Último cambio Dictino Chaos: miércoles, 10 de julio de 2013

Índice de contenido

Motivación:

El protocolo binario original es imposible de depurar al realizarse lecturas y escrituras por TCP a lo largo de todo el código.

El protocolo JiL-XML utiliza en parte el protocolo XML-RPC para definir la interacción entre JiL y Java ofreciendo las siguientes ventajas:

1) El protocolo es legible (simplifica la depuración)

2) El protocolo es fácilmente extensible.

3) Se separa completamente la implementación de los comandos de la comunicación. Esto simplifica portarlo a otros lenguajes/plataformas (JiL-c , JiL-PHP, acceso desde smartphones, JiL-matlab…… etc)

4) Permite modificar FÁCILMENTE el método de transporte (TCP, UDP, HTTP… o cualquier cosa que acabe en P).

5) Mejora la seguridad de JIL ya que se dispone de todos los parámetros antes de intentar la ejecución de los métodos (en el formato binario si el cliente no envía todos los datos necesarios el servidor puede fallar en cualquier parte del programa que lea TCP).

Mecanismo:

Cada mensaje del protocolo se codifica en XML de acuerdo a la convención XML-RPC.

Una vez extraído el comando XML se obtiene el método a ejecutar y se extraen los argumentos.

Antes de nada se comprueba el formato de mensaje recibido y el tipo y número de argumentos, si aparece cualquier problema no se ejecuta ningún método. Sólo se admite el casting de int a double y no a la inversa para evitar pérdida de precisión.

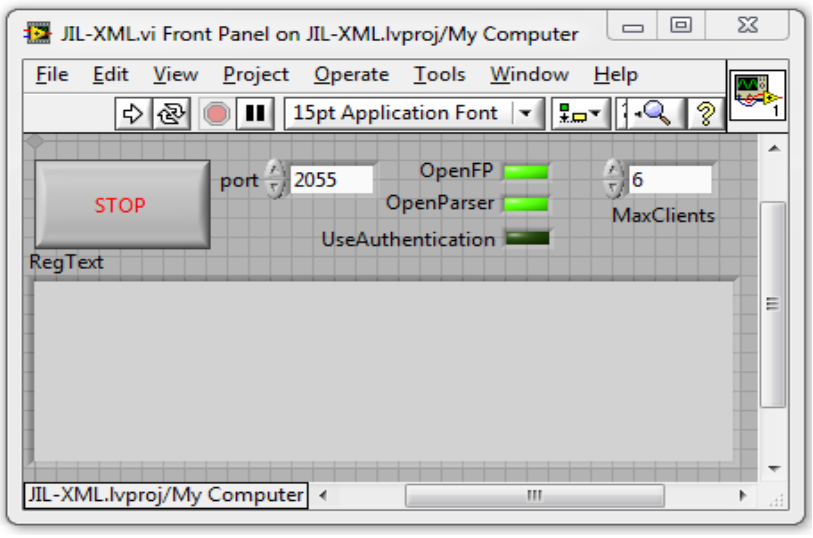
Si todo ha ido bien ejecuta el comando.

Se devuelve una respuesta que incluya el estado del comando o una **descripción textual comprensible del error que se ha producido.**

Funcionamiento:

El servidor Jil es un programa en LabVIEW que permite controlar remotamente otras aplicaciones (VI's). Para que JiL pueda acceder a un Vi este debe encontrarse en el directorio apps que a su vez está en el mismo directorio que JiL-XML.vi o JiL-XML.exe si se ejecuta en la versión compilada.

El aspecto de la aplicación es el siguiente:



Como puede verse la aplciación es muy sencilla y tiene un botón para pararla, controles para indicar el puerto de conexión (por defecto 2055), el número máximo de clientes que se pueden contectar simultaneamente así como opciones de abrir el panel frontal del VI que se ejecuta, abrir el panel frontal del Pharser y la opción de usar o no usar autenticación.

El panel de texto RegText muestra información textual de los clientes conectados, VIs abiertos y el estado de ejecución.

Una vez arrancado el servidor Jil-XML espera conexiones usando el protocolo XML-RPC y asigna un interprete “pharser” a cada petición que le llega que se comporta como una máquina de estados que son los siguientes:

Estados:

-Idle: Todavía no se ha establecido la conexión.

