Writing a Tango Device in LabVIEW

Introduction

So far, the Tango binding for LabVIEW was a pure client platform. It now allows to turn any LabVIEW application into a Tango device.

The proposed model is very close to the one offered for any other language (C++, Java or Python). Any developer with a minimum of LabVIEW and Tango knowledge should be able to implement a simple Tango device in, literally, a few tens of minutes.

This first release supports most of the Tango 9 features. Regarding the data types, the *DevEncoded* and *Pipe* are currently not supported. They require a particular attention – notably in the choice of the associated data structures on LabVIEW side. Anyway, the provided features are more than enough to start writing advanced Tango devices.

Particular attention has been paid to reliability and performances. The data exchanges between the two worlds are notably based on proved code inherited from the binding client API – which is known to be stable and fast.

Quick Guided Tour

Installing the binary distribution

Whatever is the platform on which you plan to run the Tango binding, the installation is conceptually the same:

1. uncompress the package into the location of your choice. In what follows, we assume that the Tango binding has been installed into a directory which path is ***TBFL_ DIR***.

2. make sure that the binding runtime – i.e. *TBFL_ DIR/lib* - is in your "dynamic loader path". Concretely, you have to make sure that the full set of Tango DLLs (or shared libraries) is located in a directory listed in your Windows ***PATH*** environment variable or in your Linux ***LD_LIBRARY_PATH*** environment variable. In doing so, we ensure that LabVIEW is able to successfully load the Tango binding.

Note: You can also launch LabVIEW in the required environment by adapting the provided launchers ***TBFL_ DIR/launcher/[windows, linux]/start-labview.[bat, sh]***.

Running the provided example

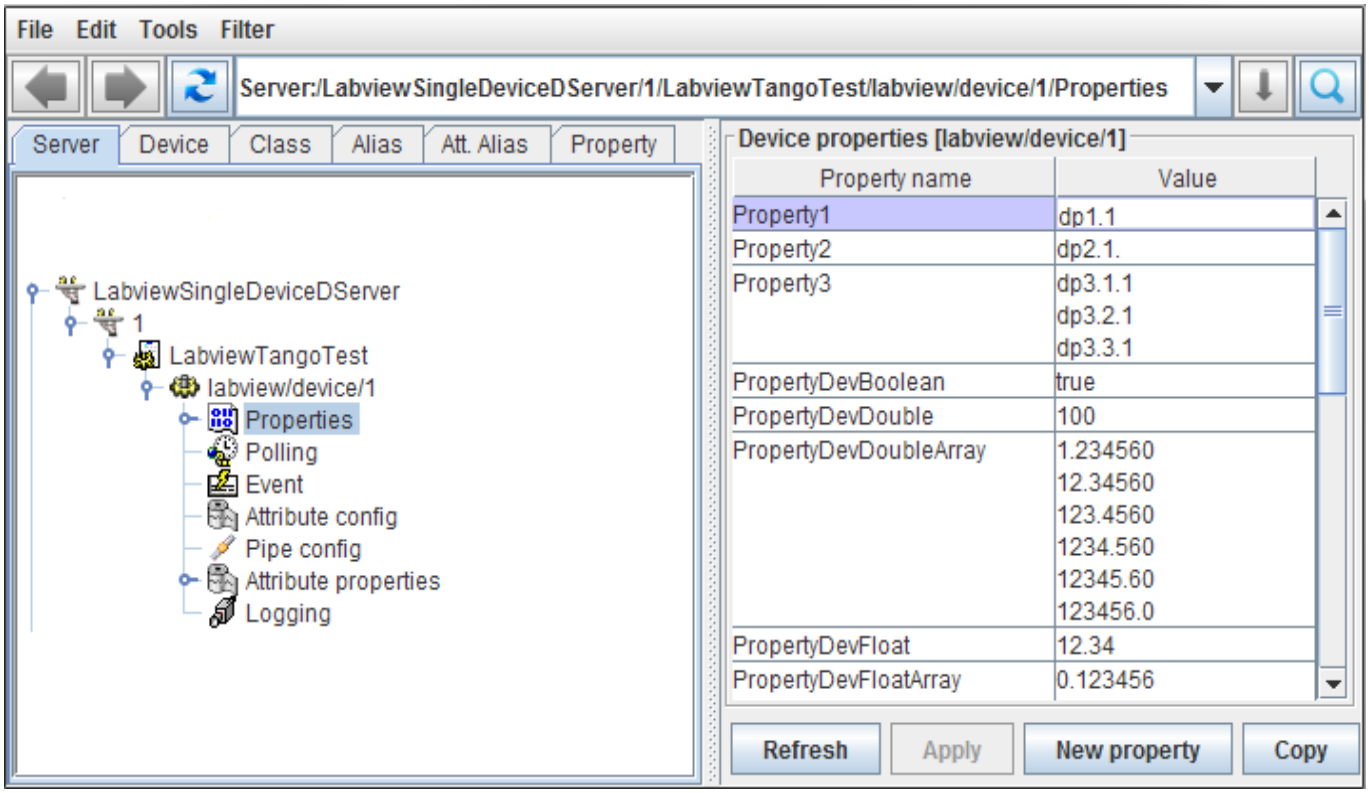
The Tango binding comes with an examples library containing a full-featured Tango device named ***SingleDeviceDServer***. This VI implements a Tango device server embedding a single instance of the so-called ***LabviewTangoTest*** class. It constitutes a good starting point to discover the main VIs involved into the implementation of a LabVIEW device.

Registering the device

Open a *Jive* instance and register the *SingleDeviceDServer* into the Tango database by loading the provided property file.

