona la posición del motor, como una variable por eventos, con un valor de su delta de 0,2. Si revisamos la Figura 39 se puede verificar que se ha llevado a cabo dicha configuración.

3.1.2.3.- RESULTADO ESPERADO

El siguiente paso es ejecutar el servidor y conectarnos como clientes a la experiencia DCMotor. Lo que se espera es que no se produzca ninguna respuesta debida a un evento de la variable *Position* aun cuando su valor haya experimentado una variación superior a la delta establecida.

Repasando la Figura 41 comprobamos a pesar de que el valor de la variable experimenta variaciones superiores a la de delta, el cliente nunca recibe información de un evento.

3.1.3.- RECUPERACIÓN DE EVENTOS PERDIDOS

3.1.3.1.- PROPÓSITO DE LA PRUEBA

La última de las capacidades que queremos comprobar del servidor original, es la de recuperar eventos perdidos tras perder un cliente una conexión y reconectarse más tarde, siempre dentro del tiempo límite establecido para ello.

3.1.3.2.- ACCIÓN A REALIZAR

Para demostrarlo hemos realizado una desconexión del servidor a los 81 segundos y nos hemos vuelto a conectar a los 94 segundos. Lo que se espera con esto es ver como el cliente recibe todos los eventos periódicos que se produjeron durante todo ese tiempo que estuvo desconectado, antes de empezar a recibir los datos de la conexión actual. Para facilitar la comprensión de la prueba que vamos a llevar a cabo en este apartado hemos

añadido la línea temporal que aparece en la Figura 42, donde se indican las diferentes acciones que vamos a realizar durante la prueba y las consecuencias de las mismas.

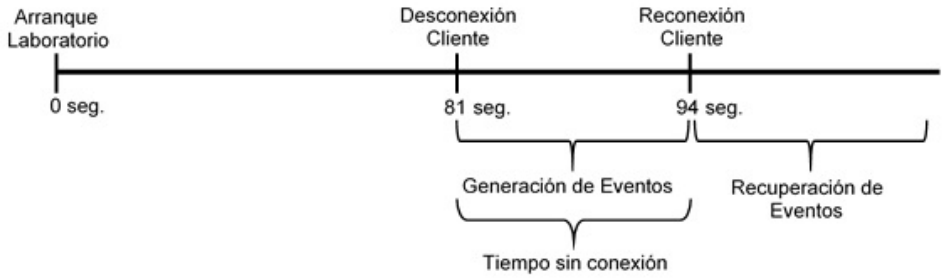

Figura 42. Línea temporal de acciones y consecuencias en el servidor original

3.1.3.3.- RESULTADO ESPERADO

El resultado que se espera es que tras la reconexión se produzca la entrega al cliente de los eventos periódicos que se han producido entre los 81 y 94 segundos para luego seguir proporcionando información en tiempo real. Atendiendo al resultado que se muestra en la Figura 43 se puede ver que el servidor comienza a darnos los eventos a partir del momento en el que nos reconectamos, perdiendo los eventos que se produjeron durante la desconexión. Como ya se dijo en el apartado *Recuperación de Eventos* dentro del apartado *Trabajo Desarrollado*, el servidor original había empezado a trabajar en ello, pero todavía no era capaz de hacerlo.

Figura 43. Respuesta a la vuelta de una desconexión del servidor original

3.2.- PRUEBAS DEL SERVIDOR MODIFICADO

Pasamos ahora a presentar los resultados obtenidos con el servidor una vez se han introducido todos los cambios que se explicaron en la parte de Trabajo Desarrollado, para poder comparar ambas versiones del servidor, hemos configurado inicialmente el Global_Configurations.vi de la nueva versión igual que el de la original.

3.2.1.- FRECUENCIA DE MUESTREO

El propósito de esta prueba es comprobar que el servidor original no es capaz de usar distintas frecuencias de muestreo para distintas experiencias, empleando siempre una misma frecuencia para todas.

3.2.1.1.- PROPÓSITO DE LA PRUEBA

El objetivo es comprobar es si el servidor es capaz de emplear distintas frecuencias de muestreo para distintas experiencias. Para cada experiencia deberá emplear la frecuencia que se haya establecido en su variable global *Sampling Freq*.