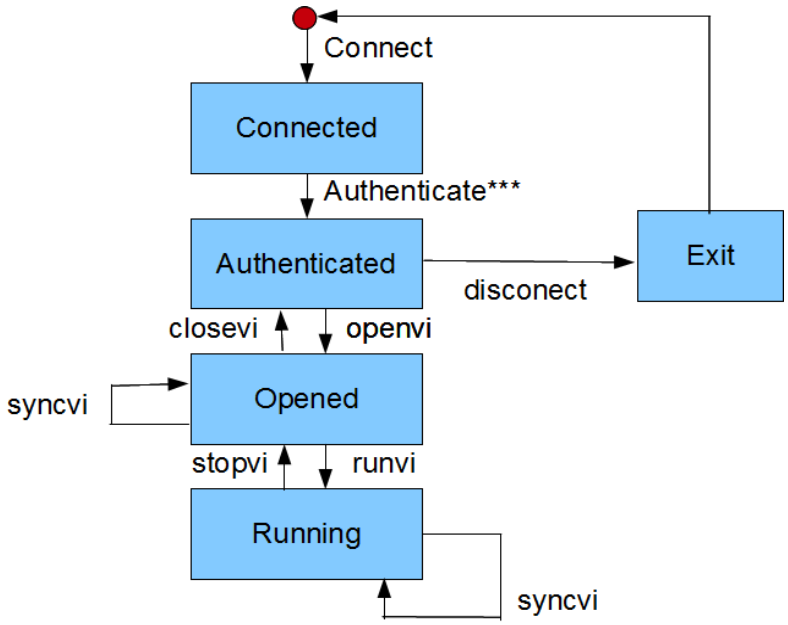
-Connected: Simplemente hemos empezado la conexión.

-Authenticated: Ya hos hemos autenticado.

-Opened: Vi abieto y listo para usarse.

-Running: Vi en ejecución.

-Exit: El servidor se está cerrando.



*** Si la autenticación está desabilitada este paso se realiza automáticamente sin necesidad de invocar el método authenticate.

Métodos:

Los métodos disponibles y su funcionamiento son los siguientes:

[string JIL-XMLversion, int SessionID]=jil.connect()

Se conecta a JiL y devuelve la versión del servidor y el identificador de conexión.

jil.uthenticate() *** por implementar

struct[]=OpenVI(string path)

Abre un VI cuyo path se pasa como argumento, devuelve un array de estructuras, cada una de las cuales contiene información sobre un control o indicador compatible con los tipos:

struct[i].name= nombre del conrol/indicador

struct[i].control_indicator=”control” o “indicator”

struct[i].DataType=”int”, “double”, “string” o “boolean”

confirmación=jil.runVI()

Ejecuta el VI abierto, devuelve un mensaje de confirmación “”

structOUT[]=jil.yncvi(structIN[])

Sincroniza variables con el servidor, recibe un array de estrucruras a sincronizar con los siguientes campos:

structIN.name=nombre del control o indicador

structIN.action=”get” o “set”

structIN.value=valor a establecier con el método “set” o valor para rellenar con el método “get”

El método devuelve una estructura con aquellos valores en los que se ha hecho “get”:

structOUT.name=nombre del dato leido

structOUT.value=valor leido

confirmación=jil.stopvi()

Detiene el VI abierto, devuelve un mensaje de confirmación “VI stopped”

confirmación=jil.closevi()

Cierra el VI abierto, devuelve un mensaje de confirmación “Vi closed sucesfully”

despedida=jil.isconect()

Cierra la sesión con servidor, devuelve un mensaje de despedida “See you soon”

Errores:

Cuando la petición es errónea o no sigue el orden adecuado de acuerdo al diagrama de estados se produce una respuesta descriptiva y un código de error.

Capa de transporte:

Se comienza usando el protocolo XML-RPC sobre HTTP en el puerto 2055.