Jive -> File menu -> Load property file then browse your file system and select the following file: ***TBFL_DIR/lib/examples/dservers-registration/LabviewSingleDeviceDServer.tdb***

You should obtain the following entry in the Tango database:

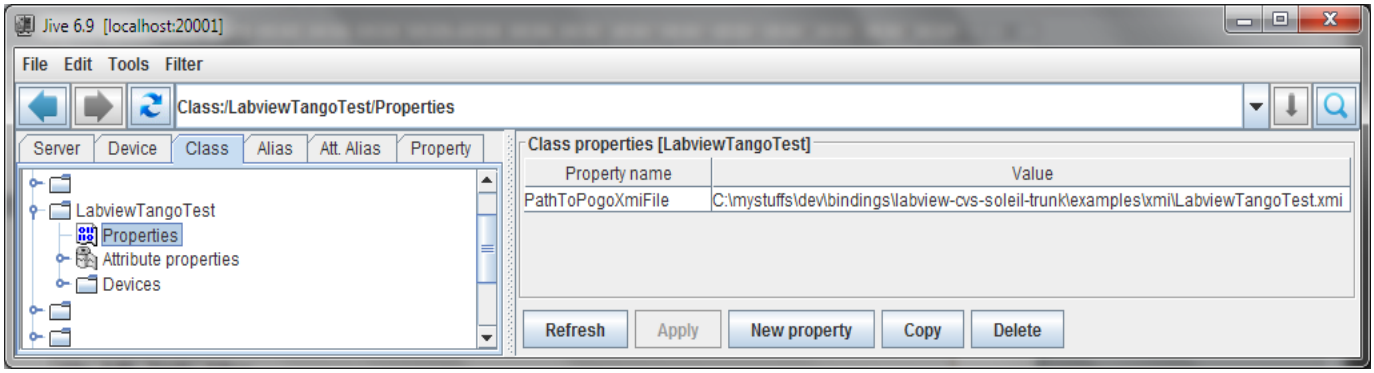


Specifying the path to the POGO ".xmi" file:

The LabVIEW binding will dynamically instantiate the Tango device and populate its interface according to the content of the ***xmi*** file of its Tango class. This *xmi* file is nothing but the *xml* file generated by ***POGO*** which describes the full device interface in terms of properties, commands, attributes, states, etc.

For each class we want to run in a LabVIEW device server, we consequently have to specify the path to the associated *xmi* file in order to allow the Tango binding to instantiate the corresponding devices. This information is passed to the binding using a specific and mandatory class property named ***PathToPogoXmiFile***.

In order to change its value, click on the *Class* tab of *Jive* panel, select the *LabviewTangoTest* class then change the *PathToPogoXmiFile* property according to your local installation: ***TBFL_DIR/examples/xmi/LabviewTangoTest.xmi***



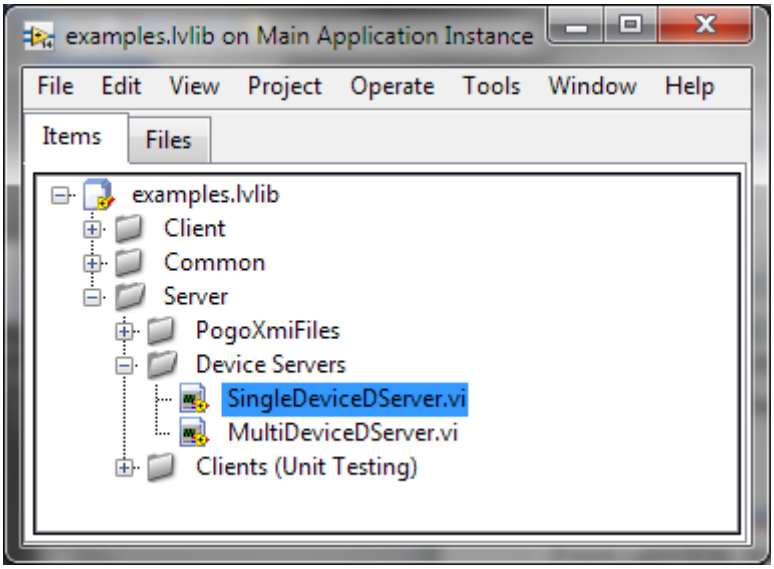
Good news, your LabVIEW device is now ready to run!

Launching LabVIEW

Launch LabVIEW from any "Tango binding aware" environment - i.e. from any environment from which the dynamic loader can find the required DLLs or shared libraries. In case you adapted the provided launcher, it's the right time to use it.

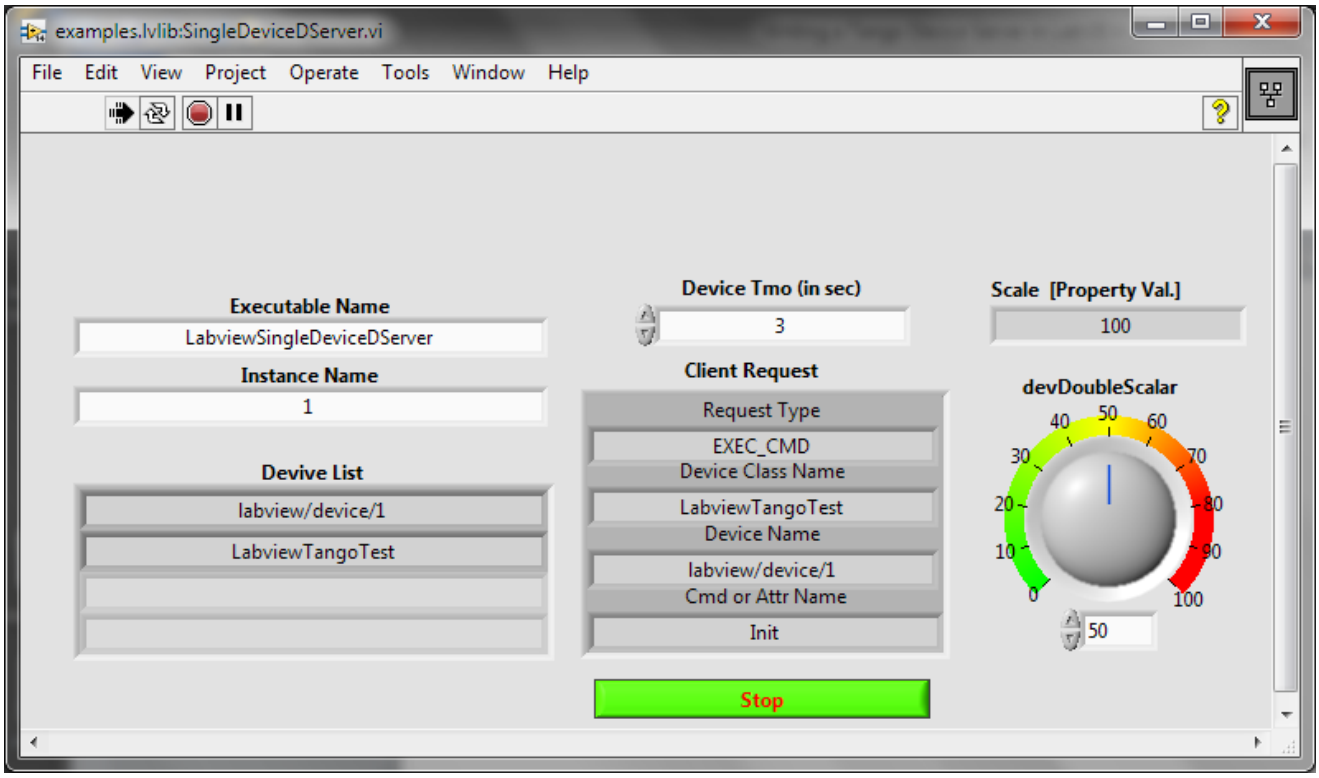
Open the SingleDeviceDServer VI

From LabVIEW, browse your file system and open the examples library: ***TBFL_DIR/examples/examples.lvlib***.



