INDI: Instrument-Neutral Distributed Interface

Protocol Version 1.7

Document Version 1.3

Abstract

A simple XML-like communications protocol is described for interactive and automated remote control of diverse instrumentation. INDI is small, easy to parse and stateless. In the INDI paradigm each Device poses all command and status functions in terms of setting and getting Properties. Each Property is a vector of one or more named members. Each Property provides timing information about how it might be sequenced with respect to other Properties to accomplish one coordinated action and provides hints as to how it might be displayed for interactive manipulation in a GUI. Clients learn the Properties of a particular Device at runtime using introspection. This decouples Client and Device implementation histories. Devices have complete authority over whether to accept commands from Clients. INDI accommodates intermediate servers, broadcasting, and connection topologies ranging from one-to-one on a single system to many-to-many between systems of different genre. The INDI protocol can be nested within other XML elements such as RTML to add constraints for automatic scheduling and execution.

Introduction

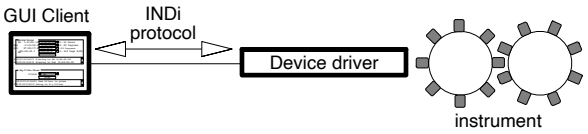
Modern astronomical instrumentation places high demand on its software infrastructure. Mounts, detectors and auxiliary equipment have many control points and configurations vary greatly across facilities and over time, creating an atmosphere of constantly changing software. Yet in the midst of this change users want consistent local and remote real-time access, freedom to use their favorite computing platforms, and ways to participate in the creation and use of clever automated scheduling techniques for efficient utilization of equipment and reduced operating costs. Keeping everything synchronized involves at least significant configuration management effort if not completely new development projects as these factors change.

This paper presents an architecture that eliminates the need for client software modifications when equipment or usage patterns change. The key feature is a communication protocol which allows a facility to describe itself in terms of devices in sufficient detail so they can be operated from a GUI for real-time command purposes or captured in a request file for later automated execution. This dynamic discovery process means that, except for the lowest level device drivers, none of the software components need to be rebuilt if and when the equipment being controlled changes. Conversely, clients may change as often as desired because they are not tied to particulars of the instrumentation. All networking and file formats use a simple subset of XML[1] which promotes freedom of choice for implementation languages and operating systems at each part of the system.

Other attempts involving XML for control purposes, such as AIML[2], differ from INDI in that they use XML to describe existing command and control channels. Our design uses XML directly for this purpose. Other attempts at capturing static observing requests in a file in XML format, such as the evolving RTML[3], intentionally restrict themselves to the control of canonical instruments. This prevents scheduling instruments with functionality that differs from its standardized conceptual framework without changing the standard. Our design captures commands for diverse and even esoteric instrumentation with no changes in tools whatsoever.

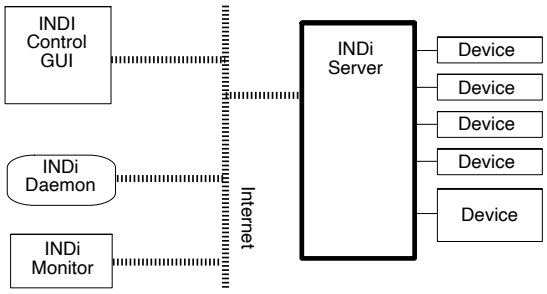
INDI Architecture

The diagram below represents the simplest possible INDI configuration: one Client connected to one Device:



The simplest INDI configuration

An INDI **Device** offers a service cast as a set of **Properties** in the INDI format. An INDI **Client** is a process which connects to an INDI Device, queries the Device for its set of control Properties and possibly send requests to change those Properties to the Device. Final authority for all instrument control rests with the Device. A Client may be a GUI which presents each command as a widget of some sort for displaying the current value of a command and possibly allowing the user to specify a new target value. A Client may be a process which never sends any changes to the Device, but monitors and logs all changes in Device state. A Client may be an automated command process that has an agenda of INDI commands and scans the agenda and decides suitable times to send new commands to the Device.



INDI with an intermediate Server

INDI Clients and Devices need not be in direct contact. The protocol is designed to accommodate arbitration and broadcasting among several Clients and Devices. For example, the diagram above shows an example of using an intermediary **Server** between INDI Clients and Devices. To each Client it appears to be a Device. To each Device it appears to be a Client. Servers can implement policies for administrative issues such as security, priority, graded access lists and other situations that arise when multiple Clients are in contact with multiple Devices.

INDI Properties

Properties are vectors of one of a small set of types. **Text** properties are collections of arbitrarily ordered characters. **Number** properties are numeric quantities and are sent with a printf-style format to recommend how a GUI should display them. **Switch** properties are always in a state of On or Off. Rules may be imposed on the behavior of the switches in a vector such as no more than one may be On at a time. **Lights** are properties that may be in one of the four states: Idle, OK, Busy or Alert. If a GUI displays this the suggested corresponding colors are gray, green, yellow and red, respectively. **BLOB** properties hold arbitrary binary large objects such as images.

Each Property has a **name** for identification purposes and a **label** for presentation purposes. Each element of the Property vector also has a name, making it in effect an associative array, and a presentation label. Changes to a Property effect all vector elements atomically.