El paquete de petición tiene una cabecera HTML y como mínimo un campo que indica la longitud:

```
HTTP/1.1 200 OK
Content-Length: longitud en bytes del mensaje
Content-Type: text/xml
```

Carga útil (Petición XML)

La respuesta va precedida de una cabecera HTTP con la siguiente información:

```
HTTP/1.1 200 OK
Content-Length: longitud en bytes del mensaje
Content-Type: text/xml
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Content-Type
Access-Control-Request-Method: POST, PUT
```

Carga útil (Respuesta XML)

Tanto el paquete de petición como el de respuesta acaban con dos CLRF sequidos.

JiL server permite compresion transparente del xml usando gzip, para ello la cabecera de la petición contendrá el siguiente parámetro:

```
Content-Encoding: gzip
```

Cuando el contenido se reciba en formato comprimido JiL contestará del mismo modo (y por tanto con el mismo parámetro en la cabecera):

```
HTTP/1.1 200 OK
Content-Length: longitud en bytes del mensaje
Content-Type: text/xml
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Content-Type
Access-Control-Request-Method: POST, PUT
Content-Encoding: gzip
```

Modificaciones respecto a JIl:

Se rescribe JiL Por completo aprovechando las ideas y el código ya existente.

Se unifican los “set” y “get” en una única instrucción versatil “syncVI”

Se elimina los métodos obsoletos para abrir y se reemplazna con los nuevos de la versión 2010.

Funcionamiento de JIL-XML

JiL-XML.vi es un servidor que espera conexiones HTTP en un determinado puerto. Cuando recibe una conexión entrante verifica si se ha alcanzado el número máximo de conexiones permitidas.

En caso afirmativo devuelve al cliente un error:

[001]: “Too many users connected” : No te deja conectar, la conexión se cierra.

Si no se ha excedido el máximo se abre una isntancia de PARSER.vi para atender al cliente y se registra la conexión mediante registry.vi.

Periódicamente JIL-XML limpia el registro invocando el comando clean sobre registry.vi y muestra el estado de las conexiones.

Cuando se pulsa el boton Stop Jil-XML cierra las conexiones TCP y deja un segundo para que los PHARSER lo derecten y se cierren limpiamente. Pasado este tiempo aborta todos los vi abiertos invocando close sobre registry.vi

Funcionamiento de PARSER.vi

PARSER.vi es un vi reentrante que atiende las conexiones del cliente. Recibe los datos de configuración que le envía JIL-XML y a partir de ahí se ejecuta de forma autónoma.

El ciclo de ejecución es el siguiete:

1) En primer lugar lee la cabecera HTTP línea a línea hasta que se encuentran los dos CLRF. Se decodifica la longitud del mensaje y se lee el XML.
2) Se comprueba la cabecera en busca de compresion gzip, si está activada descomprime el xml.
3) Cuando llega el comando realiza la interpretación del mismo invocando a XML_method_params. Este le devuelve el nombre del método, la lista de parámetros y la lista de tipos así como un indicador de error en XML.
4) Comprueba que no hay error (el XML está vacío) y a continuación pasa a realizar las comprobaciones del método en este punto se pueden producir los siguientes fallos:

[100]: Wrong HTTP header: XXX Se produce si no es posible leer la cabecera o si no contiene el campo Content-Length. Devuelve el texto que se ha leído como cabecera.
[104]: Error during XLM decompression.
[101]: Too may arguments...: este errror se produce si hay demasiados argumentos

[102]: Too few arguments...: lo contrario

[103]: Invalid arguments: function XXX, whaits for (arg list)

5) Comprueba el estado de JIL para impedir que se ejecuten los comandos en un orden incorrecto (por ejemplo ejecutar un vi sin abrirlo previamente).
6) Si todo es correcto se ejecuta el método correspondiente que está enbebido en el pharser excepto () y actualiza el estado.

Cuando se cierra la conexión TCP por cualquier motivo o cuando el cliente invoca el método DISCONECT, el parser cierra (en su caso) el vi abierto, luego termina la conexión y por último se cierra a si mismo.

Interpretación de comandos:

XML_method_params

Este vi se encarga de inspeccionar la petición en XML y de sacar el método y los parámetros independientes del contexto (todos excepto Array y Struct que necesitan saber el tipo de array de destino.