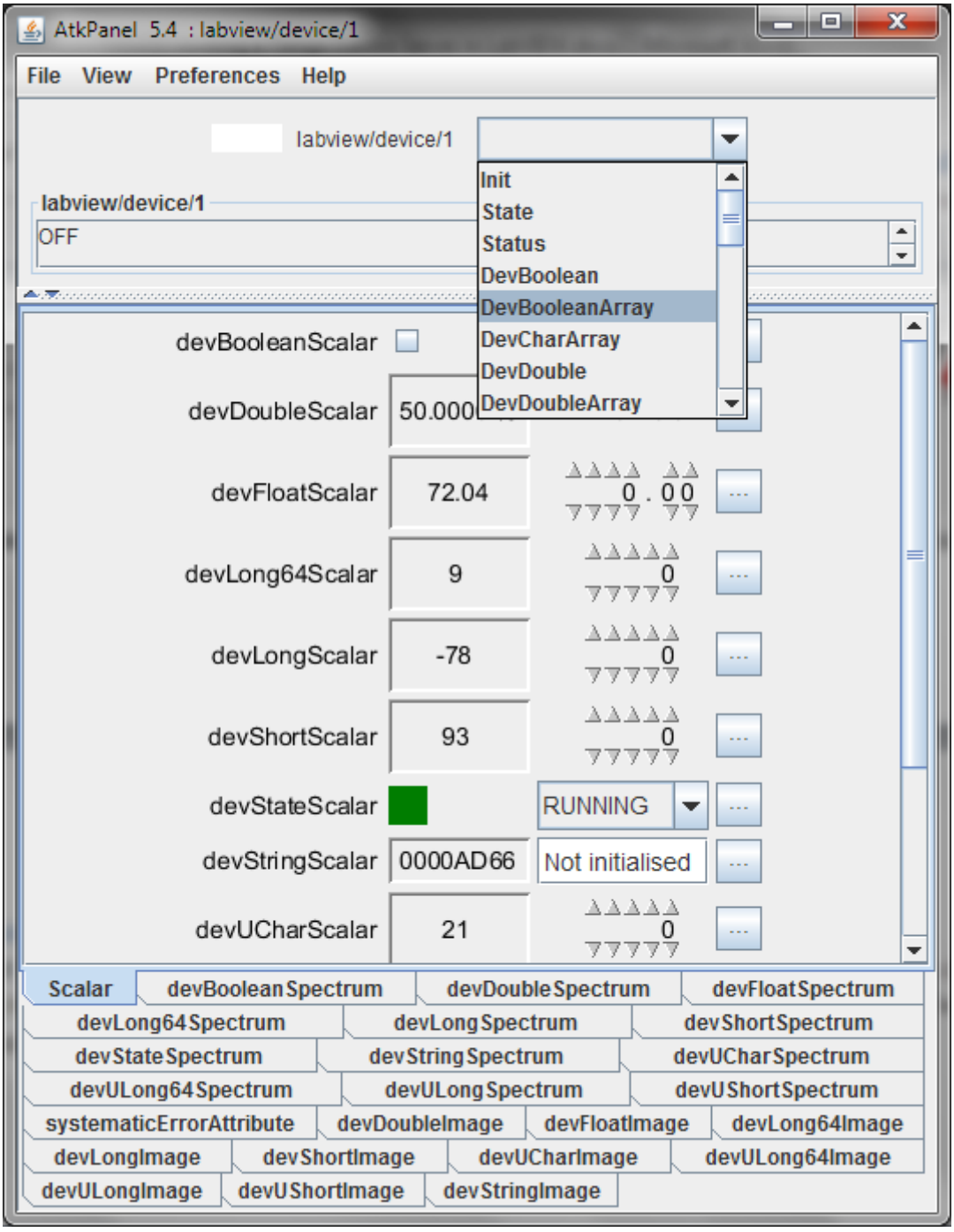
Open the ***SingleDeviceDServer*** VI then start it using the LabVIEW ***run*** button.

If everything is fine you should obtain something like:

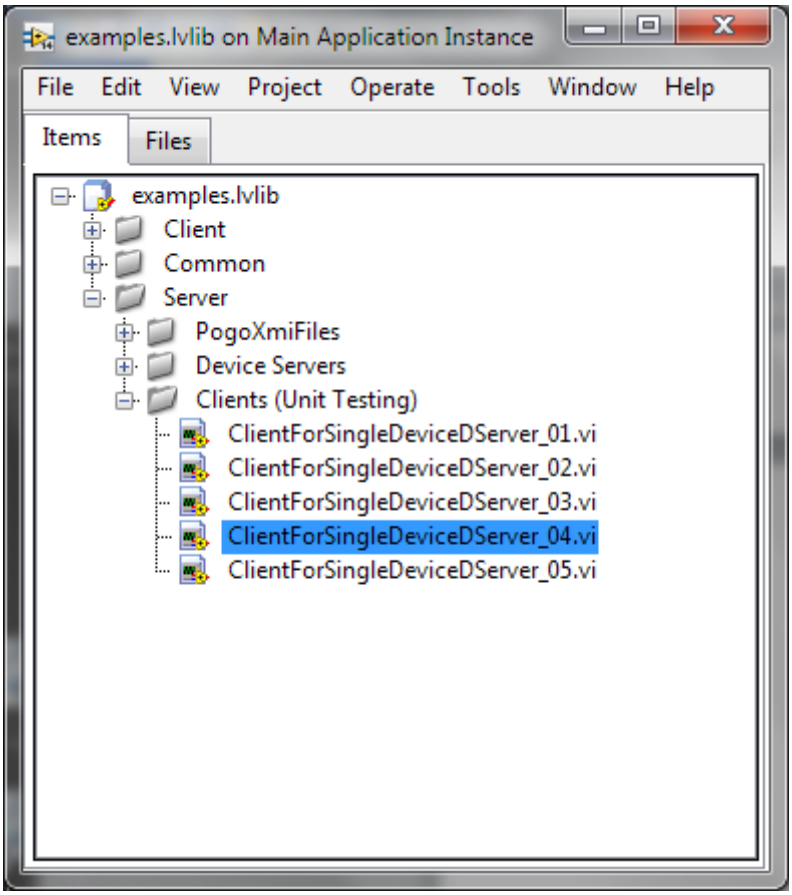


Playing with the device

The easiest way to test the device is to open an ***ATKPanel*** from Jive by double-clicking on its device name. Navigating in the device interface, you will notice that it has an "echo like" command and a read/write attribute for each supported data type (Jive only shows the ones it can display).



