Introduction to INDIGO 2.0

First of all… why another standard?

Yes, there are other initiatives for astronomy software development.

Probably the most important is ASCOM, nevertheless it is Windows only standard without any clear vision of distributed computing support and locked to .Net technologies.

Another one is INDI with opposite approach, is easy to port to Unix based systems, but is poorly standardised in some aspects, very inefficient for large amount of data and published under restrictive open-source license avoiding real commercial use and thus wider industry acceptance.

That's why we begun development of INDIGO framework with the goal to take the best from the both worlds – community driven, portable, distributed, efficient, easy to use for developers and invisible for end-users.

Why version 2.0 at the very beginning? It can use legacy INDI wire protocol version 1.7 as a fall down option and thus INDIGO protocol version started on 2.0. And unlike in INDI, INDIGO wire protocol version is coupled with the rest of framework.

Oh you asked just why the name INDIGO? It is colour #4B0082. Any similarity to other framework, living or dead, is purely coincidental ☺

What are design requirements and limitations?

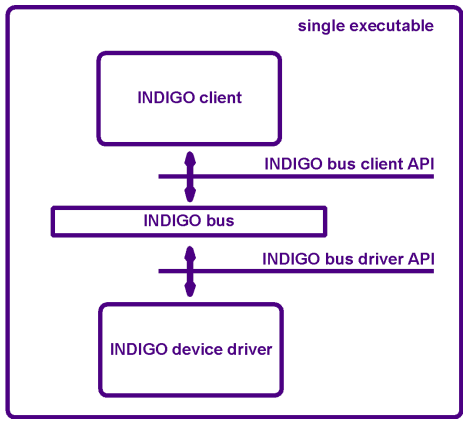
This is current list of requirements taken into consideration:

1. No GPL/LGPL code use to allow commercial use under application stores licenses.
2. ANSI C for portability and to allow simple wrapping into .Net, Java, GO, Objective-C or Swift.
3. Layered architecture to allow both direct linking of the drivers into applications or distributed use.
4. Atomic approach to device drivers. E.g. if camera has imaging and guiding chip, driver should expose two independent simple devices instead of one complex. It is much easier and transparent for client.
5. Drivers should support hot-plug at least for USB devices. If device is connected/disconnected while driver is running, its properties should appear/disappear on the bus.

What is the high-level architecture of INDIGO powered application?

INDIGO is actually a kind of software bus. There are processes in "device" role connected from "device" side and processes in "client" role connected from "client" side. Processes exchange messages modelled according INDI standard (not just because of backward compatibility, but also because it is proven good practise) over the bus and process them.

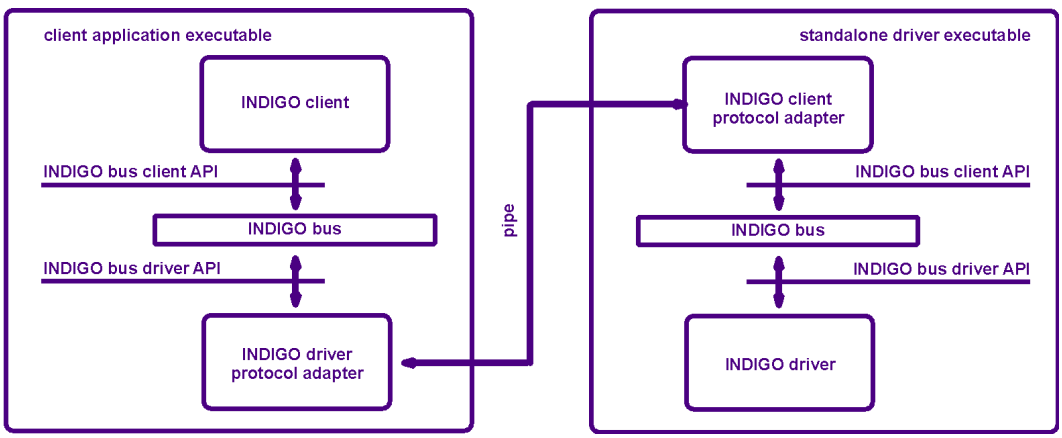
In the simplest case, on “device” side of INDIGO bus is a single device driver and on “client” side an application using this driver, all linked into single executable and operating on raw speed (only pointer to raw data is transmitted over INDIGO bus instead of base64 encoding/decoding and wire protocol transfer in INDI or type marshalling in ASCOM).