Admite como entrada el mensaje en XML y devuelve el método (un tipo enumerado con los métodos admitidos) un vector de tipos (enumerado con la descripción de los tipos) y un array de variants con los valores de todos los escalares.

En el caso de los arrays y structs los valores del variant contienen el XML sin procesar.

El mecanismo de funcionamiento consiste en varios pasos que se ejecutan solamente si el descriptor de error está vacío (no hay error en los pasos anteriores) de acuerdo a la siguiente secuencia:

1) Saca la cabecera.

Si hay algún problema se devuelve el error:
[901]: Header not found at the beginning of the XML, or or ill formed

2) Comprueba la sintaxis del cuerpo.

Si una etiqueta está vacía o no se cierra devuelve

[902]: Bad formed tag in

...

Si hay texto que no está encerrado en etiquetas devuelve el error:

[903]: Text not enclosed into tags: ...

Si una etiqueta de cierre no cierra a la anterior devuelve

[904]: Error: trying to close "tag1" whith "tag2".

Si después del análisis quedan etiquetas sin cerrar devuelve

[905]: Error, tag "tag" never closed.

3) Saco el methodCall

Obtengo el methodcall y combruebo que no hay nada fuera de él. En caso de fallo se genera:

[906]: Error: MethodCall not found or something outside methodcall

4) Obtengo el methodName

Se pueden producir varios errores. En primer lugar que encuentre algo antes del methodName:

[907]: Unexpected string before methodName: ...

Puede que el método se lea bien pero sea desconocido:

[908]: Unknown Method: "method"

5) Saco la cabecera params

Esta etiqueta puede no existir si no hay parámetros, eso no se considera un fallo. Si hay algo fuera de la cabecera se indica con:

[909]: Unespected string oustisde "params": ...

6) Separo los parámetros

En este paso se recorre el XML y se separa cada elemento “param”. El bucle recorre todos los parámetros hasta que hay algún fallo o se acaba el XML. El único fallo posible es que haya datos entre los param, el error indica el parámetro que produce el error:

[910]: Unespected string after param N

7) Proceso los parámetros:

Este paso se realiza en un vi separado XML_value_to_variant.

XML_value_to_variant

Este vi procesa cada uno de los parámetros comprobando que el tipo es válido y que puede leerse. Devuelve un variant con el contendio del parámetro (el valor si es un escalar y el XML sin procesar si es un tipo compuesto) así como su tipo. Hay tres errores que pueden aparecer en el proceso. Primero que el parámetro no esté bien formado

[811]: Error in value:

“Xml del parámetro... “

Segundo que no se conozca el tipo

[812]”Unknown data type: type” se produce al encontrar un argumento de un tipo no reconocido en XML-RPC indica textualmente el tipo recibido.

Tercero que no se ajuste al formato esperado:

[813]”Error reading value. Type recieved “tipo” string recieved “string”

Proceso de arrays: XML_to_Array

Toma un variant que contiene el XML del array y lo convierte en un array de parámetros separqados en XML. Después los convierte a un array de tipos y valores en un variant (los escalaser se convierten y los tipos compuestos de dejan tal cual). Los errores que se pueden producir son:

[700]: Unespected text ouside <data> inside an Array

[701]:Unexpected string in array after value N

Proceso de clusters: XML_read_cluster

Se le indica un parámetros a leer y los lee del cluster devolviendo el valor y el tipo.

[801]: Unexpected string in struct after member N

[802]: Member ”MemnerName“ not found on cluster

Gettag.vi

Este vi se utiliza frecuentemente en las fuciones anteriores, adminte un XML y el nobre de una etiqueta y devuelve el contenido de la misma así como el texto que hay delante y detrás y un idicador de error.