3.2.1.2.- ACCIÓN A REALIZAR

Las acciones a realizar coinciden con las explicadas en el apartado 5.1.1.2, configurando la frecuencia de muestreo de las experiencias TestNO_STOP y DCMotor. A la primera se le ha asignado una frecuencia de muestreo de 500 ms, para la segunda el valor es de 300 ms.

3.2.1.3.- RESULTADO ESPERADO

Tras ejecutar el servidor y conectarnos como cliente a ambas experiencias. Lo que se espera es que la frecuencia entre los mensajes enviados por el servidor sea distinta para cada una de las experiencias y que coincida en cada caso con el valor que se ha fijado en la *Sampling Freq*.

En las Figuras 44 y 45 se aprecia como la frecuencia de las respuestas para cada experiencia coinciden con las frecuencias definidas es sus variables *Sampling Freq*, por lo se confirma que ahora el servidor es capaz de emplear frecuencias de muestreo independientes para todas las experiencias.

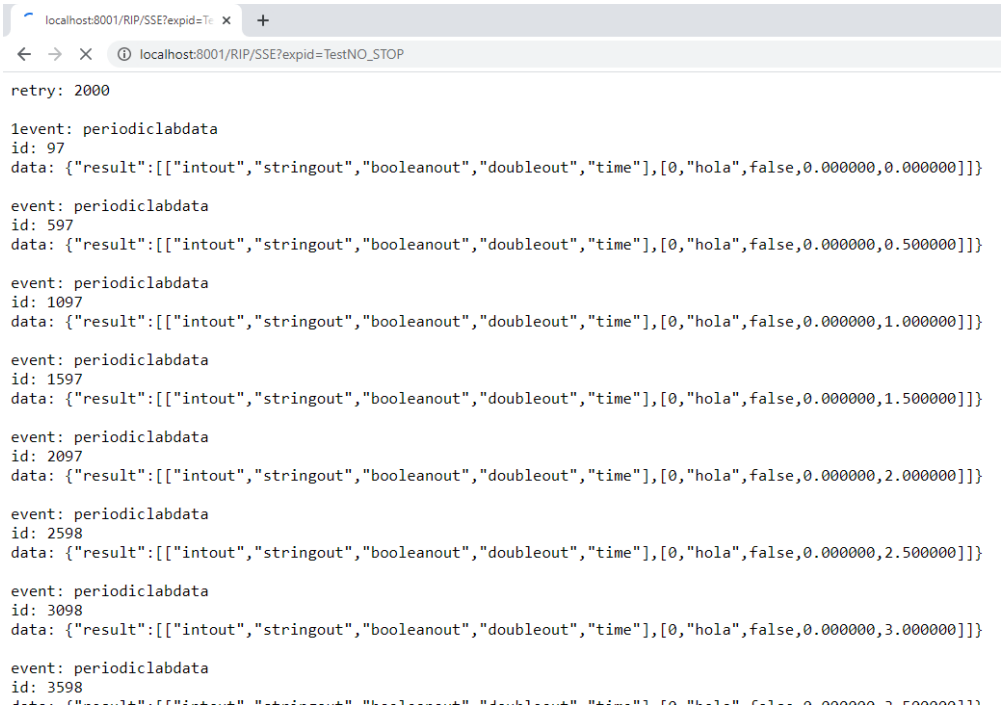

Figura 44. Respuesta entregada al cliente del servidor modificado para la experiencia TestNO_STOP