All Property types except Lights have a **permission** attribute (lights are conceptually always read-only). Text and Number may be Read-Only, Write-Only or Read-Write; Switches may be Read-Only or Read-Write. Permission terminology is with respect to the Client but this does not bestow any true ability. Permission serves only as a hint to a Client as to whether a Device is potentially willing to allow a Property to be changed. If a Device reports a Property to be Read-Only it still must not trust the Client to comply but enforce the policy in case of rogue Clients that try to set a new value anyway. The permission hints allow Clients to treat Property values in sensible ways. For example a GUI Client may display the Property in a way that conveys whether it can be changed. For writable Properties, GUIs are encouraged to provide two fields: one that is passive and displays the last value received, and one that is interactive in which the user may type or otherwise modify a value without it being subject to spontaneous changes.

Each Property as a whole is always in one of four **states**: Idle, OK, Busy and Alert. If a GUI displays this in different colors, the colors suggested are gray, green, yellow and red, respectively. A Device is strongly encouraged to send an accompanying message whenever it informs a Client of a change of state. When a Client sends a command to change a Property, the Client shall henceforth consider the Property state to be Busy. When the Device accomplishes the associated action it sends a message that indicates the Property state has changed to, say, OK.

Each Property has a **timeout** value that specifies the worst-case time it might take to change the value to something else. The Device may report changes to the timeout value depending on current device status. Timeout values give Clients a simple ability to detect dysfunctional Devices or broken communication and also gives them a way to predict the duration of an action for scheduling purposes as discussed later.

Properties may be assembled into **groups** to suggest how Clients might organize them, for presentation purposes for example, but groups serve no functional purpose.

INDI Protocol

Each command between Client and Device specifies a **Device name** and **Property name**. The Device name serves as a logical grouping of several Properties. Property names must be unique per Device, and a Server must report unique Device names to any one Client.

The INDI protocol does not have the notion of query and response. A sender does not save context when it sends a command and wait for a specific response. A command may be sent for which a complementary command back is likely but all INDI participants must always be prepared to receive any command at any time. There is no notion of meta-errors such as illegally formatted commands, inappropriate commands or problems with the underlying communication mechanism. The proper response to all unusual or unexpected input is expressly to ignore any such problems (although some form of logging outside the scope of INDI might be judicious). With these rules the INDI protocol is small and simple; defines-away the possibility of deadlocks at the protocol level; automatically accommodates broadcasting; permits flexible and transparent routing; and eliminates the need for complex sequencing and synchronization.

When a Client first starts up it knows nothing about the Devices and Properties it will control. It begins by connecting to a Device or indiserver and sending the **getProperties** command. This includes the protocol version and may include the name of a specific Device and Property if it is known by some other means. If no device is specified, then all devices are reported; if no name is specified, then all properties for the given device are reported. The Device then sends back one **def*type*** element for each matching Property it offers for control, limited to the Properties of the specified Device if included. The def*type* element shall always include all members of the vector for each Property.

Note again that by sending a request for Property definitions the Client is not then waiting specifically for these definitions in reply. Nor is the Device obligated to supply these definitions in any order or particular time frame. The Client may learn of some Properties soon and perhaps others much later. The Client may also see messages for Properties about which it is as yet

unaware in which case the Client silently ignores the message. Thus a Client must have the ability to dynamically expand its collection of Properties at any time and gracefully ignore unexpected information.

To inform a Device of new target values for a Property, a Client sends one **new*type*** element. The Client must send all members of Number and Text vectors, or may send just the members that change for other types. Before it does so, the Client sets its notion of the state of the Property to Busy and leaves it until told otherwise by the Device. A Client *does not* poll to see whether the current reported values happen to agree with what it last commanded and set its state back to Ok on its own. This policy allows the Device to decide how close is close enough. The Device will eventually send a state of Ok if and when the new values for the Property have been successfully accomplished according to the Devices criteria, or send back Alert if they can not be accomplished with a message explaining why.

To inform a Client of new current values for a Property and their state, a Device sends one **set*type*** element. It is only required to send those members of the vector that have changed. In the case of Properties whose values change rapidly, the Device must insure that communication of the new values are not sent so often as to saturate the connection with the Client. For example, a socket implementation might send a new value only if writing to the socket descriptor would not block the process.

In order to allow for the likelyhood of requiring special efficiency considerations in the design of Clients to handle BLOB Properties, the element **enableBLOB** allows a Client to specify whether **setBLOB** elements will arrive on a given INDI connection. The Client may send this element with a value of **Only**, **Also** or **Never**. The default setting upon making a new connection is **Never** which means **setBLOB** elements will never be sent over said connection. If the Client sends **Only**, from then on only **setBLOB** elements shall be sent to the Client on said connection. If the Client sends **Also**, then all other elements may be sent as well. A Client may send the value of **Never** at any time to return to the default condition. This flow control facility allows a Client the opportunity, for example, to open a separate connection and create a separate processing thread dedicated to handling BLOB data. This behavior is only to be implemented in intermediate INDI server processes; individual Device drivers shall disregard **enableBLOB** and send all elements at will.