The binding also comes with a set of unit tests you can use to test the device and evaluate its performances and stability. Don't hesitate to launch a couple of them in order to estimate the responsiveness of the device under heavy load (e.g. play with the ***devDoubleScalar*** knob while the device is responding to client requests). Launching the ***ClientForSingleDeviceDServer_[03,04].vi*** generates something like 1000 requests/s on a i7 laptop.



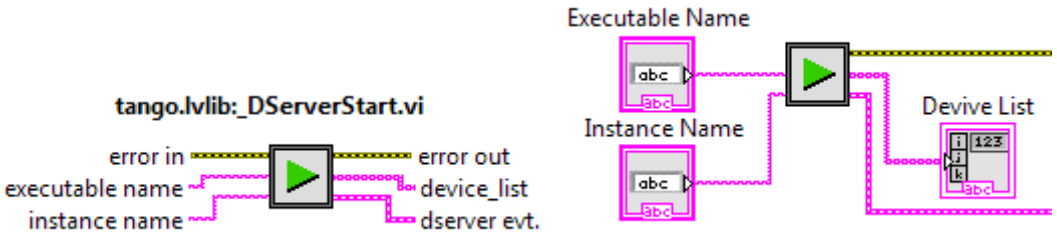
Implementation details

Main VIs

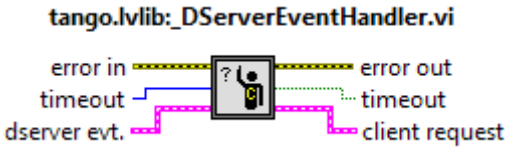
The implementation scheme of a LabVIEW device is really simple. Basically, all you need to do is to start the device server, to reply to client requests and, eventually, to quit one day or another.

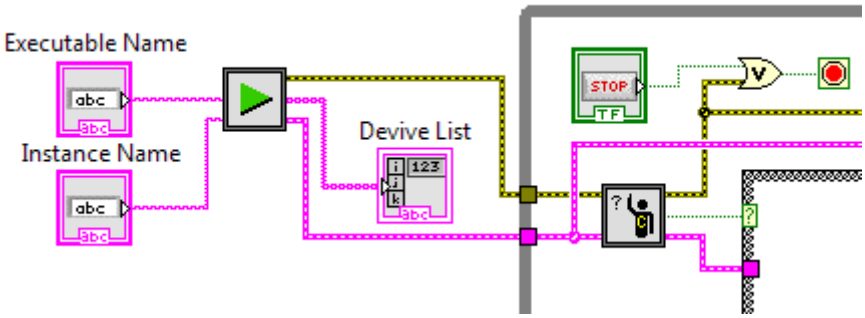
In what follows, we illustrate the "single device device-server" case and assume that the associated *xmi* file has already been generated from POGO and properly registered in the Tango database - remember the *PathToPogoXmiFile* class property?

The device implementation starts with a call to the ***_DServerStart.vi***. As its name indicates, this VI starts the Tango device server which ***executable*** and ***instance*** names are specified as input parameters. This simple call does a lot in fact - there's a lot of magic behind it. Let's say that once it return all the devices belonging to device-server are up and running. They will simply do nothing until you start to handle the incoming client requests.

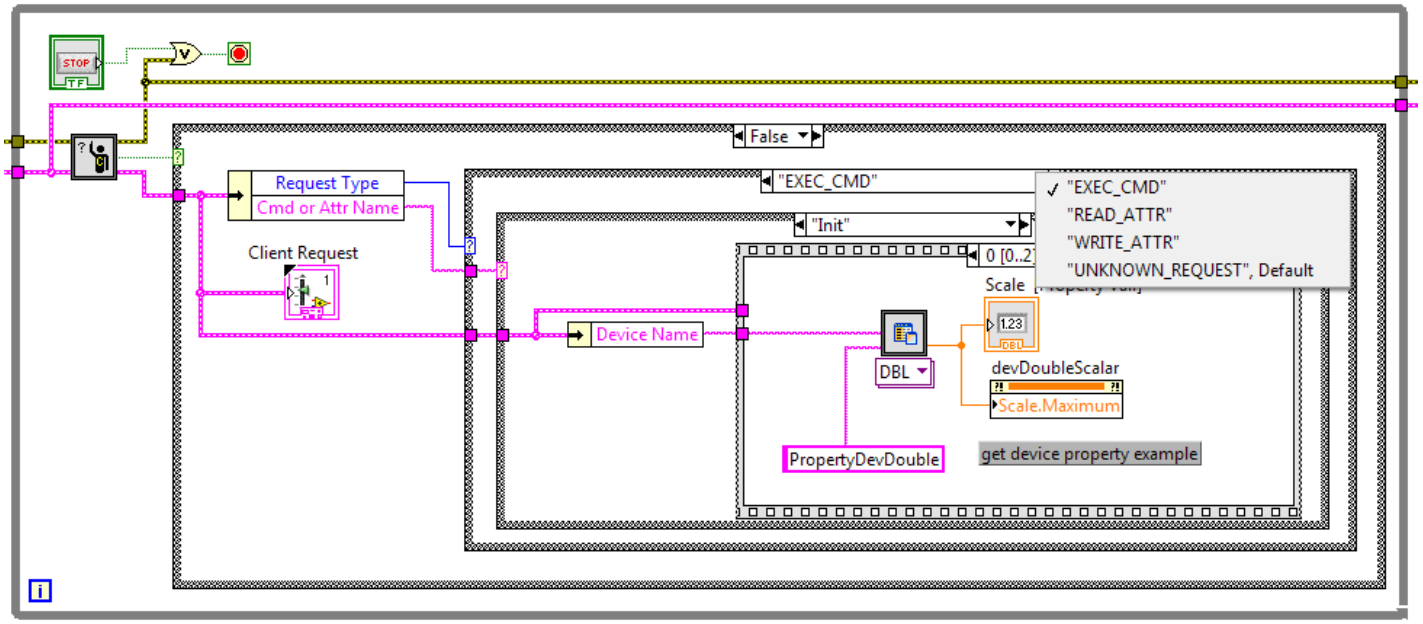


The *dserver evt.* terminal of the *_DServerStart.vi* is then injected into the ***_DServerEventHandler.vi*** which is itself running into an infinite while loop. The *_DServerEventHandler.vi* is the unique entry point of any ***client request***.



