

# Remote API

September, 2018

## Overview

The Remote API is a network wrapper around the Plug-in API. The Remote API Plug-in acts as the server to clients that may reside on the same computer or on different computers. The Remote API is robust enough to include the full functionality of the Plug-in API, allowing users to develop network based or out-of-process SIMDIS plug-ins.

## Plug-in

The Remote API Server is a plugin that is compatible with SIMDIS 10, Plot-XY, and SIMDIS 9. Selecting the Remote API Server from the Plug-in menu displays the dialog below.

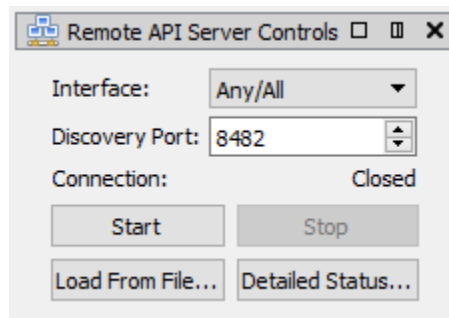


Figure 1: Remote API Server User Interface

The user specifies a Discovery Port and Interface using the GUI. This is used as a well-known address on which Remote API client applications can connect. Multiple clients can connect to the same server. Two additional ports are created automatically to service commands from the plug-in and reply with responses from SIMDIS. This is covered in detail in the Architecture section below.

Click the “Start” button to bind to the port indicated in the GUI. Clients will be able to connect at this point. Clicking “Stop” will close the socket on the Discovery Port and close all active connections. Note that the GUI does not need to remain open while clients communicate with the plug-in. The plug-in can be automatically started using command line options.

The Connection field is updated to indicate the current state of the server. Connection states include:

<b>Close</b>	Server is not currently active. No clients are connected.
<b>Open</b>	Server is currently active and bound to the discovery port. Clients may be connected.
<b>Invalid</b>	Server was unable to bind to the port provided.

Remote API Files are binary files containing protobuf messages from a client. The format is a series of serialized binary protobuf messages, preceded by a two-byte unsigned short integer of the message size. These messages can be captured per-client using the Detailed Status GUI, and can be loaded using the main window's "Load From File" button. The commands will be executed in sequence as though from a new client.

The "Detailed Status" button brings up the server status GUI:

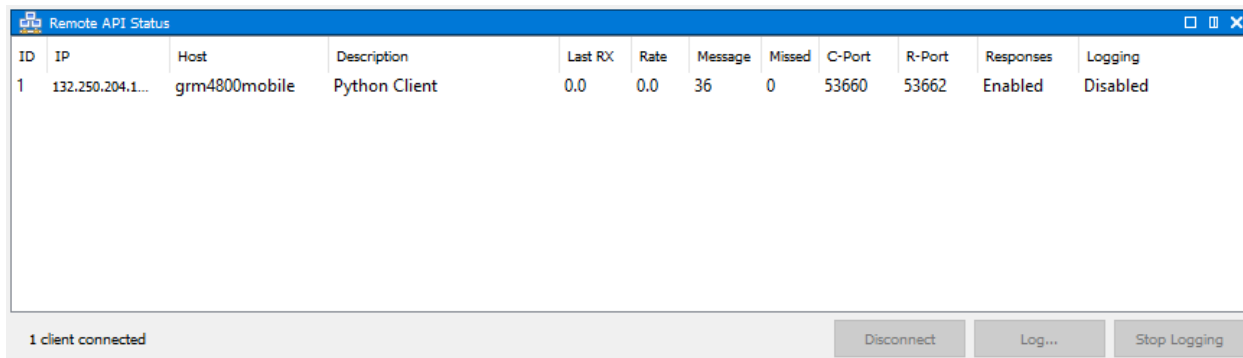


Figure 2: Remote API Status GUI

The window shows detailed status for each connected client. The user can enable and disable logging of messages to a binary file via the "Log.." and "Stop Logging" buttons. Clients can be forcibly disconnected via the "Disconnect" button. The columns include:

<b>ID</b>	Serial identifier assigned by the plug-in to the client.
<b>IP</b>	IP Address of the client, as supplied by the client itself in the discovery message.
<b>Host</b>	Hostname of the client, as supplied by the client itself in the discovery message.
<b>Description</b>	Text description of client, as supplied by client itself in the discovery message.
<b>Last RX</b>	Seconds elapsed since the last received message.
<b>Rate</b>	Calculated number of messages per second from the client.
<b>Messages</b>	Most recently received sequence ID from the client.
<b>Missed</b>	Number of commands detected to have missed from the client.
<b>C-Port</b>	Dedicated command port on which the client sends Remote API commands.
<b>R-Port</b>	Dedicated response port to reply to client with SIMDIS return values.
<b>Responses</b>	Indicates whether responses are enabled or disabled. Clients can opt to omit responses. Note that omitting responses does not close the response socket.
<b>Logging</b>	Indicates whether logging is currently enabled for the client connection.