A Device may send out a **message** either as part of another command or by itself. When sent alone a message may be associated with a specific Device or just to the Client in general. Messages are meant for human readers and should be sent by a Device whenever any significant event occurs or target is reached. The INDI protocol syntax provides a means for a Device to time stamp each message and is encouraged to do so to maintain consistent time across all its Clients. If the Device does not include a time stamp for some reason (such as if it is a very simple device without local time-keeping capability) the Client is encouraged to add its own time stamp.

A Device may tell a Client a given Property is no longer available by sending **delProperty**. If the command specifies only a Device without a Property, the Client must assume all the Properties for that Device, and indeed the Device itself, are no longer available.

One Device may snoop the Properties of another Device by sending the **getProperties** command. The command may specify one Device and one Property, or all Properties for a Device or even all Devices, depending on whether the optional device and name attributes are given. Once specified, all messages from the matching Devices and Properties will be copied to the requesting Device as well.

Protocol Version Compatability

Version 1.7 added getProperties from Devices to add snooping functionality, and added the name attribute to getProperties from Clients to increase specificity. These changes are compatable with all prior versions. There was no change in the Protocol DTD itself from 1.5 to 1.6, just a clarification in the documentation regarding when all members of a vector must be transmitted or just those members that changed. The only change from 1.4 to 1.5 is making the size attribute of the oneBLOB element required instead of optional. Thus 1.5 Clients remain compatable with 1.4 Drivers in all respects if the Drivers happened to transmit BLOB size. Version 1.4 is compatable with 1.3 in all respects except for incompatable changes in the **oneBLOB** element. Version 1.3 is the same as 1.2 in all respects with the addition of the BLOB elements for transfering binary data. Thus, 1.2, 1.3, 1.4 and 1.5 Clients and Drivers may be freely intermixed, to the extent they do not use BLOB elements.

INDI XML Syntax Specification

This section defines the exact XML syntax of each INDI command and its behavior as part of the INDI protocol. A snippet of annotated DTD is given for each command. This section ends with some examples.

The DTDs make use of the following Entities:

```
<!ENTITY % propertyState "(Idle|Ok|Busy|Alert)" >
<!ENTITY % switchState "(Off|On)" >
<!ENTITY % switchRule "(OneOfMany|AtMostOne|AnyOfMany)" >
<!ENTITY % propertyPerm "(ro|wo|rw)" >
<!ENTITY % numberValue "(#PCDATA)" >
<!ENTITY % numberFormat "(#PCDATA)" >
<!ENTITY % labelValue "(#PCDATA)" >
<!ENTITY % nameValue "(#PCDATA)" >
<!ENTITY % textValue "(#PCDATA)" >
<!ENTITY % timeValue "(#PCDATA)" >
<!ENTITY % groupTag "(#PCDATA)" >
<!ENTITY % BLOBlength "(#PCDATA)" >
<!ENTITY % BLOBformat "(#PCDATA)" >
<!ENTITY % BLOBenable "(Never|Also|Only)" >
```

All PCDATA shall use character set ISO 8651-1[4].

The format of a numberValue shall be any one of integer, real or sexagesimal; each sexagesimal separator shall be any one of space( ), colon (:) or semicolon (;); each sexagesimal component specified shall be integer or real; unspecified components shall default to 0; negative values shall be indicated with a leading hyphen (-). For example, the following are all the same numeric value: "-10:30:18", "-10 30.3" and "-10.505".

A numberFormat shall be any string that includes exactly one printf-style format specification appropriate for C-type double or one INDI style "m" to specify sexagesimal in the form "%<w>.<f>m" where

```
<w> is the total field width
<f> is the width of the fraction. valid values are:
   9  ->  :mm:ss.ss
   8  ->  :mm:ss.s
   6  ->  :mm:ss
   5  ->  :mm.m
   3  ->  :mm
```

For example:

```
to produce     use

  "-123:45"    %7.3m
"  0:01:02"    %9.6m
```

A timeValue shall be specified in UTC in the form YYYY-MM-DDTHH:MM:SS.S. The final decimal and subsequent fractional seconds are optional and may be specified to whatever precision is deemed necessary by the transmitting entity. This format is in general accord with ISO 8601[5] and the Complete forms defined in W3C Note "Date and Time Formats"[6].