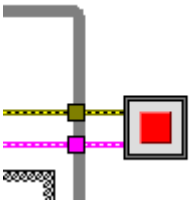


A client request is primarily identified by its type: ***EXEC_CMD***, ***READ_ATTR*** or ***WRITE_ATTR***. It also contains the device name and the name of the command or attribute to which it applies. Using case structures, it then becomes straightforward to identify and reply to client requests:



As we can see, client requests handling is basically based on 3 imbricated case structures. The first one is related to ***_DServerEventHandler.vi*** timeout. Here we choose to do nothing in case the timeout of the underlying event handler expires (i.e. no idle activity here). In case we get a valid request for the event handler, we unbundle it to access both the ***request type*** and the ***command or attribute name*** **-** which constitute respectively the second and third level of the client request handling. Can't be simpler…

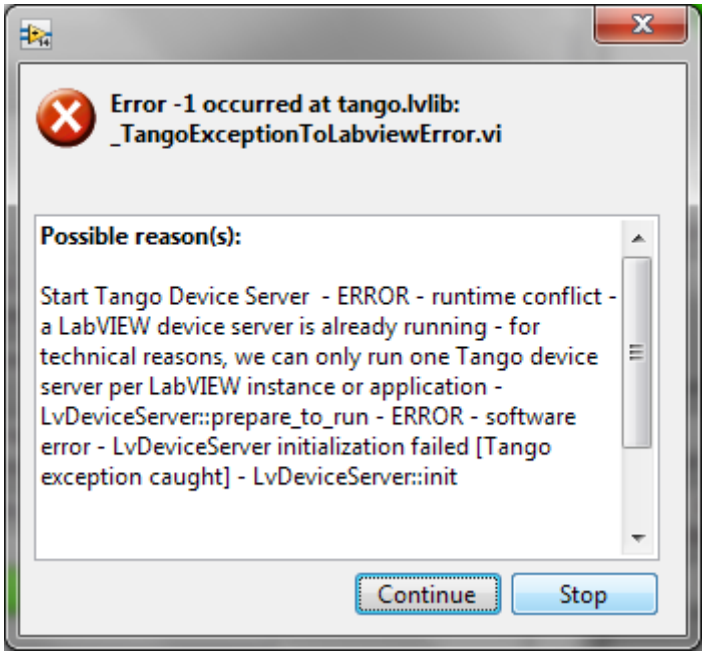
At the right end of the block diagram, we find a call to the ***_DServerStop.vi***. This VI stops the Tango device(s) activity and does the necessary cleanup.



Note: the Tango binding smoothly supports "brutal interruption" through the LabVIEW *abort* button. One can abort the device(s) activity, change his/her implementation then restart the application using the *start* button.

In fact, whatever is the way the device-server is aborted on LabVIEW side, the devices will actually still be running on the C++ side. Incoming requests will be simply trashed and an exception will be thrown to the client. This behavior is imposed by the Tango kernel itself. The main consequence is that, during the development stage of our application, we will have to quit/restart LabVIEW in case we want to start another device-server.

Trying to start a device-server while another one is already running will generate the following error:



Don't be confused. Running several devices into the same device-server is supported – see the ***examples.lvlib::MultiDeviceDServer.vi*** for an example. What is not supported is running two device-servers into the same LabVIEW session (or application). This is also a Tango kernel constraint inherited from the C++ implementation – i.e. running two device-servers into the same process is not supported (which clearly makes sense).

Replying to client requests

Every client request is handled at C++ level by a dedicated thread. This thread forwards the incoming request by posting a *user-event* to the LabVIEW side of the device implementation. It then waits for a reply or, at least, an acknowledgment. In case the LabVIEW implementation takes too long to reply, the predefined timeout will expire and an error will be returned to the client. In such a case, the late reply or acknowledgement coming from LabVIEW will be silently trashed by the binding (can't do anything else in fact).

It's important to note that the thread handling a given request will block while waiting for the reply and will prevent any other request from being handled (due the Tango serialization model). We consequently understand that it's a good idea to always reply to a client request in order to avoid blocking the device for several seconds each time an "unhandled request" is received.

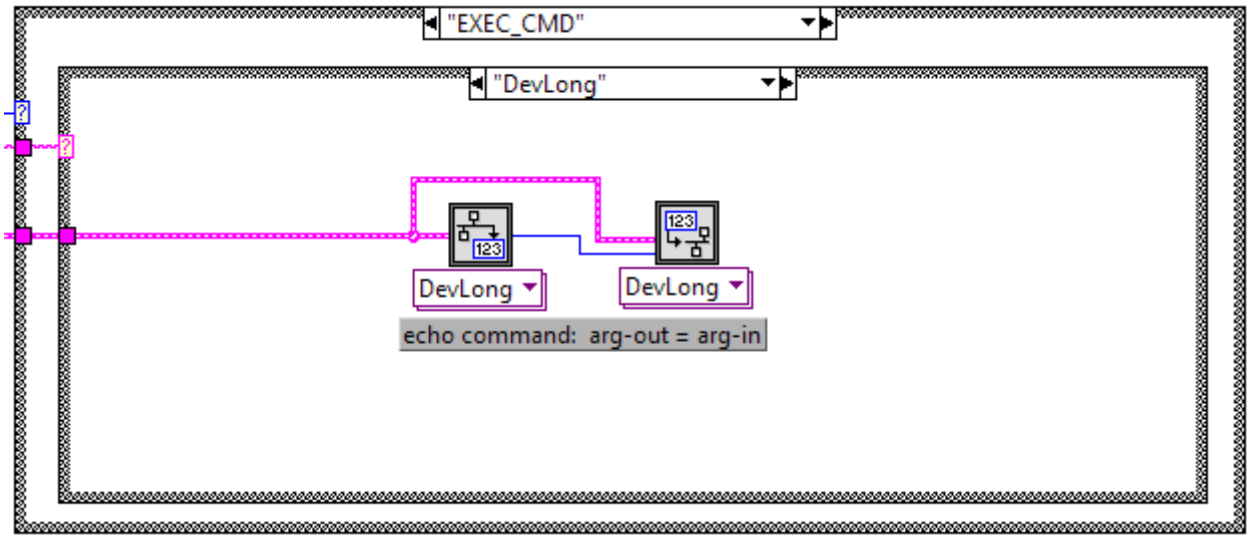
The ***default timeout*** value ***is 3 seconds***. This value can be changed at class or device level using dedicated VIs: ***_DServerSetClassTimeoutInSec.vi*** and ***_DServerSetDeviceTimeoutInSec.vi***.