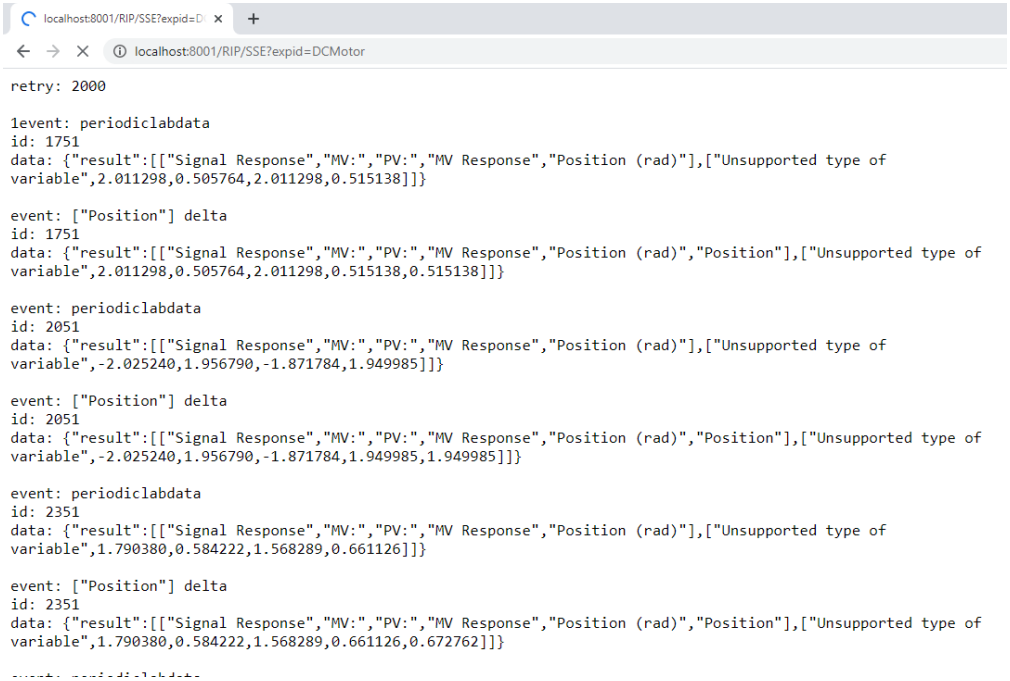

Figura 45. Respuesta entregada al cliente del servidor modificado para la experiencia DCMotor

3.2.2.- DETECCIÓN DE EVENTOS

3.2.2.1.- PROPÓSITO DE LA PRUEBA

Lo siguiente que vamos a estudiar es si el nuevo servidor es capaz de detectar los eventos de una variable cuando su valor varíe en más de lo establecido en su delta y enviar al cliente dicha información.

3.2.2.2.- ACCIÓN A REALIZAR

La acción es idéntica a la explicada en el apartado 5.1.1.2.

3.2.2.3.- RESULTADO ESPERADO

Repasando la Figura 45 observamos como ahora, además de las respuestas periódicas, el cliente recibe eventos debidos a la variación en más de 0,2 unidades del valor de la variable *Position*. Con esto queda demostrado como ahora el servidor sí es capaz de detectar variaciones superiores a la delta establecida para una variable y proporcionan los correspondientes eventos asociados a dichas variaciones.

3.2.3.- DETECCIÓN DE MÚLTIPLES EVENTOS

3.2.3.1.- PROPÓSITO DE LA PRUEBA

En este apartado se quiere comprobar si el servidor es capaz de detectar eventos de más de una variable, cada una de ella además con su propio valor de delta.

3.2.3.2.- ACCIÓN A REALIZAR

El laboratorio que estamos utilizando cuenta con una variable denominada *Difference* que representa la diferencia entre el valor de la señal de posición de motor (PV) y la consigna (SP). Lo que hacemos es definir la variable *Difference* como por eventos, estableciendo el valor de su delta en 0,5. La experiencia DCMotor cuenta así con dos variables definidas por eventos *Position* y *Difference*. La configuración de esta última se presenta en la Figura 46.