The following command line options are supported by the plug-in:

<b>-RemoteAPI:autoStart</b>	Start Remote API Server.
<b>-RemoteAPI:iface &lt;arg&gt;</b>	Interface address for Remote API network sockets.
<b>-RemoteAPI:port &lt;arg&gt;</b>	Port for Remote API Discovery Service.

## Architecture

The Remote API is a client/server architecture, with the Remote API Plug-in acting as a server to multiple clients. The message communication flow diagram for a single client is shown below:

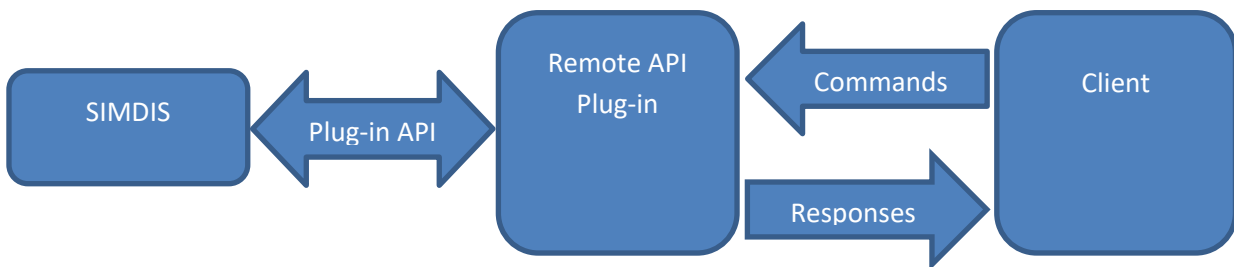


Figure 3: Remote API Flow Chart

The clients issue commands to the server and receive responses from the server. A client is analogous to a Plug-in and may be written in any language, including but not limited to C++, MATLAB (using MEX), and Python.

Remote API clients connect to a Discovery port opened by the plug-in and request a command and response port from the Remote API Plug-in. Additional metadata is sent at this time, including a description of the client. The plug-in then opens two new ports to communicate with the new client and replies with the port values. The client is then responsible for connecting to these ports. Clients always connect to the Remote API server, and the Remote API server plug-in always binds to ports to service clients. The underlying socket technology used is ZeroMQ. A network topology diagram is below:

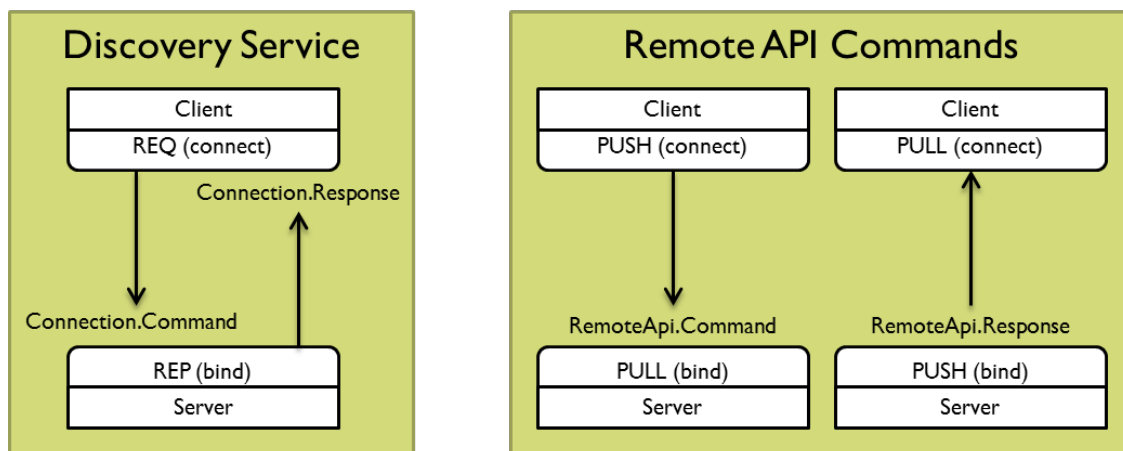


Figure 4: Remote API Network Topology

The communication between the server and the client is divided into the Message Format and the Communication Protocol. The Message Format is definition of the messages between the server and

the client while the Communication Protocol is how the messages are passed between the server and the client.

## Message Format

The Remote API uses protobuf messages to define the messages between the server and the client. Details on protobuf messages are at: <https://developers.google.com/protocol-buffers/> . The top level messages are defined in RemoteApi.proto. Messages are divided into Commands which are sent from the client to the server and Responses which go from the server to the client. Every command sent to the server will generate one and only one response. A command can only request one piece of information. Another words only one “has\_” should be valid for a command message.

The layout of the Remote API messages is modeled after the Plug-in API. The Remote API follows the same hierarchical structure as the Plug-in API. For details on most of the messages in the Remote API refer to the Plug-in API documentation at:

<https://simdis.nrl.navy.mil/codepages/doc/PluginAPI/PluginClient/doc/html/index.html>

Where the name of the Remote API message deviates from the expected Plug-in API name, the documentation in the protobuf file will reference the Plug-in API name so that searching the protobuf files with the Plug-in API name will locate the appropriate information.