Specific VIs are provided for valid requests acknowledgement and unimplemented (i.e. not supported) requests. See the ***DevVoid*** and ***default*** cases of the ***EXEC_CMD*** case structure for an example.

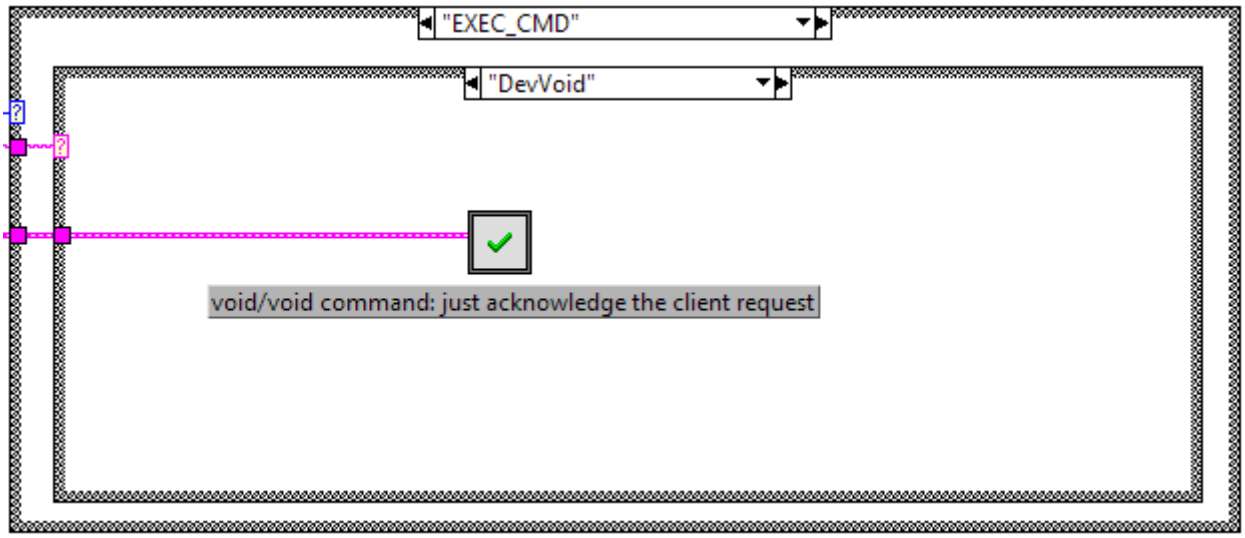
Replying to a command execution request

In case of a command execution request, the input argument (*argin*) is delivered with the request. Unless, the argin data type is *DevVoid*, the polymorphic ***_DServerGetCommandArgin.vi*** is used to access the associated data. Symmetrically, unless the output argument (argout) of the command is *DevVoid*, the polymorphic ***_DServerSetCommandArgout.vi*** will be used to send the reply to the client. In case the argout data is *DevVoid*, we will simply acknowledge the request using the ***_DServerRequestAcknowledgement.vi***.

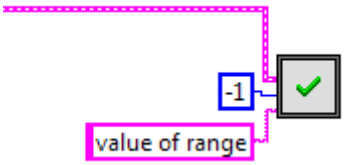
Here is the implementation of the ***DevLong/DevLong*** echo like command of our example device:



The ***DevVoid/DevVoid*** case is even simpler:



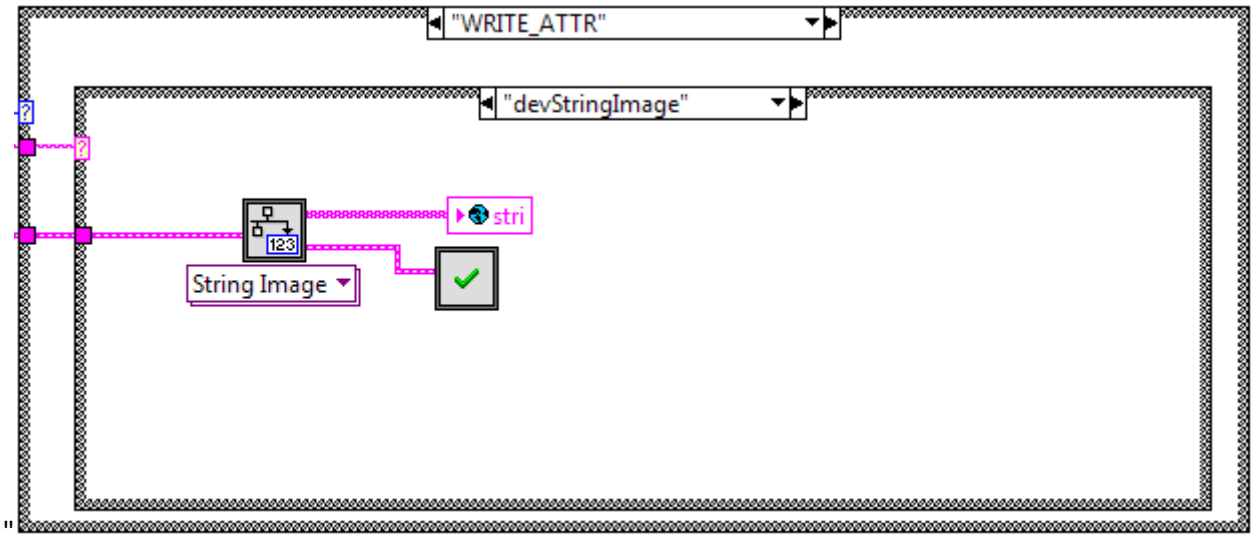
Please note that, whatever is the request type, the *_DServerRequestAcknowledgement.vi* can be used to send an error to the client - e.g. invalid argument, value out of range, etc.