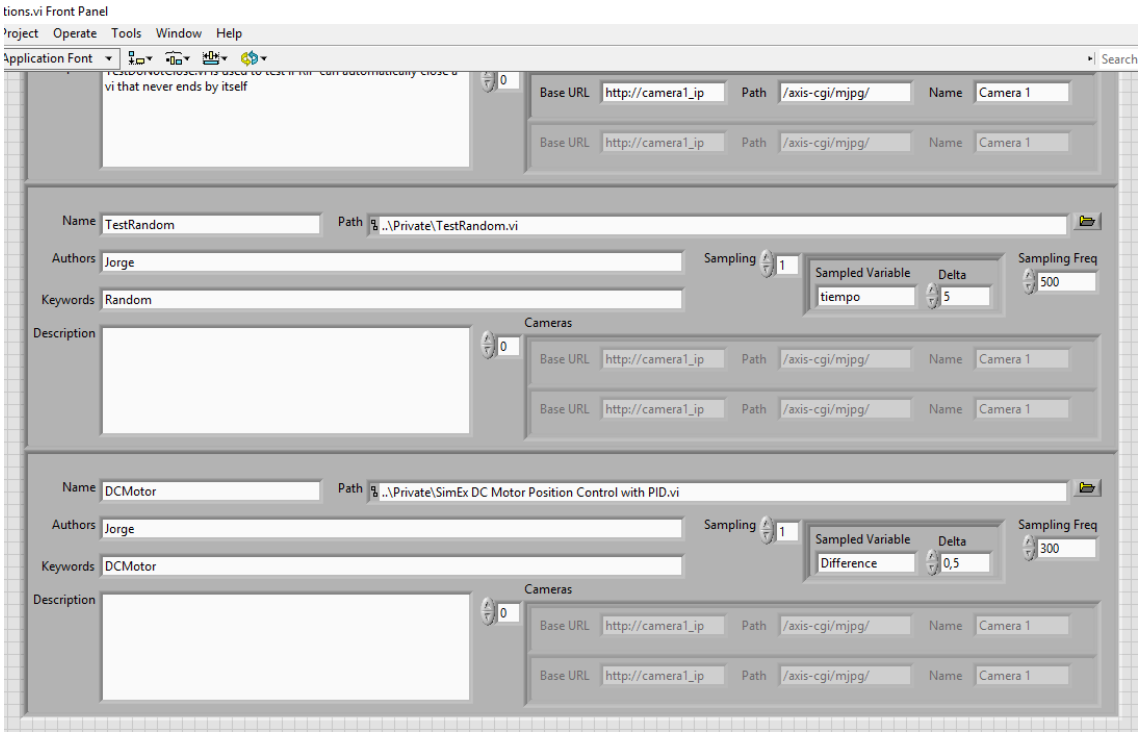

Figura 46. Declaración de la variable *Difference*

3.2.3.3.- RESULTADO ESPERADO

Arrancamos el servidor, nos conectarnos al experimento y dejamos pasar unos instantes hasta que el sistema se estabiliza. En ese momento la diferencia es cero y el valor del setpoint es de 1,23259. Lo que hacemos entonces modificar este valor en el slider o mediante el campo de entrada para ir introduciendo diferencias mayores de 0,5.

En la Figura 47 podemos ver la respuesta que entrega el servidor tras modificar varias veces el valor del setpoint. En el tiempo 2646671 ms se puede apreciar cómo se acaba de producir una variación del setpoint que ha hecho que la diferencia sea mayor de 0,5, en concreto es una variación de 1,17. Esto ha provocado también una reacción del PID, lo que ha conllevado un cambio rápido en la posición del motor, que ha pasado de estar en -0.301 a estar en -2.112. Ambos eventos han sido identificados por el servidor y presentados al cliente con la indicación correspondiente.

Posteriormente se siguen produciendo eventos por posición ya que este valor está cambiando para acercarse al solicitado. Seguimos viendo un valor de diferencia muy grande pero ese valor no experimenta modificaciones de más de 0,5 de una iteración a la siguiente, es por ello que ya no aparece eventos debidos a esta variable.

Figura 47. Respuesta entregada tras variar el valor del setpoint

3.2.4.- MEJORA DE LA INFORMACIÓN DE LA RESPUESTA

3.2.4.1.- PROPÓSITO DE LA PRUEBA

En este apartado vamos a comprobar si se ha conseguido mejorar la información que se entrega al cliente.

3.2.4.2.- ACCIÓN A REALIZAR

Las únicas acciones que necesitamos realizar son ejecutar el servidor y conectarnos a la experiencia DCMotor.