Registry.vi

Para controlar que vi están abiertos se crea un registro. El regitro se almacena en una variable global “GlobalRegitry” que solo se accede mediante Registry.vi. Admite las siguientes operaciones:

init

Crea un registro vacío

regitry

Crea un registro nuevo y admite como entrada una estructura client con la IP y el puerto así como la referencia del parser. Es llamado cada vez que JIL-XML admite una conexión nueva.

clean

Comprueba el estado del registro y quita las entradas obsoletas. Para ello verifica los parsers que están en ejecución y si detecta alguna referencia obsolieta cierra las referencias y los vi que pudiera tener abiertos.

update

Admite una estructura OpenedVi con la referencia y el path del vi abierto así como la referencia del parser que lo llama. Este método comprueba en primer lugar que el vi no está siendo usado varias veces sin ser reentrante. Si puede efectua el registro y si hay un error lo indica activando la salida **vi in use?**

Este método es utilizado por el parser cuando se hace OpenVI o CloseVI (en este caso se pasa una rereferencia vacía). Su utilidad es impedir que se abran dos veces un mismo vi (no reentrante) e informar a JIL-XML de que vis están abiertas y quién las ha abierto.

close

Cierra todos los vi referencias y vacía el registro. Es utilizado por JIL-XML para asegurarse de que

se cierra todo al salir.

Después de ejecutar cada métido el registro produce como salida una cadena de texto que describe de forma comprensible el estado del registro indicando que clientes están conectados y que vi (o instancia del vi reentrante) tienen abierto. También devuelven el número de conexiones activas y la señal de error en el registro (**vi in use?**)

Métodos del protocolo:

Connect()

Hay un pequeño intercambio de datos entre JIL y el cliente. En el intercambio el cliente y el servidor se identifican especificando las versiones del protocolo (para detectar posibles errores o incompatibilidades antes de continuar). Se indica también si es necesario hacer autenticación. Se incluye un identificador de sesión aleatorio. Pasa al estado Connected.

EL CLIENTE ENVÍA:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
	<methodName>jil.connect</methodName>
</methodCall>
```

SI TODO VA BIEN EL SERVIDOR RESPONDE:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
	<params>
		<param>
			<value>
			<struct>
				<member>
					<name>version</name>
					<value><string>%JIL-XMLversion%</string></value>
				</member>
				<member>
					<name>sessionID</name>
					<value><i4>%SessionID%</i4></value>
				</member>
			</struct>
			 </value>
		</param>
	</params>
</methodResponse>
```

POSIBLES FALLOS:

[001]: “Too many users connected” : No te deja conectar, la conexión se cierra.

(este es un fallo que se produce antes de ejecurarse el método connect (de hecho no llega a ejecutarse)

[201]: “Warnig: User alredy connected” : No pasa nada, el usuario estaba conectado se queda todo igual.

Authenticate()******

Se usa un usuario y una contraseña. El usuario y la contraseña no se envían en texto plano, se concatenan con el identificador y se aplica MD5, esto se envía al servidor para que lo compruebe y si concuerda con algún usuario autorizado se pasa al estado Authenticated.

****Todavía no está hecho, de momento asumo que todo está bien y autentico por las buenas. Se considera dos errores relacionados con el estado

[209]: Error: user not connected.
[210]: "Warnig: User alredy authenticated" (no hace nada)

OpenVI()

Por su complejidad esta operación se realiza por medio de un vi separado OpenVI.vi. Este recibe la ruta del Vi que hay que abrir. Sólo se invoca en estado Authenticado y si tiene éxito pasa al estado Opened. Devuelve un array de estructuras con la descripción de los controles e indicadores, cada elemento del array tiene un string "nombre" y un string control_indicador que puede valer "control" o "identificador" y un string tipo que indica el tipo de dato (int double booleano...).

EL CLIENTE ENVÍA:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
	<methodName>jil.openvi</methodName>
	<params>
		<param><value><string>vi_path</string></value></param>
	</params>
</methodCall>
```

SI TODO VA BIEN EL SERVIDOR RESPONDE:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
<params>
   <param>
<value>
	<array>
<data>
<value>
	<struct>
	  <member>
		<name>name</name>
		<value><string>"nombre"</string></value>
	  </member>
	  <member>
		<name>control_indicator</name>
		<value><string>"control" ó "indicator"</string></value>
```

```
        </member>
        <member>
            <name>DataType</name>
            <value><string>tipo</string></value>
        </member>
    </struct>
</value>
…
…
</data>
</array>
</value>
  </param>
</params>
</methodResponse>
```

NOTA1: Solo se devuelven los indicadores y controles con un formato compatible (por el momento int, double, string y boolean)

NOTA2: la variable "stop" es una variable reservada por JIL para terminar de forma limpia un Vi, por este motivo la función openVI no mosratá ningún control con ese nombre.