Replying to a write attribute request

Replying to a "write attribute" request is a two stages process. The attribute value (i.e. the setpoint) is first extracted using the polymorphic ***_DServerGetAttributeValue.vi***. Then, the request is acknowledged (or rejected) using the *_DServerRequestAcknowledgement.vi*.

Here is the implementation of the "write *devStringImage*" case of our example. The setpoint is extracted in the expected format then pushed into a global variable named *stri*. In parallel, we also acknowledge the request.

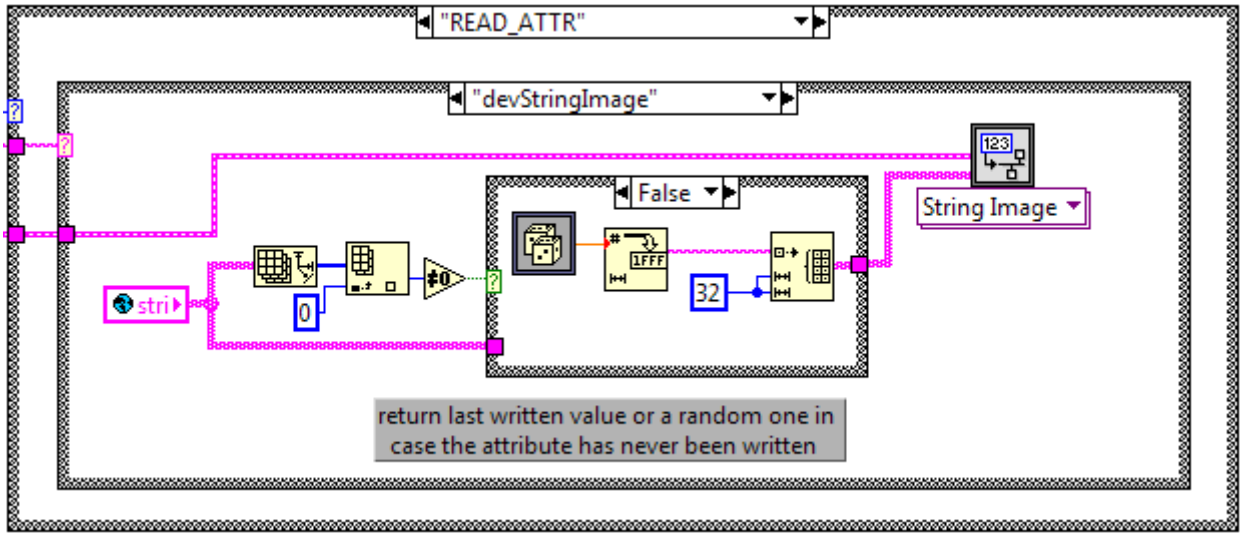


Note: in real life application, it would have been a good idea to validate the extracted attribute value before acknowledging the request.

Replying to a read attribute request

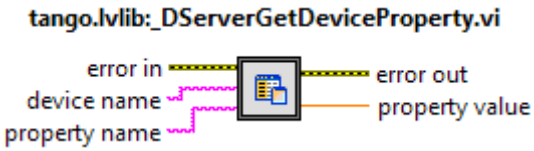
Here, the game is simple: we have to provide the current value of the requested attribute. Whatever is the way we store our data in the LabVIEW application (global, queue, …), the idea is to push it into the polymorphic ***_DServerSetAttributeValue.vi*** specifying the associated attribute data type (the system can't guess it).

Here is the "read *devStringImage*" case of our example device. In this example, we choose to return the last written value or a random one in case the attribute as never been written. Here again, we could return an error to the client - e.g. hardware in fault, no data available, etc. - using the *_DServerRequestAcknowledgement.vi*.

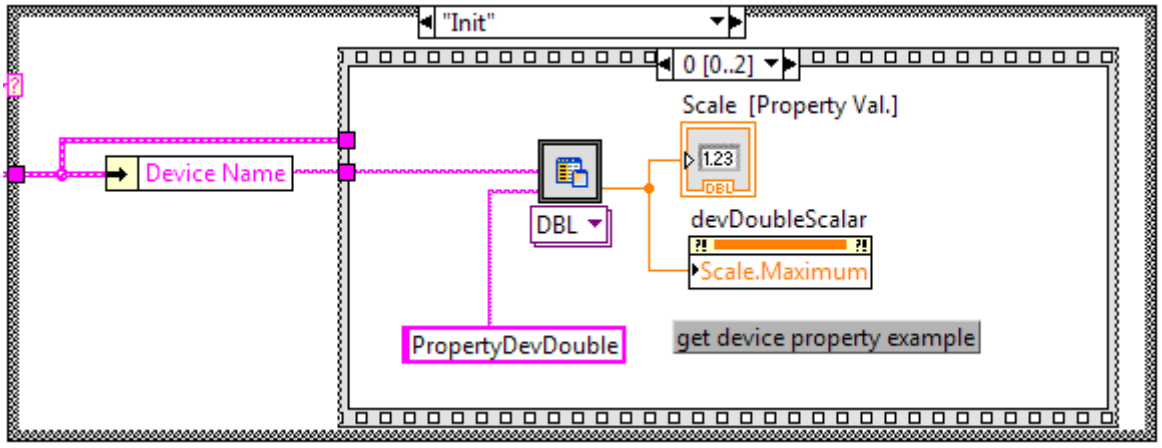


Accessing device property

A dedicated VI is provided to access a device property, the ***_DServerGetDeviceProperty.vi***.



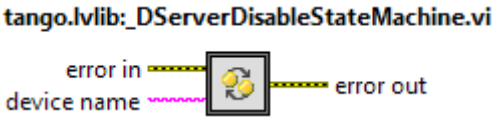
Device properties are usually read upon reception of an "*Init*" command request. Here is an example:



About the device state machine

The Tango binding provides a default implementation of the state machine you define in POGO. The idea is to systematically read and check the *State* attribute of the device before forwarding the request to the LabVIEW implementation. If the requested action is not allowed for the current device *State*, it's aborted and an error is returned to the client.

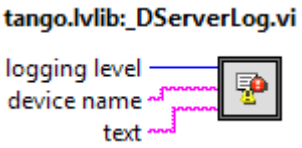
In this case you want to handle the *State* machine yourself, the default behavior can be disabled using the ***_DServerDisableStateMachine.vi***.



