Table of Contents

Overview

Sometimes one desires the ability to change the procedure based on the results of incoming data. This could be as simple as retaking a single shot, or as complex as manipulating variables to alter a ramp shape and duration.

To enable this kind of communication, we've built a JSON-based interface to a dynamically chosen TCP port. The port is looked up by your client via the NI Service Locator. This interface is available since SetList version v2.0.0 [1)]

TCP Communications

Getting the Port: NI Service Locator

The NI Service Locator is available at port 3580 on any machine with an active LabView installation. Visiting the link http://localhost:3580/dumpinfo? takes you to a page listing the services it currently knows about. When SetList is running, it will create a named service `SetList/JSON`. The service locator will be informed and the port number will be available at http://localhost:3580/SetList/JSON on the computer running SetList. You should be able to access it from another computer by replacing `localhost` with the IP address of the computer running SetList.

The response is of the form `Port=<port #>` which should be easily parse-able by whatever software is initiating the feedback. This port is static over individual runs of `SetList.vi`, so it is wise to cache it and only look it up at startup or when communication fails.

Communication Protocol

Once you have the port number, you can initiate a TCP connection with that port on the SetList computer. SetList is expecting a JSON object string preceded by its length in binary. That is four bytes representing the integer number of bytes in the string being sent, followed by the string itself.

When it finishes processing the data you've sent, SetList will send a response (formatted as above) and then close the connection. If more information needs to be sent, open a new connection to the same port.

Expected JSON Contents

General Format

The string is expected to conform to the JSON format specification[2].

SetList parses the top level of the JSON object for its `members`. It assumes each member's name[3][4] is a string matching to some internal command. The `value` part of the pair is passed to the command as a string.

If a matching command is not found, SetList will respond with an error and skip that member.

SetList's response is formatted as a JSON object as well, usually containing an array for responses that are normal and a separate array for errors.

Variable Format

Representing a variable as a JSON object works as follows:[5]

```
{
  "name": <string (required)>,
  "defaultValue": <number>,
  "sequenceFunction": <string>,
  "informIgor": <bool>,
  "sequence": <bool>
}
```

Where:

- All members' names correspond to their function in the SetList Variable manager.
- The “`name`” member has a **required** string value.
- All other members may be given the value `null` (e.g. “`informIgor`”:`null`) to indicate the variable should retain the previous (or, if creating a new variable, the default) value.

Variable Set

Variables can also be grouped into “Variable Sets” using JSON arrays:[6]

```
[
  <variable 1>,
  <variable 2>,
  ...,
  <variable N>
]
```

Command Formats

Immediately Apply Variables

This command has SetList update the values of the variables right away, regardless of the status of a running sequence. The format is:

```
"instantVariables":<variable set>
```

where <variable set> is a single variable set object as specified above.

Sequence Set Variables

This command has SetList update the values of the variables after the end of the currently-running sequence. SetList maintains an ordered (FIFO) list of variable sets, and applies the next set at the end of a sequence just before re-starting the sequence.

The full command JSON is then:[7)]

```
"sequenceSets":[
  <variable set 1>,
  <variable set 2>,
  ...,
  <variable set N>
]
```

Where <variable set i> is a variable set as defined above.

Mulligans

Mulligans have the simplest format:

```
"mulligan":[<number>,...]
```

where <number> is the file number of an element you are trying to mulligan.

SetList will only check if the elements of the array are numbers, not if they are in the history (and thus mulligan-able). It reports the total count of numbers it finds, and sends errors for the non-numbers.

Example Clients

codes→Utils→ExternalFeedback→ExampleClients[8)] contains subdirectories with simple example clients for:

- Igor
- LabView
- Python

which should be enough of a starting point to either integrate into your data acquisition package or build a client in some other language.

Why TCP?

Because TCP libraries exist for many programming languages (Python 3.5, Python 2.7, C++, IGOR, MatLab)[9] in addition to LabView, this allows you to "close the loop" however you wish.

LabView has a built-in TCP Listener which waits for incoming connections on a specified port before handing the connection off to your program to handle. This allows us to do a low-overhead loop that just sits and waits for outside data or to be shutdown.

[1] Previously we used a custom protocol over fixed TCP ports to do this. It was too clunky given the existence of standard data exchange formats like JSON, so it has been discontinued. Documentation is still available on old_feedback.
[2] see also JSON
[3] The first part of a `name:value` pair, described in the spec as a `string:value` pair
[4] not to be confused for the value of a member named 'name'
[5], [6], [7] with optional white space added for legibility
[8] or on GitHub
[9] These are the results of a quick Google search, no guarantee implied