Los errores que pueden aparecer son:

que el estado no sea correcto, si el estado es idle se devuelve

[202]: Error: user not authenticated.

Si el estado es otro (opened, running or closing...)

[203]: Error: There is a vi alredy opened.

Intermamente al método OpenVI.vi realiza las siguientes acciones:

1) comprueba que el path es correcto y obtiene el path absoluto a partir del relativo. Puede aparecer el error:

[301]: Error: path not valid "path"

2) Si el path es correcto intenta abrir el vi (primero como reentrante y si no es posible como vi normal):

[302]: Error, cannot open Vi:

"descriptor de error de labiew"

3) Si consigue abrirlo lo registra, entonces puede producirse el siguiente error:

[303]: Error opening: this Vi is opened by other client

4) Si no se produce ningún error abre el vi obtiene la lista de controles e indicadores y devuelve la referencia al mismo así como la lista (como estrucura y como XML)

RunVI()

EL CLIENTE ENVÍA:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
	<methodName>jil.runvi</methodName>
</methodCall>
```

Sólo funciona devuelve un ACK:

<?xml version="1.0" encoding="UTF-8"?>

<methodResponse>

	<params>

		<param>

			<value><string>VI running</string></value>

		</param>

	</params>

</methodResponse>

Si el vi no está abierto devuelve:

```
[204]: Error runing: Vi is not opened.
Si el vi esta corriendo devuelve
[205]: Error: Vi is running.
```

Si se produce un error de otro tipo:

```
[401]: Unexpected error: Vi cannot be run, Vi closed for safety reasons.
En este caso ante un error inesperado se cierra el vi.
```

StopVI()

Pasa al estado Opened y para el Vi. Por la complejidad del proceso se usa StopVI.vi

EL CLIENTE ENVÍA:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
	<methodName>jil.stopvi</methodName>
</methodCall>
```

Sólo funciona devuelve un ACK:

<?xml version="1.0" encoding="UTF-8"?>

<methodResponse>

	<params>

		<param>

			<value><string>VI stopped</string></value>

		</param>

```
	</params>
</methodResponse>
```

Si el vi no está corriendo devuelve:

```
[206]: Error stopping: Vi is not running.
```

Si el vi no está corriendo devuelve:

```
[501]: Error: Vi not respondign, cannot be stopped. Trying to close it.
```

StopVI.vi intenta realizar un cerrado limpio del vi para ello busca un botón de nombre STOP (sin distinquir entre mayúculas y minúsculas) y lo pulsa. Espera un tiempo prudencial y si no consigue apagar al vi lo cierra a la fuerza.

CloseVI()

Pasa al estado Authenticated y cierra el Vi.

EL CLIENTE ENVÍA:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
	<methodName>jil.closevi</methodName>
</methodCall>
```

SI TODO VA BIEN EL SERVIDOR RESPONDE:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
	<params>
		<param>
			<value><string>Vi closed sucesfully</string></value>
		</param>
	</params>
</methodResponse>
```

Sólo se puede invocar desde el estado opened. Si se llama desde otro estado produce los siguiente errores:

Si está en run

```
[207]: Error: Vi is running, close it first.
```

Si es cualquier otro estado:

```
[208]: Error closing: Vi is not opened.
```

```
[601]: unespected error during close.
```

Disconect()

Cierra el parser, devuelve un mensaje de despedida y pasa al estado Exit.

EL CLIENTE ENVÍA:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
      <methodName>jil.disconnect</methodName>
</methodCall>
```

SI TODO VA BIEN EL SERVIDOR RESPONDE:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
      <params>
            <param>
                  <value><string>See you soon</string></value>
            </param>
      </params>
</methodResponse>
```

Sólo se puede invocar desde el estado connected o authenticated. Si se llama desde otro estado produce los siguiente error:

```
Si está en open o run
[209]: Error: There is a Vi opened close it before disconnecting.
```

SyncVI()

El JiL Original disponía de muchos métodos para leer (get) y escribir (set) cada uno de los cuales operaba con un formato distinto y requería una implementación distinta en el cliente y el servidor, esto hace que el mantenimiento sea complejo. Además muchos métodos no tenían implementación en el cliente. En su lugar JIL-XML usa una aproximación distinta utilizando las capacidades de XML-RPC para establecer un **único método en el servidor** que es flexible y permite hacer muchas operaciones así como agregar fácilmente nuevos tipos de datos.