Commands from Device to Client

In the following descriptions, permission is always with respect to the Client.

```
Command to enable snooping messages from other devices. Once enabled, defXXX and setXXX messages
for the Property with the given name and other messages from the device will be sent to this
driver channel. Enables messages from all devices if device is not specified, and all Properties
for the given device if name is not specified. Specifying name without device is not defined.
<!ELEMENT getProperties EMPTY >
<!ATTLIST getProperties
  device %nameValue; #IMPLIED              device to snoop, or all if absent
  name %nameValue; #IMPLIED                property of device to snoop, or all if absent
>
```

```
Define a property that holds one or more text elements.
<!ELEMENT defTextVector (defText+) >
<!ATTLIST defTextVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  label %labelValue; #IMPLIED               GUI label, use name by default
  group %groupTag; #IMPLIED                 Property group membership, blank by default
  state %propertyState; #REQUIRED           current state of Property
  perm %propertyPerm; #REQUIRED             ostensible Client controlability
  timeout %numberValue; #IMPLIED            worse-case time to affect, 0 default, N/A for ro
  timestamp %timeValue #IMPLIED             moment when these data were valid
  message %textValue #IMPLIED               commentary
>

  Define one member of a text vector
  <!ELEMENT defText %textValue >
  <!ATTLIST defText
   name %nameValue; #REQUIRED               name of this text element
   label %labelValue; #IMPLIED              GUI label, or use name by default
  >

Define a property that holds one or more numeric values.
<!ELEMENT defNumberVector (defNumber+) >
<!ATTLIST defNumberVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  label %labelValue; #IMPLIED               GUI label, use name by default
  group %groupTag; #IMPLIED                 Property group membership, blank by default
  state %propertyState; #REQUIRED           current state of Property
  perm %propertyPerm; #REQUIRED             ostensible Client controlability
  timeout %numberValue; #IMPLIED            worse-case time to affect, 0 default, N/A for ro
  timestamp %timeValue #IMPLIED             moment when these data were valid
  message %textValue #IMPLIED               commentary
>

  Define one member of a number vector
  <!ELEMENT defNumber %numberValue >
  <!ATTLIST defNumber
   name %nameValue; #REQUIRED               name of this number element
   label %labelValue; #IMPLIED              GUI label, or use name by default
   format %numberFormat; #REQUIRED          printf-style format for GUI display
    min %numberValue; #REQUIRED             minimal value
   max %numberValue; #REQUIRED              maximum value, ignore if min == max
   step %numberValue; #REQUIRED             allowed increments, ignore if 0
  >

Define a collection of switches. Rule is only a hint for use by a GUI to decide a suitable
presentation style. Rules are actually implemented wholly within the Device.
<!ELEMENT defSwitchVector (defSwitch+) >
<!ATTLIST defSwitchVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  label %labelValue; #IMPLIED               GUI label, use name by default
  group %groupTag; #IMPLIED                 Property group membership, blank by default
  state %propertyState; #REQUIRED           current state of Property
  perm %propertyPerm; #REQUIRED             ostensible Client controlability
  rule %switchRule; #REQUIRED               hint for GUI presentation
  timeout %numberValue; #IMPLIED            worse-case time, 0 default, N/A for ro
  timestamp %timeValue #IMPLIED             moment when these data were valid
  message %textValue #IMPLIED               commentary
>

  Define one member of a switch vector
  <!ELEMENT defSwitch %switchState >
  <!ATTLIST defSwitch
   name %nameValue; #REQUIRED               name of this switch element
   label %labelValue; #IMPLIED              GUI label, or use name by default
  >

Define a collection of passive indicator lights.
<!ELEMENT defLightVector (defLight+) >
<!ATTLIST defLightVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  label %labelValue; #IMPLIED               GUI label, use name by default
  group %groupTag; #IMPLIED                 Property group membership, blank by default
  state %propertyState; #REQUIRED           current state of Property
  timestamp %timeValue #IMPLIED             moment when these data were valid
  message %textValue #IMPLIED               commentary
>
```

```
  Define one member of a light vector
  <!ELEMENT defLight %propertyState >
  <!ATTLIST defLight
   name %nameValue; #REQUIRED               name of this light element
   label %labelValue; #IMPLIED              GUI label, or use name by default
  >

Define a property that holds one or more Binary Large Objects, BLOBs.
<!ELEMENT defBLOBVector (defBLOB+) >
<!ATTLIST defBLOBVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  label %labelValue; #IMPLIED               GUI label, use name by default
  group %groupTag; #IMPLIED                 Property group membership, blank by default
  state %propertyState; #REQUIRED           current state of Property
  perm %propertyPerm; #REQUIRED             ostensible Client controlability
  timeout %numberValue; #IMPLIED            worse-case time to affect, 0 default, N/A for ro
  timestamp %timeValue #IMPLIED             moment when these data were valid
  message %textValue #IMPLIED               commentary
>

  Define one member of a BLOB vector. Unlike other defXXX elements, this does not contain an
  initial value for the BLOB.
  <!ELEMENT defBLOB EMPTY >
  <!ATTLIST defBLOB
   name %nameValue; #REQUIRED               name of this BLOB element
   label %labelValue; #IMPLIED              GUI label, or use name by default
  >

Send a new set of values for a Text vector, with optional new timeout, state and message.
<!ELEMENT setTextVector (oneText+) >
<!ATTLIST setTextVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  state %propertyState; #IMPLIED            state, no change if absent
  timeout %numberValue; #IMPLIED            worse-case time to affect a change
  timestamp %timeValue #IMPLIED             moment when these data were valid
message %textValue #IMPLIED                 commentary
>

Send a new set of values for a Number vector, with optional new timeout, state and message.
<!ELEMENT setNumberVector (oneNumber+) >
<!ATTLIST setNumberVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  state %propertyState; #IMPLIED            state, no change if absent
  timeout %numberValue; #IMPLIED            worse-case time to affect a change
  timestamp %timeValue #IMPLIED             moment when these data were valid
  message %textValue #IMPLIED               commentary
>

Send a new set of values for a Switch vector, with optional new timeout, state and message.
<!ELEMENT setSwitchVector (oneSwitch+) >
<!ATTLIST setSwitchVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  state %propertyState; #IMPLIED            state, no change if absent
  timeout %numberValue; #IMPLIED            worse-case time to affect a change
  timestamp %timeValue #IMPLIED             moment when these data were valid
  message %textValue #IMPLIED               commentary
>

Send a new set of values for a Light vector, with optional new state and message.
<!ELEMENT setLightVector (oneLight+) >
<!ATTLIST setLightVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  state %propertyState; #IMPLIED            state, no change if absent
  timestamp %timeValue #IMPLIED             moment when these data were valid
  message %textValue #IMPLIED               commentary
>

Send a new set of values for a BLOB vector, with optional new timeout, state and message.
<!ELEMENT setBLOBVector (oneBLOB+) >
<!ATTLIST setBLOBVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  state %propertyState; #IMPLIED            state, no change if absent
  timeout %numberValue; #IMPLIED            worse-case time to affect a change
  timestamp %timeValue #IMPLIED             moment when these data were valid
  message %textValue #IMPLIED               commentary
>

Send a message associated with a device or entire system.
<!ELEMENT message >
```

```
<!ATTLIST message
  device %nameValue; #IMPLIED               considered to be site-wide if absent
  timestamp %timeValue #IMPLIED             moment when this message was generated
  message %textValue #IMPLIED               commentary
>

Delete the given property, or entire device if no property is specified.
<!ELEMENT delProperty >
<!ATTLIST delProperty
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #IMPLIED                 entire device if absent
  timestamp %timeValue #IMPLIED             moment when this delete was generated
  message %textValue #IMPLIED               commentary
>

Send a message to specify state of one member of a Light vector
<!ELEMENT oneLight %propertyState >
<!ATTLIST oneLight
  name %nameValue; #REQUIRED                name of this light element
>
```