3.2.4.3.- RESULTADO ESPERADO

Lo que se espera es que las respuestas periódicas presenten datos únicamente de las variables periódicas, y las respuestas por eventos proporcionen información al respecto de que variable ha experimentado un evento, así como el valor de todas las variables de la experiencia cuando el evento se ha producido.

Si revisamos la propia Figura 47 podemos apreciar las diferencias entre las respuestas periódicas y por eventos. En el caso de las respuestas periódicas no se nos proporciona información acerca de las variables *Position* y *Difference* ya que éstas están configuradas por eventos. En el caso de una respuesta debida a un evento lo primero que se nos indica es la variable o variables que han experimentado un evento y luego se nos

proporcionan los valores de todas las variables de la experiencia. Teniendo en cuenta lo anterior podemos decir que el resultado se corresponde con lo esperado.

3.2.5.- RECUPERACIÓN DE EVENTOS PERDIDOS

3.2.5.1.- PROPÓSITO DE LA PRUEBA

La última de las capacidades que vamos a comprobar del nuevo servidor, es la de la entrega de eventos perdidos después de que un cliente haya perdido la conexión, pero logre reconectarse dentro del tiempo límite establecido para ello.

3.2.5.2.- ACCIÓN A REALIZAR

En primer lugar, vamos a arrancar el servidor y conectarnos como cliente a la experiencia DCMotor. Dejamos que el motor alcance una posición estable y con una diferencia cero respecto al set point. Entonces, cuando el tiempo es de aproximadamente 36 segundos desconectamos al cliente. Seguidamente nos mantenemos a la espera durante unos nueve segundos, durante los cuales el servidor debería almacenar los eventos periódicos que se generen. Pasado este tiempo introducimos perturbaciones en el laboratorio, lo que conlleva la aparición de una serie de eventos de la variable Position. Por último, nos reconectaremos a los 68 segundos.

Para facilitar la comprensión de la prueba que vamos a llevar a cabo en este apartado hemos añadido la línea temporal que aparece en la Figura 48, donde se indican las diferentes acciones que vamos a realizar durante la prueba y las consecuencias de las mismas.

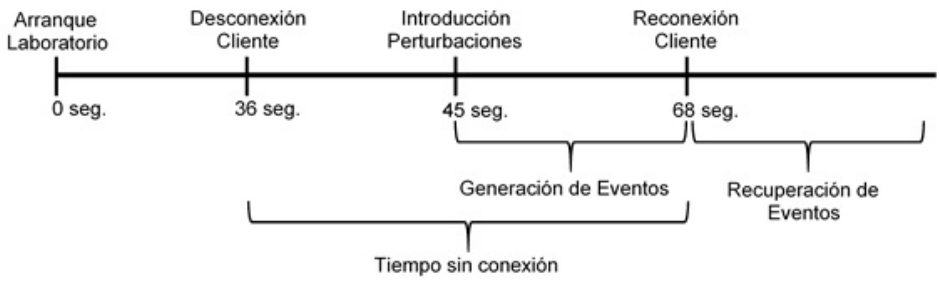

Figura 48. Línea temporal de acciones y consecuencias

3.2.5.3.- RESULTADO ESPERADO

El resultado que se espera es que tras la reconexión se produzca la entrega al cliente de los eventos periódicos que se han producido entre los 36 y 68 segundos para luego seguir proporcionando información en tiempo real.

Si observamos la Figura 49 vemos que primero se nos informa de que se ha producido una reconexión y seguidamente se nos empieza a proporcionar las respuestas periódicas que generó el servidor a partir de los 37 segundos.

Figura 49. Respuesta entregada al cliente justo tras reconectar

En la Figura 50 vemos como, a partir de los 45 segundos se nos entregan también los eventos de posición que se produjeron.