Este método es una instrucción versátil **SyncVI** que se implementa en SyncVI.vi y que sirve para leer o escribir variables. Admite como entrada un número arbitrario de variables expresado como un array de clusters. Cada varianble está definida en un cluster que indica si hay que leer o escribir (action) el nombre de la variable (name) y el valor de dicha variable (`nombre de la variable` que implicatamente indica el tipo). **Los tipos soportados por el momento son "int, double, string y boolean", se comprueba que los tipos de los datos recibidos son iguales a los del control/indicador correspondiente y sólo se permite la conversión de entero a doble preción. Es importante destacar que SyncVI no actualiza el control "stop" ya que es una variable reservada por JIL para la finalización límpia de un VI.**

EL CLIENTE ENVÍA:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
      <methodName>jil.syncvi</methodName>
<params>
    <param>
     <value>
      <array>
       <data>
        <value>
         <struct>
               <member>
                 <name>name</name>
                 <value>nombre de la variable</value>
```

```
                </member>
                <member>
                  <name>action</name>
                  <value>”get” ó “set”</value>
                </member>
                <member>
                  <name>value</name>
                  <value><tipo>valor</tipo></value>
                </member>
          </struct>
         </value>
            ....
        </data>
       </array>
       </value>
      </param>
    </params>
</methodCall>
```

SI TODO VA BIEN EL SERVIDOR RESPONDE con un array de clusters que contiene los nombres y valores de las variables sobre las que se ha hecho “get”.

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
<params>
     <param>
<value>
       <array>
<data>
<value>
<struct>
          <member>
             <name>name</name>
             <value><string>nombre de la variable</string></value>
          </member>
          <member>
             <name>value</name>
             <value><tipo>valor</tipo></value>
       </member>
</struct>
…
…
…

</value>

</data>
</array>
</value>
   </param>
</params>
</methodResponse>
```

Con este formato es muy fácil emular los comandos que existían en el cliente EJS para cada get y set (basta con crear la estructura correspondiente al tipo que se quiere leer o escribir, enviarla y obtener la respuesta).

Antes de ejecutar el comando además de los errores 1XX pueden aparecer los siguientes errores:

Si no está en open o run

```
[210]: Error: Vi is not opened/running.
```

El funcionamiento del vi es el siguiente:

1) Extraigo los parámetros del array mediante XML_2_Array

   `[701]:Error, SyncVI waits for a array of structs.`

2) Extraigo los elementos de caca cluster (parámetro) y compruebo los posibles errores

   `[702]:Error sync elements must be clusters with name (string), action (string) and value`

   `[703]:Error type of "nombre" not on SyncVI suported (jet)`

3) Compruebo que hay un control/indicador compatible con la operación pedida

   `[704]:Error variable "nombre" not found or wrong type.`

4) Realizo las operaciones.

   `[705]:Error , unknown command "comando" in SyncVI`

   `[706]:Internal LabVIEW error:`

   `"Source of error in LV" in SyncVI`

5) Creo el cluster de respuesta.

Fallos: Error2XML.vi

Si un método no funciona se devuelve un mensaje de error descriptivo usando la especificación del protocolo XML-RPC. Para ello se crea el vi Error2XML. Dicho vi admite el identificador de fallo (numérico) y un descriptor de fallo legible y genera el siguiente mensaje de respuesta:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
<fault>
	<value>
		<struct>
			<member>
				<name>faultCode</name>
				<value><int>ID de fallo</int></value>
			</member>
			<member>
				<name>faultString</name>
				<value><string>Descriptor de  fallo</string></value>
			</member>
		</struct>
	</value>
</fault>
</methodResponse>
```

Puesto que el descriptor de fallo puede tener cualquier tipo de carácter se hacen las substituciones necesarias en el descriptor de fallo para mantener el formato XML-RPC, esto es cambiar & por &amp y < por &lt.