Commands from Client to Device

```
Command to ask Device to define all Properties, or those for a specific Device or specific
Property, for which it is responsible. Must always include protocol version.
<!ELEMENT getProperties EMPTY >
<!ATTLIST getProperties
  version %nameValue; #REQUIRED             protocol version
  device %nameValue; #IMPLIED               name of Device, or all if absent
  name %nameValue; #IMPLIED                 name of Property, or all if absent
>

Command to control whether setBLOBs should be sent to this channel from a given Device. They can
be turned off completely by setting Never (the default), allowed to be intermixed with other INDI
commands by setting Also or made the only command by setting Only.
<!ELEMENT enableBLOB %BLOBenable >
<!ATTLIST enableBLOB
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #IMPLIED                 name of BLOB Property, or all if absent
>

Commands to inform Device of new target values for a Property. After sending, the Client must set
its local state for the Property to Busy, leaving it up to the Device to change it when it sees
fit.

<!ELEMENT newTextVector (oneText+) >        send new target text values
<!ATTLIST newTextVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  timestamp %timeValue #IMPLIED             moment when this message was generated
>

<!ELEMENT newNumberVector (oneNumber+) >    send new target numeric values
<!ATTLIST newNumberVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  timestamp %timeValue #IMPLIED             moment when this message was generated
>

<!ELEMENT newSwitchVector (oneSwitch+) >    send new target switch states
<!ATTLIST newSwitchVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  timestamp %timeValue #IMPLIED             moment when this message was generated
>

<!ELEMENT newBLOBVector (oneBLOB+) >        send new target BLOBs
<!ATTLIST newBLOBVector
  device %nameValue; #REQUIRED              name of Device
  name %nameValue; #REQUIRED                name of Property
  timestamp %timeValue #IMPLIED             moment when this message was generated
>
```

Elements describing a vector member value, used in both directions.

```
One member of a Text vector
<!ELEMENT oneText %textValue >
<!ATTLIST oneText
  name %nameValue; #REQUIRED                name of this text element
```

```
>

One member of a Number vector
<!ELEMENT oneNumber %numberValue >
<!ATTLIST oneNumber
  name %nameValue; #REQUIRED                name of this number element
>

One member of a switch vector
<!ELEMENT oneSwitch %switchState >
<!ATTLIST oneSwitch
  name %nameValue; #REQUIRED                name of this switch element
>

One member of a BLOB vector. The contents of this element must always be encoded using base64[7].
The format attribute consists of  one or more file name suffixes, each preceded with a period,
which indicate how the decoded data is to be interpreted. For example .fits indicates the decoded
BLOB is a FITS[8] file, and .fits.z indicates the decoded BLOB is a FITS file compressed with
zlib[9]. The INDI protocol places no restrictions on the contents or formats of BLOBs but at
minimum astronomical INDI clients are encouraged to support the FITS image file format and the
zlib compression mechanism. The size attribute indicates the number of bytes in the final BLOB
after decoding and after any decompression. For example, if the format is .fits.z the size
attribute is the number of bytes in the FITS file. A Client unfamiliar with the specified format
may use the attribute as a simple string, perhaps in combination with the timestamp attribute, to
create a file name in which to store the data without processing other than decoding the base64.
<!ELEMENT oneBLOB %BLOBValue >
<!ATTLIST oneBLOB
  name %nameValue; #REQUIRED                name of this BLOB element
  size %BLOBlength; #REQUIRED               number of bytes in decoded and uncompressed BLOB
  format %BLOBformat; #REQUIRED             format as a file suffix, eg: .z, .fits, .fits.z
>
```