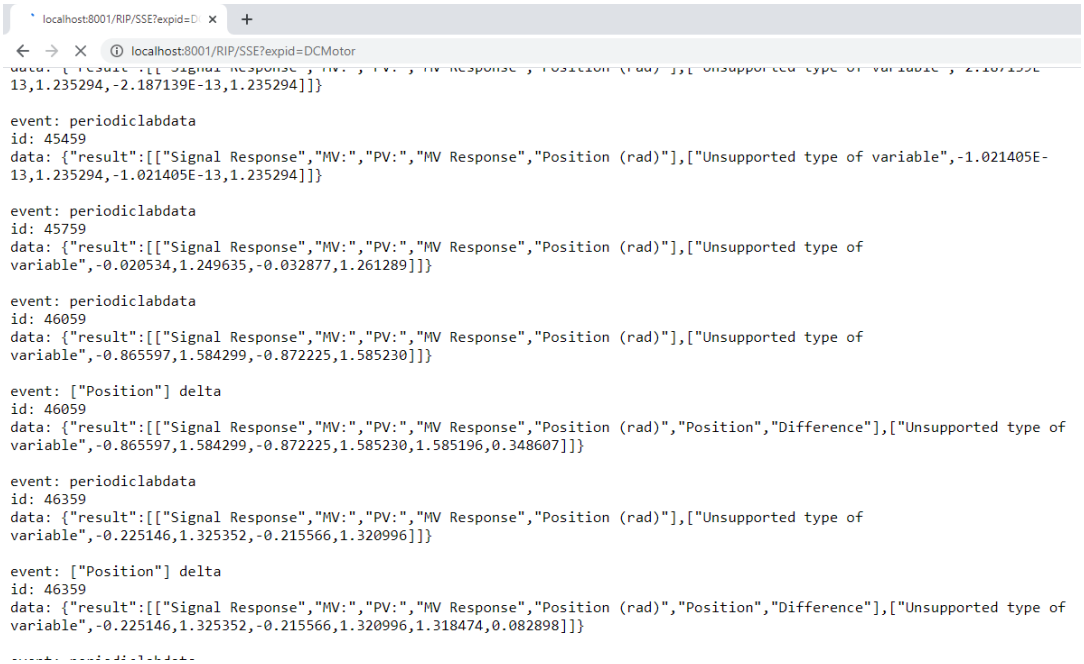

Figura 50. Respuesta entregada al cliente con eventos producidos mientras estaba desconectado

Por tanto, se confirma que el nuevo servidor es capaz de almacenar los eventos que se producen durante la desconexión de un cliente y entregárselos tras su reconexión.

3.3.- ANÁLISIS DE LAS PRUEBAS

En base a los resultados de las pruebas aquí presentadas, podemos afirmar que se ha conseguido una mejora del servidor. No solo se han desarrollado capacidades que hasta ahora no existían o estaban incompletas, sino que también se han mejorado otras ya existentes.

4.- CONCLUSIONES

El objetivo principal de este trabajo era la de añadir soporte a la detección de eventos a la implementación de RIP server en LabVIEW ya existente, para permitir que realice un envío de datos no necesariamente periódico, sino en función de los cambios que puedan ocurrir en las variables de interés; por ejemplo, cuando la diferencia entre el valor actual y el último enviado supera un cierto umbral.

De esta manera, se consigue proporcionar la detección de eventos como una característica nativa del servidor y, por ende, disponible para aplicar a cualquier laboratorio sin necesidad de tener que llevar a cabo una implementación específica, sino sólo configurar en base a las necesidades del muestreo.

Todo ello permite la incorporación de los métodos de control basado en eventos al marco de desarrollo de laboratorios remotos para la enseñanza de Control Automático, de forma que los alumnos puedan trabajar y familiarizarse con ellos.

Otro propósito que se tenía con este trabajo era extender algunas de las capacidades con las que ya contaba el propio servidor, como la frecuencia de muestreo y la delta de generación de eventos. Respecto a la frecuencia de muestreo, originalmente el servidor contaba con una que era la misma para todas las experiencias, tras el desarrollo que hemos realizado ahora es posible definir una frecuencia diferente para cada experiencia.

En lo referente al umbral delta de generación de eventos para el mecanismo de detección por cruce de nivel, nos encontrábamos con una situación parecida, sólo se podía fijar un valor de delta que era el mismo para todas las variables en todas las experiencias. Ahora es posible definir un valor de delta distinto para cada una de las variables de una experiencia.