In the more complicated case, application can be divided into separated client and driver executables.

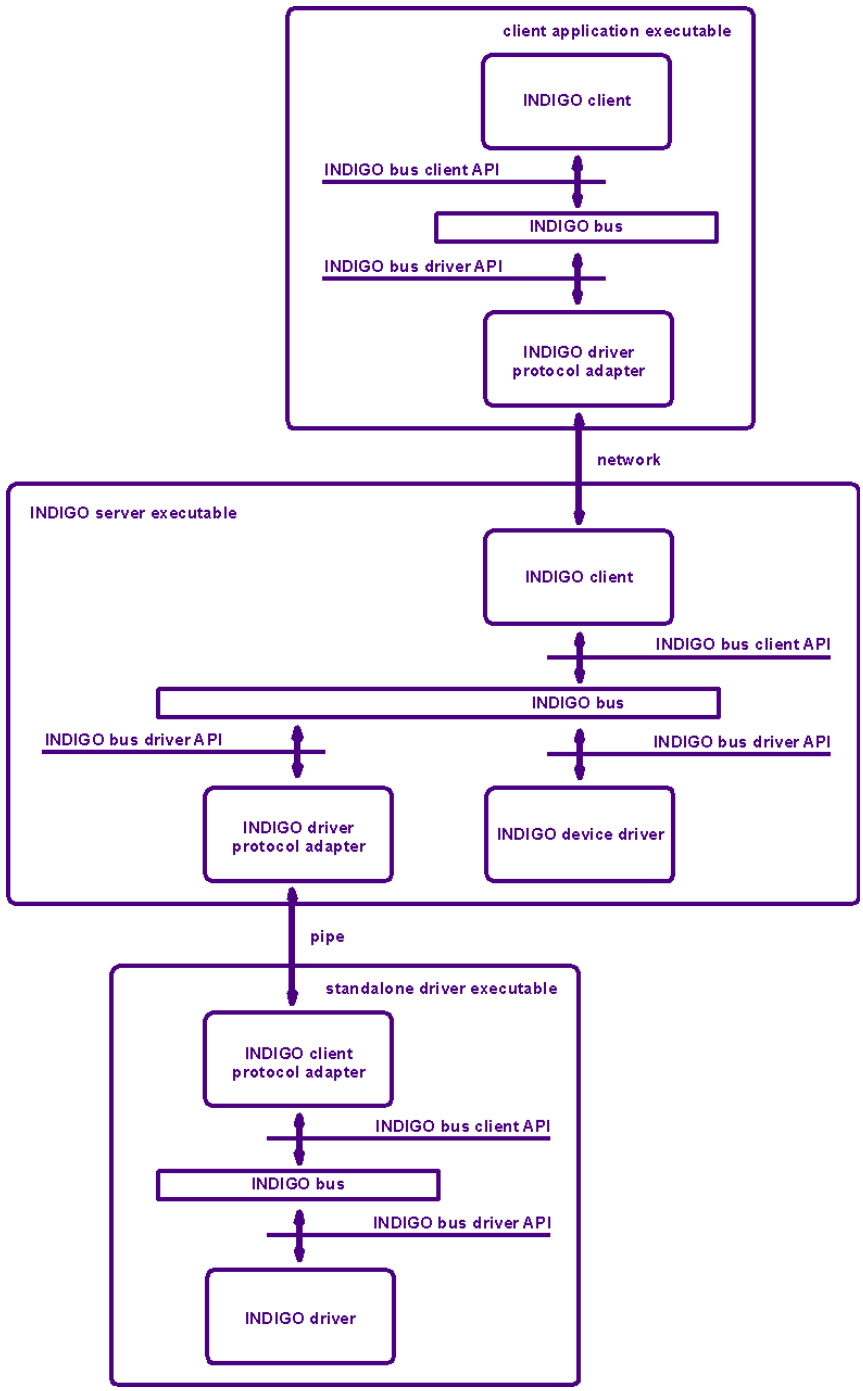
In the client executable, INDIGO wire protocol adapter is in the role of “device” and on “device” side and actual application code again on “client” side of INDIGO bus. Client executable can spawn the driver executable and communicate with it over pipes. In this case driver can be selected on run-time. Also vendor provided driver executable or legacy INDI driver can be used.