Example Messages from Device to Client

Define a read-write numeric field whose valid range is -100 to +100 in steps of 10, with initial value 50:

```
<defNumberVector device="OTA" name="Focus" state="Idle" perm="rw" timeout="50"
    label="Focus position, μM"
  <defNumber name="Focus" label="" format="%4.0f" min="-100" max="100" step="10">50</defNumber>
</defNumberVector>
```

Define a read-write vector with two members, one for RA and one for Dec:

```
<defNumberVector device="Mount" name="EQUATORIALJ2000_COORD" state="Idle" perm="rw" timeout="50"
    label="J2000 Equatorial Position"
  <defNumber name="RA"  label="RA H:M:S"  format="%11.8m"  min="0"   max="24">0</defNumber>
  <defNumber name="Dec" label="Dec D:M:S" format="%9.6m  " min="-90" max="90">0</defNumber>
</defNumberVector>
```

Define a collection of switches for various binning settings, default setting 2:1:

```
<defSwitchVector device="Camera" name="Binning" rule="OneOfMany" state="Ok" perm="w" timeout="0"
    label="Binning">
  <defSwitch name="One"   label="1:1">Off</defSwitch>
  <defSwitch name="Two"   label="2:1">On </defSwitch>
  <defSwitch name="Three" label="3:1">Off</defSwitch>
  <defSwitch name="Four"  label="4:1">Off</defSwitch>
</defSwitchVector>
```

Define a set of lights that indicate the security level of several alarms around the building:

```
<defLightVector device="Building" name="Security" state="Ok" label="Building Alarms">
  <defLight name="Front" label="Front door"  >Ok</defLight>
  <defLight name="Back"  label="Back door"   >Ok</defLight>
  <defLight name="Dock"  label="Loading dock">Ok</defLight>
</defLightVector>
```

Inform Client the current value of a switch has changed, assuming the switch rule was OneOfMany. Note how changes in both switches are reported, the switch change from On to Off preceding the one from Off to On and the Property state set to OK:

```
<setSwitchVector device="Camera" name="Binning" state="Ok" timestamp="2002-03-13T16:04:02"
message="Binning changed to 1:1">
  <oneSwitch name="Two">Off</oneSwitch>
  <oneSwitch name="One">On"</oneSwitch>
</setSwitchVector>
```

Send a generic system message:

```
<message timestamp="2002-03-13 12:00:00" message="It's 12 o'clock and all is well." />
```

Send a progress message about a device named Camera:

```
<message device="Camera" timestamp="2002-03-13 16:06:20" message="TEC is approaching target
temperature" />
```

Inform Client that the loading dock security alarm has been tripped:

```
<setLightVector device="Building" name="Security" state="Alert">
  <oneLight name="Dock">Alert</oneLight>
</setLightVector>
```

Request copies from now on of the Property named Now from the Device Environment:

```
<getProperties device="Environment" name="Now" />
```

Example Messages from Client to Device

Inform Device of a new target value for a OneOfMany switch Property. Note how only the switch coming On is reported. The Client sets its local notion of the Property state to Busy. The Device implements the "one of Many" discipline, not the Client. The Device will send back the command to turn the other switch Off and set the state back to OK when the command has been accomplished. This results in completely correct behavior and relieves the Client from being expected or even trusted to do this.

```
<newSwitchVector device="Camera" name="Binning">
  <oneSwitch name="Four">On</oneSwitch>
</newSwitchVector>
```

Send new target value for RA and Dec *atomically:*

```
<newNumberVector device="Mount" name="EQUATORIALJ2000_COORD">
  <oneNumber name="RA" >10:20:30</oneNumber>
  <oneNumber name="Dec">40:50:60</oneNumber>
</newNumberVector>
```

INDI Network Behavior

INDI is a session layer protocol. This means it *presumes* the existence of a reliable sequenced byte stream between each INDI participant. The transport mechanism for the INDI protocol may be anything that satisfies these requirements. It might be direct driver calls, local pipes, fifos, a UNIX socket, a TCP socket perhaps secured with SSL[10] or tunneling through ssh[11], or the protocol might be the payload within a framework built upon P2P[12], JXTA[13], Jabber[14], XML-RPC[15] or SOAP[16]. If a straight TCP/IP Socket implementation is used, the IANA[17] has assigned INDI to tcp port 7624.

The following sections contain pseudocode that describe the required logic by which a player on an INDI network must operate. This code does not in any way attempt to address the application being accomplished by the Client or the equipment or service being operated by the Device. A key point to realize is conditions that arise that do not fit these templates, *ie,* the missing `else` clauses, are legitimately ignored although perhaps prudently logged.