## User-supplied Unique ID

Many of the Create messages and Add messages allow the client to specify the unique ID. This allows the client to add data to the newly created entity without having to wait for the response to the Create/Add message. It is the client’s responsibility to make sure the ID is unique. If the client provides a non-unique ID, the Remote Server will assign a unique ID. All the Create/Add messages that support a user-supplied unique ID have an optional field called “suppliedId”. The field is an unsigned 32-bit field. Setting the field to a non-zero value tells the Remote Server that the client is specifying the unique ID. Entities not created by the client or created by the client with a non-unique ID will have ID values greater than an unsigned 32-bit value. Once a client specifies a unique ID for an entity the client is **not** obligated to specify a unique ID of other entities. A client can simultaneously create entities with and without supplied unique IDs.

## Communication Protocol

It is necessary to control the flow of messages between the server and the client. A client can either use a Sequential approach or a Streaming approach. In the Sequential approach, a client issues a command and waits for the response. The command must be processed by the main thread of the host application (SIMDIS, Plot-XY) which results in a throughput rate of 50 messages or less per second. In the Streaming approach the client issues multiple commands without waiting for the response to the previous command. The client keeps track of what responses it is waiting for and handles the responses with they eventually arrive. The Streaming approach can have throughput rate of 1000s of messages per second. The client can pick either approach or mix them together. The message rate will be

influenced by the CPU load of the host application, the speed of the network, the amount of network traffic, the type of messages and the implementation of the client.

	Sequential	Streaming
Complexity	Relatively simple	More complex
Throughput	About 50 messages / second	Thousands of messages / second

**Table 1: Sequential versus Streaming**

When the client sends a message to the server the library returns a Sequence Number that is used to identify the response. The client periodically asks the library if the response with a given Sequence Number has arrived. When the response arrives, the client can then parse the response for any necessary information. The client should use the Sequence Number as an opaque value and should not attempt to interpret the value.

It is possible for a client to turn off all responses. The returning of responses is controlled by the `protobuf` field of `RemoteAPI.Configuration.Command.responseSetting`. Setting the field to `NO_RESPONSES` and sending it to the server will turn off all responses include the response to the message turning off responses. Setting the field to `ALL_RESPONSES` and sending the message to the server will turn on all responses starting with the response to the message turning on responses. Before sending the `ALL_RESPONSES` message the client MUST flush all pending responses.

A client can control the timeout used to declare a connect invalid. An invalid connection is defined as a connection without any messages for the duration of the timeout. The timeout is defined in seconds and defaults to 300 seconds. The server will immediately close a connection that becomes invalid. A timeout value of 0 means no timeout and the connection will never be declared invalid. The client can set the timeout using the `protobuf` field of `RemoteAPI.Configuration.Command`.

## Differences

Some notable differences between the Remote API and the Plug-in API interfaces include:

- All Remote API messages return either a `ReturnCodeOnlyResponse` message or the requested information (when responses are enabled).
  - A successful message returns either `RETURN_OK` or the requested information.
  - An unsuccessful message will always return a `ReturnCodeOnlyResponse` message with the appropriate fields set.
- Most names for the Remote API structs and enums are based off the Plug-in API names without the PI prefix.
- Where appropriate some Plug-in API commands have been expanded to support multiple points. Supporting multiple points will reduce the number of network calls. If one of the points

is invalid than processing of the command will terminate and the server will return an error message indicating the point in error. All points before the error will have been processed. Examples include PIPlotXY::getPairOption and PIPlotXY::setPairOption.

- Instead of using the term “Object” the Remote API uses the term “Entity”
- All ID types are now uint64.
- GetColor and GetStatus support a time range with a flag for initial condition. If the initial condition flag is set and there is no data point at the requested start time, then the routine will calculate the last value before the requested time range and return the value as the first value with the start time of the range. If no value exists before the requested start time than no initial value is returned.
- GetGeneric supports a time range with a flag for including the time span in the calculation. The flag set to false means only compare the time of the Generic data against the requested time range. The flag set to true means compare the time range of the Generic data against the requested time range; any overlap means include the Generic data. For a requested time range of 10 seconds to 20 seconds the following tables show different examples:

Time, Expire Time	Flag = True	Flag = False
1, -1	Include	Exclude
1, 3	Exclude	Exclude
1, 11	Include	Exclude
1, 30	Include	Exclude
11, 1	Include	Include
11, -1	Include	Include
21, 1	Exclude	Exclude
21, -1	Exclude	Exclude

- For data access, Tables now follow the pattern set by other Entity types with a GetRows message. Data access via Iterators and functors is not supported by the Remote API.
- Individual retrieval and setting of data have been replaced by single methods that handle all relevant data. Examples include:
  - All the information on a table can be retrieved by the single call of GetTableHeader. The individual calls for getting and setting table information will not be implemented.
  - The message getColumns is used to get all the column header information for all the columns. Individual access of column information is not supported by the Remote API.

## Support Requests

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF

LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Support requests are handled on the SIMDIS Help Desk at <https://simdis.nrl.navy.mil/jira>. Support is provided on a best effort basis, and support is minimal at best for individuals and organizations that are not active sponsors.