In the driver executable, INDIGO wire protocol adapter is in the role of “client” on “client” side and the device driver is again on “device” side of INDIGO bus.



In the most generic case one or more drivers, linked-in or standalone executables and INDIGO or INDI, can be used by specialised application called INDIGO server (similar to INDI server with the exception of much more efficient linked-in driver option). Client application will discover one or more INDIGO servers on the network, connect over wire protocol adapters on client and server side and use remote devices the same way as if they are linked-in.

INDIGO wire protocol has the ability to fall down to legacy INDI wire protocol 1.7 to maintain backward compatibility with INDI drivers. In this mode also property and item names are mapped to INDI standard.



There are infinite possibilities how to combine real or virtual device drivers, client or server side protocol adapters and real or virtual clients sharing common design principles and messaging API.

How INDIGO bus API looks like?

INDIGO bus is described by “indigo_bus.h” header file. It defines INDIGO property, related enumerations and types and bus, client and device driver API function prototypes.

INDIGO property represents a physical feature of the device (e.g. coordinates of a slew target of the telescope mount or status of camera cooler) or operation on the device (e.g. slew to the target or trigger camera exposure). It is a container for specified number items of the same type (e.g. RA and Dec part of equatorial coordinates). It is modelled according to the property in legacy INDI protocol.

Each property is related to a particular device, has the name, human readable group and label or description, state, type, permission, switch rule (optional for switch messages) and version (to allow mapping between different names or different behaviour in different versions of the standard).

The state of a property can be

- “idle” if property is passive or unused,
- “ok” if property is in correct state or if operation on property was successful,
- “busy” if property is transient state or if operation on property is pending and
- “alert” if property is in incorrect state or if operation on property failed.

The type of a property and property item can be

- “text” for strings of limited width,
- “number” for float numbers with defined min, max values and increment,
- “switch” for logical values representing “on” and “off” state,
- “light” for status values with four possible values “idle”, “ok”, “busy” and “alert” and
- “blob” for binary data of any type and any length.

The permission of a property can be “read only”, “read write” and “write only” with obvious meaning.

The switch property can have also switch rule defining dependencies between values of its items. Possible values are

- “one of many” and “at most one” for radio button group like behaviour and
- “any of many” for checkbox button group like behaviour.

Properties are defined in the “device” and used by “client”, pieces of code participating in bus communication and defined by structure containing among other pointers to functions called by the bus if requested by other side.

In case of “device”, function pointers for the following events need to be defined:

- “attach” called when device is attached to the bus (e.g. on plug-in event),
- “enumerate properties” called when client request the list of available properties of the device (the list can actually differ according of the state of the device),
- “update property” called when client request the change of particular property (e.g. to set target temperature of CCD) and

- “detach” called when device is detached from the bus (e.g. on un-plug event).

In case of “client”, function pointers for the following events need to be defined:

- “attach” called when client is attached to the bus (e.g. handler process for remote client is created),
- “define property” called when device broadcast the definition of a new property (e.g. in reply to “enumerate properties” request or if connection state of device changed),
- “update property” called when device broadcast the value change of a property (e.g. if current temperature of CCD changed),
- “delete property” called when device broadcast removal of a property (e.g. if device is disconnected or its state changed).
- “send message” called when device broadcast a message (e.g. for debugging purposes).
- “detach” called when client is detached from the bus (e.g. when remote client closes the connection).

“Device” or “client” can be attached to or detached from the bus by calling function defined in bus API. Once connected, “device” can “define property”, “update property”, “delete property” or “send message”, while “client” can request “property enumeration” or “property update”. Property used in “property enumeration” request may have undefined device and/or property name, in this case all devices should response and/or all properties should be defined.

Among function pointers, both “device” and “client” structures contain pointer to device or client context (private data used by the code), result of last bus request and version (together with property version used for name transformation and/or specific request processing).