Client Processing

```
   if receive <setXXX> from Device
     change record of value and/or state for the specified Property
   if receive <defProperty> from Device
     if first time to see this device=
       create new Device record
     if first time to see this device+name combination
       create new Property record within given Device
   if receive <delProperty> from Device
     if includes device= attribute
       if includes name= attribute
         delete record for just the given Device+name
```

```
      else
        delete all records the given Device
    else
      delete all records for all devices

  if Client wants to learn all Devices and all their Properties
    send <defProperties>

  if Client wants to change a Property value or state
    set State to Busy
    send <newXXX> with device, name and value

  if Client wants to query a Property's target value or state
    send <getTarValue> with device and name attributes
```

Device Processing

```
  if receive <newXXX> from Client
    if element contains acceptable device, name and value
      set new target value and commence
  if receive <getProperties> from Client
    if element contains recognized device
      send one <defProperty> for each name for specified device
    else if element contains no device attribute
      send one <defProperty for each name for each device
  if any Property's value changes, even as a result of a <newXXX>
    send <setXXX> specifying device+name with new value and state
```

Server Behavior Discussion

An INDI Server is really any processing steps involved with the transporting of INDI messages so long as it presents the behavior of a Client to all Devices and a Device to all Clients. In its simplest form each command the Server receives from any Device might be sent unaltered to all Clients and each command the Server receives from any Client is sent unaltered to all Devices. The efficiency of this simple Server may be improved in several ways. An efficient server could send each command received from a Client to only the Device known to be responsible for the `device` attribute in the command. The Server could also snoop the `getProperties` commands from Clients and send setXXX commands only to Devices known to be interested. An INDI Server might cache setXXX commands and use them to respond to getProperties commands without resorting to extra Device queries (this is an important consideration when implementing scripting, described later).

INDI Servers must take special precautions to deal with large BLOB Properties. They must maintain and honor the `enableBLOB` state for each Client connection. Servers are allowed to drop BLOBs if they arrive faster than slow recipients can accept them. Servers must take care not to block while writing large BLOBs to slow Clients thus starving traffic to faster Clients.

It is possible to imagine an INDI server that blocks a Client from sending newXXX messages, even to otherwise writable Properties, based on an authentication strategy. This would open the possibility of a multiuser environment with some users having more control ability than others, perhaps leaving lower priority users with only read-only visibility into the INDI Properties at a facility.

It is possible to build INDI Servers such that they can be chained together, by virtue of the fact that Server-Device traffic is (very nearly) the same as Client-Server traffic. Thus one Server could connect to another to gain access to a Device, rather than run the Driver locally. This allows the INDI network to be distributed and take advantage of multiprocessing.

Servers might send all `newXXX` commands they receive from one Client connection to all other connected Clients. If Clients choose to support receiving `newXXX` commands, they might interpret them to mean update the editable field of a writable Property in the Client GUI. Thus, when both Client and Server support this activity, it allows Clients to mirror the commands other Clients have issued.

Scripting

INDI Clients need not be GUI programs, they can also be command line programs. For example a program could be written to get and display one or more INDI Properties specified on its command line. A program could set INDI Properties based on

command line arguments. A program could be written to accept a boolean expression of INDI Properties on its command line then connect to an INDI server and wait for the Properties to cause the expression to evaluate to true. These sorts of programs could be put together to perform complex functionality within the convenience of a scripting language.

Automatic Scheduled Operation with INDI

So far we have discussed using INDI to control a system in real-time. The INDI self-describing paradigm can also be used to operate any system automatically. Two new XML elements are defined for the purposes of timing the actions performed. For concreteness we discuss an automated telescope application.

Scheduling involves two more components:

- a GUI that displays the Devices and Properties of a site, adds target and constraint parameters and saves the entire description to a request file;
- an INDI client serving as a scheduling dæmon that is in an infinite loop: scanning all requests to select the next observation and communicating with the site Devices to accomplish the work.

The schedule preparation GUI could look very much like the real-time GUI, perhaps even be the same program. It certainly shows each Device and Property in much the same way. But it also contains a second section which allows the user to define a targeting scheme, timing requirements and to place constraints on the observation.

Observation timing might be specified as a particular Date and Time, or JD, with an acceptable error margin; be specified as a cyclical event with reference, period and phase timing window specifications; or might not be specified at all and be left totally up to the scheduler. Whichever way the time is decided, it serves to define the reference start time of the observation.

Other constraints might be a required such as maximum seeing spread or an acceptable range of Moon phase. In the former case, the scheduler could know to ask the Device for the current value of a read-only property which reports the seeing from, say, a DIMM[18] device. In the latter case, the daemon consults its own lunar ephemeris and knowledge of the location of the site.

To capture the timing of events comprising one observation, two more elements are added to the XML INDI protocol. An INDI `new`*`type`* command can be wrapped in a **by** element with an attribute **t** that specifies with respect to the start of the observation the number of seconds the Property should achieve status OK. At runtime the Scheduler can determine the time is takes to achieve a given target value using the timeout parameter for the Property.

An INDI `new`*`type`* command can also be wrapped in an **at** element, also with an attribute **t**. This tells the scheduler Client the exact moment when the `new`*`type`* should be issued. It can be used with any Property but makes most sense when used with ones whose timeout value is 0, for example a `switch` that takes no time to accomplish such as a camera shutter control.

The role of the Scheduler process is to keep the telescope busy. Whenever the telescope is not busy, the scheduler scans and ranks all observing requests still to be performed. The ranking includes local circumstances such as hour angle and imminent sun rise and set. Constraints are determined to be met by asking the Devices to report the current property values and status. When a request is finally chosen, it is executed by simply issuing the INDI property commands *exactly as they appear in the file*, timed according to their `at` or `by` attributes.

