

## Índice de contenido

Motivación:	2
Mecanismo:	2
Capa de transporte:	2
Modificaciones respecto a JIL:	2
Funcionamiento de JIL-XML:	3
Estados de JiL:	3
Funcionamiento de PARSE.vi	3
Interpretación de comandos:	4
XML_method_params	4
XML_value_to_variant	6
Proceso de arrays: XML_to_Array	6
Proceso de clusters: XML_read_cluster	6
Gettag.vi	6
Registry.vi	7
init	7
regitry	7
clean	7
update	7
close	7
Métodos del protocolo:	7
Connect()	7
Authenticate()*****	8
OpenVI()	8
RunVI()	10
StopVI()	10
CloseVI()	11
Disconnect()	12
SyncVI()	12
Fallos: Error2XML.vi	14

# 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.

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.**

# Capa de transporte:

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

El paquete tiene una cabecera de tipo **int32** que codifica el tamaño del mensaje precedido del XML.

# 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 reemplaza con los nuevos de la versión 2010.

## Funcionamiento de JIL-XML

JiL-XML.vi es un servidor que espera conexiones TCP 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 instancia 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 botón Stop JiL-XML cierra las conexiones TCP y deja un segundo para que los PARSER lo detecten y se cierren limpiamente. Pasado este tiempo aborta todos los vi abiertos invocando close sobre registry.vi

## Estados de JiL:

El servidor será una máquina de estados con los siguientes estados:

- Idle:** Todavía no se ha establecido la conexión.
- Connected:** Simplemente hemos empezado la conexión.
- Authenticated:** Ya nos hemos autenticado.
- Opened:** Vi abierto y listo para usarse.
- Running:** Vi en ejecución.
- Exit:** El servidor se está cerrando.

## 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 siguiente:

- 1) En primer lugar lee TCP y espera de un comando.
- 2) 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.
- 3) 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:

[101]: Too many arguments...: este error se produce si hay demasiados argumentos

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

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

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

Cuando se cierra la conexión TCP por cualquier motivo o cuando el cliente invoca el método DISCONNECT, 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 compruebo 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 contenido 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 separados en XML. Después los convierte a un array de tipos y valores en un variant (los escalares se convierten y los tipos compuestos se dejan tal cual). Los errores que se pueden producir son:

[701]:Unexpected string in array after value **N**

## Proceso de clusters: XML\_read\_cluster

Se le indica un parámetro 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 funciones anteriores, admite un XML y el nombre de una etiqueta y devuelve el contenido de la misma así como el texto que hay delante y detrás y un indicador de error.

# Registry.vi

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

## init

Crea un registro vacío

## registry

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 obsoleta 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 método 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><string>version</string></value>
    </param>
    <param>
      <value><i4>SessionID</i4></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 ejecutarse 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 Authenticated 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>
    <array>
      <value>
        <struct>
          <member>
            <name>name</name>
            <value><string>nombre</string></value>
          </member>
          <member>
            <name>control_indicator</name>
            <value><string>"control" ó "indicator"</string></string>
          </member>
          <member>
            <name>DataType</name>
            <value><string>tipo</string></value>
          </member>
        </struct>
      </value>
    </array>
  </param>
</params>
</methodResponse>
```

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

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 already opened.

Interamente 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 estructura 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 distinguir entre mayúsculas 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]: unexpected error during close.
```

## Disconnect()

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 variable 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 implicativamente indica el tipo). **Los tipos soportados por el momento son “int, double, string y boolean”**

EL CLIENTE ENVÍA:

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

```

<param>
  <array>
    <value>
      <struct>
        <member>
          <name>name</name>
          <value><string>nombre de la variable</string></value>
        </member>
        <member>
          <name>action</name>
          <value><string>"get" ó "set"</string></string>
        </member>
        <member>
          <name>value</name>
          <value><tipo>valor</tipo></value>
        </member>
      </struct>
    </value>
  </array>
</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>
    <array>
      <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>
    </array>
  </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 cada 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 supported (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.