Symmetrically, it can be re-enabled using the ***_DServerEnableStateMachine.vi.***

Device Logging

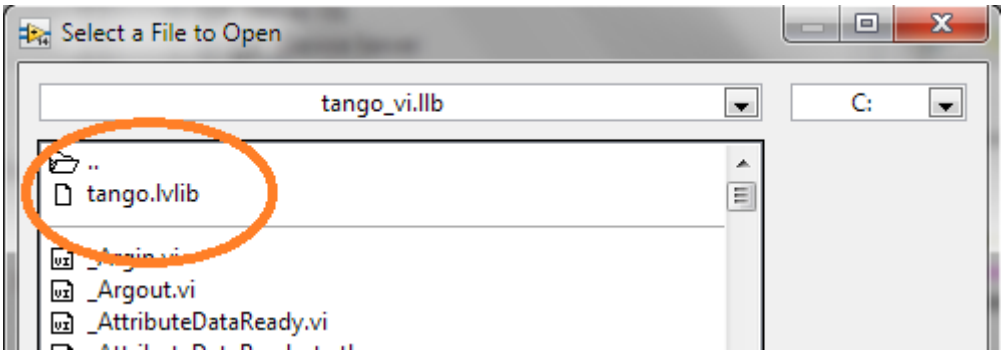
Finally, logging on behalf of a given tango device is supported. Here is the associated VI:

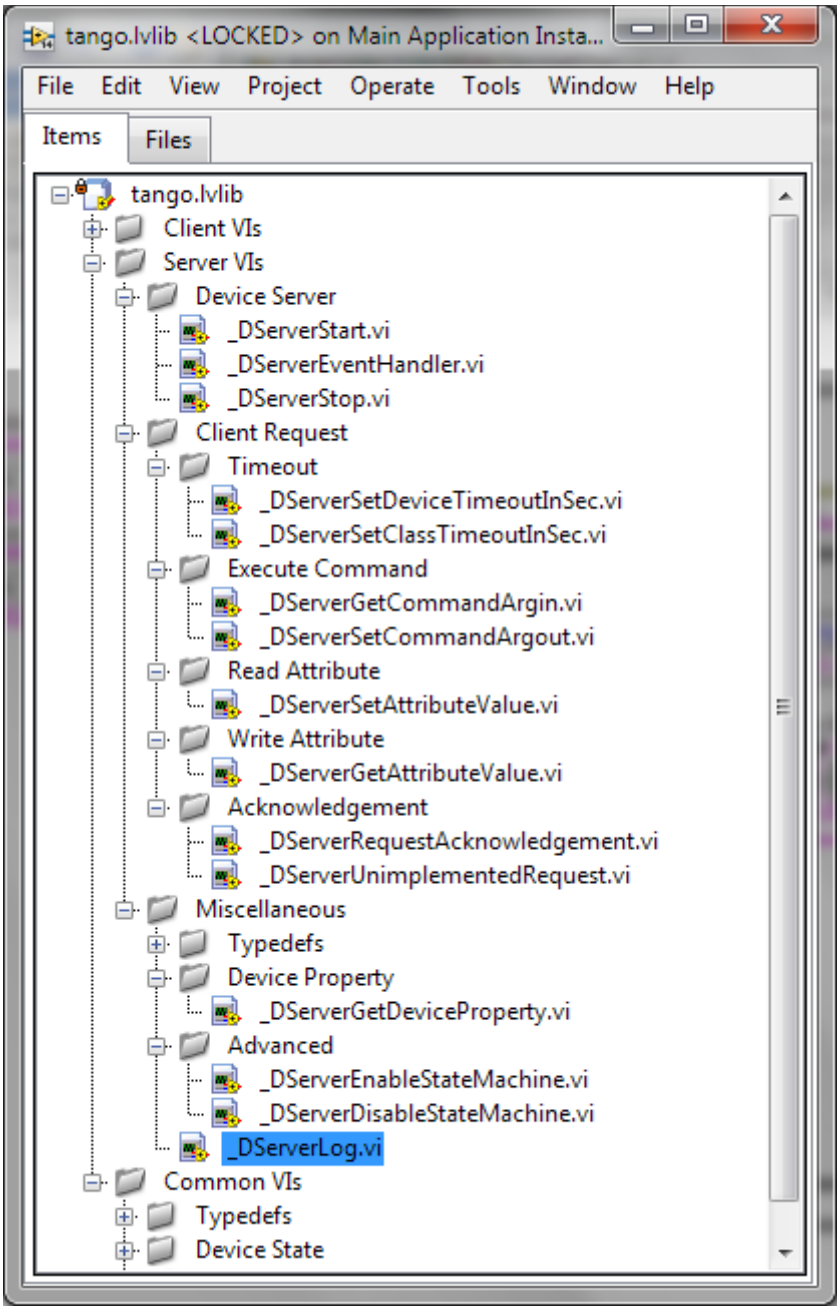


The logging level is controlled by the ***tango.lvlib::_LoggingLevel.ctl*** (typedef) .

The tango.lvlib

The VIs we just described are available from the ***tango.lvlib***. A library which is itself embedded into the *tango llb*: ***TBFL_DIR/vis/tango.llb.*** Not that this library also contains the ***Client VIs*** - which can obviously be used simultaneously with the Server VIs.


Select a File to Open
tango_vi.llb
C:
..
tango.lvlib
_Argout.vi
_AttributeDataReady.vi


tango.lvlib <LOCKED> on Main Application Insta...
File Edit View Project Operate Tools Window Help
Items
Files
tango.lvlib
Client VIs
Server VIs
Device Server
_DServerStart.vi
_DServerEventHandler.vi
_DServerStop.vi
Client Request
Timeout
_DServerSetDeviceTimeoutInSec.vi
_DServerSetClassTimeoutInSec.vi
Execute Command
_DServerGetCommandArgin.vi
_DServerSetCommandArgout.vi
Read Attribute
_DServerSetAttributeValue.vi
Write Attribute
_DServerGetAttributeValue.vi
Acknowledgement
_DServerRequestAcknowledgement.vi
_DServerUnimplementedRequest.vi
Miscellaneous
Typedefs
Device Property
_DServerGetDeviceProperty.vi
Advanced
_DServerEnableStateMachine.vi
_DServerDisableStateMachine.vi
_DServerLog.vi
Common VIs
Typedefs
Device State