Operational specifications use the Devices and Properties for the observatory directly to record which commands are to be issued for the observation. The schedular will cancel the observation if any of these return other than `Busy` or `OK` status values. For read-only properties, the scheduler requires that the Devices return values that match those in the observation request before choosing that observation to execute.

The following is an example of one observation request. Each of the Constraints were specified by the user in the Schedule Builder GUI. The Operations were compiled using the exact Device and Property names returned to the Builder by the Devices for which the schedule was built. Note the ability to specify different property values to occur at different times; in this case, to implement a multiple exposure to determine the direction of any trailed objects in the image.

```
<INDIobservation>
 <moon>
   <minSeparation>20</minSeparation>
   <maxPercentLit>10</maxPercentLit>
 </moon>
 <timing method="phase">
   <phaseTiming>
     <phaseZeroTime>2345678.901</phaseZeroTime>
     <phasePeriod>3.45<phasePeriod>
     <phaseMin>.3</phaseMin>
     <phaseMax>.4</phaseMax>
   </phaseTiming>
 </timing>
 <repeat>
   <count>10</count>
   <minSep>1</minSep>
 </repeat>
  <INDI>
   <by t="0">
     <newNumberVector device="Monster Scope" name="EQUATORIALJ2000_COORD">
       <oneNumber name="RA">10:20:30</oneNumber>
       <oneNumber name="Dec">-4:5:6</oneNumber>
     </newNumber>
     <newTextVector device="OTA" name="Big-O Filters">
       <oneText name="setting">Red</oneText>
     </newTextVector>
   </by>
   <at t="0">
     <newSwitchVector device="Wonder Cam" name="Shutter">
       <oneSwitch name="open">On</oneSwitch>
     </newSwitchVector>
   </at>
   <at t="10">
     <newSwitchVector device="Wonder Cam" name="Shutter">
       <oneSwitch name="open">Off</oneSwitch>
     </newSwitchVector>
   </at>
   <at t="20">
     <newSwitchVector device="Wonder Cam" name="Shutter">
       <oneSwitch name="open">On</oneSwitch>
     </newSwitchVector>
   </at>
   <at t="60">
     <newSwitchVector device="Wonder Cam" name="Shutter">
       <oneSwitch name="open">Off</oneSwitch>
     </newSwitchVector>
   </at>
  </INDI>
  <Log>
  </Log
</INDIobservation>
```

As the Scheduler daemon executes each command for this request, the Log portion is filled in. Each entry in the Log is timestamped in format ISO 8601. This forms a permanent record of all actions and results in behalf of this observation.

Summary

INDI Clients know nothing about the meaning of the Devices and Properties with which they deal and Devices know nothing of how or why Clients are using their services. Devices present a standard protocol to all potential Clients, so they need only be written one time (for a given operating system) and thus are defined by and move with their equipment and not with their control programs. Clients control any Devices. A real-time GUI Client just displays Properties and lets the user change some of them. A schedule-creation GUI displays the Properties and saves their target values to an XML file containing `new`*`type`* elements along with additional tags to define constraints and timing. A scheduler daemon reads the file and copies the tags directly to the devices at the appropriate time.

Thus, GUIs, file formats and control daemons remain completely unchanged and drivers are only written once to be accessed from anywhere making INDI truly an *Instrument Neutral Distributed Interface*.

Protocoll Revision History	**Changes**
1.1	Initial release
1.2	Corrections
1.3	Add BLOBs
1.4	permit fraction seconds in timestamp; require encoding all BLOBs in base64
1.5	make **size** attribute in **oneBLOB** REQUIRED
1.6	clarify when to send all members of a vector or just those that changed
1.7	add getProperties message from Devices. Add name to getProperties from Clients.

Document Revision History	**Changes**
1.1	Started separate document version after Protocol Rev 1.6
1.2	Fix typos, no substantive changes
1.3	Add description of drivers snooping via getProperties. Remove candidate standardized property names.

Resources

---

1 http://www.w3.org/TR/2000/REC-xml-20001006

2 http://pioneer.gsfc.nasa.gov/public/aiml

3 http://monet.uni-sw.gwdg.de/twiki/bin/view/RTML/WebHome

4 http://www.htmlhelp.com/reference/charset/latin1.gif

5 http://www.cl.cam.ac.uk/~mgk25/iso-time.html

6 http://www.w3.org/TR/NOTE-datetime

7 http://www.faqs.org/rfcs/rfc2045.html §6.8, and sample code at http://www.fourmilab.ch/webtools/base64

8 http://fits.gsfc.nasa.gov

9 http://www.faqs.org/rfcs/rfc1951.html and http://www.zlib.org

10 http://home.netscape.com/eng/ssl3/draft302.txt, http://www.openssl.org

11 http://www.openssh.com

12 http://www.openp2p.com

13 http://www.jxta.org

14 http://www.jabber.org

15 http://xml-rpc.com

16 http://www.w3.org/TR/SOAP

17 http://www.iana.net

18 http://www.astro.washington.edu/rest/dimm