Se ha desarrollado también un componente que permite que un cliente que ha sufrido una desconexión y se reconecta dentro de un tiempo establecido pueda recibir los eventos que se ha perdido durante ese tiempo. Los eventos se almacenan en un fichero temporal y al producirse la reconexión, aprovechando el uso del protocolo SSE que realiza el RIP, el cliente recibe los eventos perdidos ordenados en el tiempo y con el mismo formato que los eventos que recibe cuando está conectado. Esta nueva capacidad que pérdidas

temporales de conexión no supongan la pérdida del trabajo realizado, el cual se puede retomar tras la reconexión.

En el apartado “Pruebas y validación” se recogen las pruebas realizadas para validar el correcto funcionamiento del servidor. Tal y como se puede ver el dicho apartado, se han mejorado aspectos tales como la frecuencia que emplea el servidor, la detección de eventos de una variable y la información que proporciona una respuesta a un evento. También se ha profundizado en aspectos en los que el servidor original sólo disponía de una implementación parcial tales como la detección de eventos de múltiples variables y la recuperación de eventos perdidos.

La entrega de información tanto periódica como por eventos que se lleva a cabo permite al alumno comparar ambas formas de control. Así pueden apreciar algunas de las características que se comentaron en la introducción de este trabajo. En especial se puede observar el gran flujo de información que se genera en el caso de muestreo periódico y la diferencia en el caso de muestreo por eventos. Lo anterior sirve también para verificar la menor exigencia a la que se somete a la CPU en el caso de control por eventos.

4.1.- LÍNEAS FUTURAS

En el trabajo aquí desarrollado nos hemos centrado en trabajar con laboratorios virtuales, por lo que un aspecto en el que se podría trabajar en el futuro es el desarrollo de la parte de comunicación con hardware real que permita la interacción con laboratorios remotos.

Otro aspecto cuyo desarrollo podría ser interesante tendría que ver con la detección de eventos y el acceso a un mismo laboratorio de varios usuarios a la vez. Actualmente si varios usuarios acceden a un mismo laboratorio cada uno va a recibir los eventos en base al momento en el que se conectan al mismo por primera vez y el estado de laboratorio en ese instante, por lo que los eventos no tienen por qué ser los mismos para todos los usuarios. Es justamente este aspecto el que se podría modificar de forma que todos los usuarios recibieran exactamente los mismos eventos independientemente del momento en que se hayan conectado al experimento.

Ahondando en el tema de los eventos que recibe cada usuario conectado a un laboratorio, ahora mismo el servidor permite establecer un valor de delta para una de las variables de una experiencia que hayan sido configuradas como eventos y dichos valores

pueden ser modificados sin problema. Si embargo, ese valor de delta que se ha fijado para una variable es el mismo para todos los usuarios que se conectan a una aplicación. Se podría valorar la posibilidad de que cada usuario pudiese fijar el valor de delta para cada variable, en función de lo que a ese usuario le interese.

Por último, otra posible mejora que podía añadirse en el futuro tiene que ver con la respuesta que proporciona el servidor al producirse una reconexión. Si bien se indica al cliente que se ha producido una reconexión, seguidamente se empiezan a presentar los eventos perdidos y luego los actuales, pero sin ningún tipo de distinción entre unos y otros, quizá podría añadirse algún mecanismo para indicar en qué punto se han dejado de dar eventos perdidos y se está empezando a ofrecer los datos actuales.

5.- REFERENCIAS Y BIBLIOGRAFÍA

[1] Åström, K. J. y B. Wittenmark (1997). *Computer controlled systems: Theory and design*. Third Edition. Prentice Hall.

[2] Dorf, R.C., M. C. Farren, C. A. Phillips (1962). *Adaptive sampling for sampled-data control systems*. IEEE Transactions on Automatic Control, 7 (1), 38-47.

[3] Hsia, T. C. (1974). *Analytic design of adaptive sampling control laws*. IEEE Transactions on Automatic Control, 19 (1), 39-42.

[4] Årzén, K. J. (1999). *A Simple event-based PID controller*. Proceedings of 14th IFAC World Congress. vol. 18. Beijing, China. 423-428.

[5] Åström, K. J. and B. M. Bernhardsson (1999). *Comparison of periodic and event based sampling for first order stochastic systems*. Proceedings of the 14th IFAC World Congress, Beijing, China. 11, 301-306.

[6] Aranda-Escolástico, E., Guinaldo, M., Heradio, R., Chacón, J., Vargas, H., Sánchez, J. and Dormido, S. (2020). *Event-Based Control: A Bibliometric Analysis of Twenty Years of Research*. IEEE Access, vol. 8, pp. 47188-47208.

[7] Dormido, S., Sánchez, J., Kofman, E. (2008). *Muestreo, control y comunicación basados en eventos*. Revista Iberoamericana de Automática e Informática Industrial, 5(1), 5-26.

[8] de la Torre, L., Chacón, J., Chaos, D., Dormido, S., Sánchez, J. (2019). *Using Server-Sent Events for Event-Based Control in Networked Control Systems*. IFAC-PapersOnLine, 52(9), 260-265.

[9] Liu, Q., Wang, Z., He, X., Zhou, D.H. (2014). *A survey of eventbased strategies on control and estimation*. Systems Science & Control Engineering: An OpenAccess Journal, 2(1), 90-97.

[10] de la Torre, L., Chacón, J., Chaos, D., Dormido, S., Sánchez, J. (2019). *Using Server-Sent Events for Event-Based Control Laboratory Practices in Distance and Blended Learning*. European Control Conference, Naples, Italy, 3053-3058.

[11] de la Torre, L. (2019). *UNEDLabs/rip-spec: Second incomplete release (Version 0.361)*. doi: 10.5281/zenodo.3548490

6.- GLOSARIO

Array: vector.

Cookie: pequeño archivo enviado desde un servidor web a un cliente y que sirve al primero para recopilar datos al respecto de la navegación del último.

CPU: Unidad Central de Procesamiento. Se trata del elemento físico de un ordenador o dispositivo electrónico encargado de interpretar y ejecutar las instrucciones provenientes de un programa informático.

FGV: Function Global Variable. Tipo de variable global que se emplea en LabVIEW principalmente para poder disponer de un mismo dato en procesos que ocurren de forma paralela.

HTTP: Hypertext Transfer Protocol. Protocolo de comunicación que permite las transferencias de información en la World Wide Web.

JSON: JavaScript Object Notation. Formato de fichero e intercambio de datos de estándar abierto.

LabVIEW: software de ingeniería que emplea un lenguaje de programación visual gráfico pensado para diseñar sistemas hardware y software de pruebas y control, tanto reales como simulados.

NCS: Networked Control System. Sistemas de control en los que los lazos de control se implementan mediante una red de comunicaciones.

PID: Proportional-Integrative-Derivative Controller. Lazo de control que, en base a la desviación entre valor deseado de una variable y el valor real, proporciona una respuesta de actuación sobre el sistema compuesta por tres parámetros: proporcional, integral y derivativo

PV: Process Value. Valor real de una variable de proceso.

RIP: Remote Interoperability Protocol. Protocolo que permite la comunicación entre software dedicado para el usuario y software dedicado a la simulación o comunicación con equipos físicos.

Send-On-Delta: Concepto de detección de eventos, consistente en producir un evento cuando una variable experimenta una variación de su medida mayor a la de un valor predeterminado.

Sistemas de control embebidos: Sistemas de computación diseñados para la realización de tareas específicas y que se encuentran integrados dentro de un sistema mayor.

Slider: Control deslizante de un sistema de control cuyo valor cambia según lo deslizamos.

SP: Set Point. Valor deseado de una variable de proceso, también conocido como consigna.

SSE: Server Sent Events. Sistema de comunicación cliente-servidor en el que el cliente recibe los datos provenientes del servidor a partir de peticiones de envío generadas por el propio servidor.

TFM: Trabajo Fin de Master.

Timeout: tiempo máximo de espera de reconexión tras el cual expira una sesión de un usuario.

UNED: Universidad Nacional de Educación a Distancia.

VI: Virtual Instruments. Nombre que reciben los programas desarrollados con el software LabVIEW.