SiPyCo Documentation

Release 1.0

M-Labs

Dec 12, 2019

CONTENTS

SiPyCo (Simple Python Communications) is a library for writing networked Python programs. It was originally part of ARTIQ, and was split out to enable light-weight programs to be written without a dependency on ARTIQ.

CHAPTER

ONE

API DOCUMENTATION

1.1 sipyco.pyon module

This module provides serialization and deserialization functions for Python objects. Its main features are:

- Human-readable format compatible with the Python syntax.
- Each object is serialized on a single line, with only ASCII characters.
- Supports all basic Python data structures: None, booleans, integers, floats, complex numbers, strings, tuples, lists, dictionaries.
- Those data types are accurately reconstructed (unlike JSON where e.g. tuples become lists, and dictionary keys are turned into strings).
- Supports Numpy arrays.

The main rationale for this new custom serializer (instead of using JSON) is that JSON does not support Numpy and more generally cannot be extended with other data types while keeping a concise syntax. Here we can use the Python function call syntax to express special data types.

sipyco.pyon.**decode**(*s*)
Parses a string in the Python syntax, reconstructs the corresponding object, and returns it.

sipyco.pyon.**encode**(*x*, *pretty=False*)
Serializes a Python object and returns the corresponding string in Python syntax.

sipyco.pyon.**load_file**(*filename*)
Parses the specified file and returns the decoded Python object.

sipyco.pyon.**store_file**(*filename*, *x*)
Encodes a Python object and writes it to the specified file.

1.2 sipyco.pc_rpc module

This module provides a remote procedure call (RPC) mechanism over sockets between conventional computers (PCs) running Python. It strives to be transparent and uses *sipyco.pyon* internally so that e.g. Numpy arrays can be easily used.

Note that the server operates on copies of objects provided by the client, and modifications to mutable types are not written back. For example, if the client passes a list as a parameter of an RPC method, and that method `append()`s an element to the list, the element is not appended to the client's list.

class sipyco.pc_rpc.**AsyncioClient**
This class is similar to *sipyco.pc_rpc.Client*, but uses `asyncio` instead of blocking calls.

All RPC methods are coroutines.

Concurrent access from different asyncio tasks is supported; all calls use a single lock.

close_rpc()
Closes the connection to the RPC server.

No further method calls should be done after this method is called.

connect_rpc(*host*, *port*, *target_name*)
Connects to the server. This cannot be done in __init__ because this method is a coroutine. See *sipyco.pc_rpc.Client* for a description of the parameters.

get_local_host()
Returns the address of the local end of the connection.

get_rpc_id()
Returns a tuple (target_names, description) containing the identification information of the server.

get_selected_target()
Returns the selected target, or `None` if no target has been selected yet.

select_rpc_target(*target_name*)
Selects a RPC target by name. This function should be called exactly once if the connection was created with `target_name=None`.

class sipyco.pc_rpc.**AutoTarget**
Use this as target value in clients for them to automatically connect to the target exposed by the server. Servers must have only one target.

class sipyco.pc_rpc.**BestEffortClient**(*host*, *port*, *target_name*, *firstcon_timeout=1.0*, *retry=5.0*)
This class is similar to *sipyco.pc_rpc.Client*, but network errors are suppressed and connections are retried in the background.

RPC calls that failed because of network errors return `None`. Other RPC calls are blocking and return the correct value.

Parameters

- **firstcon_timeout** – Timeout to use during the first (blocking) connection attempt at object initialization.
- **retry** – Amount of time to wait between retries when reconnecting in the background.

close_rpc()
Closes the connection to the RPC server.

No further method calls should be done after this method is called.

class sipyco.pc_rpc.**Client**(*host*, *port*, *target_name=<class 'sipyco.pc_rpc.AutoTarget'>*, *timeout=None*)
This class proxies the methods available on the server so that they can be used as if they were local methods.

For example, if the server provides method `foo`, and `c` is a local *Client* object, then the method can be called as:

```
result = c.foo(param1, param2)
```

The parameters and the result are automatically transferred from the server.

Only methods are supported. Attributes must be accessed by providing and using "get" and/or "set" methods on the server side.

At object initialization, the connection to the remote server is automatically attempted. The user must call *close_rpc()* to free resources properly after initialization completes successfully.

Parameters

- **host** – Identifier of the server. The string can represent a hostname or a IPv4 or IPv6 address (see `socket.create_connection` in the Python standard library).
- **port** – TCP port to use.
- **target_name** – Target name to select. `IncompatibleServer` is raised if the target does not exist. Use *AutoTarget* for automatic selection if the server has only one target. Use `None` to skip selecting a target. The list of targets can then be retrieved using *get_rpc_id()* and then one can be selected later using *select_rpc_target()*.
- **timeout** – Socket operation timeout. Use `None` for blocking (default), `0` for non-blocking, and a finite value to raise `socket.timeout` if an operation does not complete within the given time. See also `socket.create_connection()` and `socket.settimeout()` in the Python standard library. A timeout in the middle of a RPC can break subsequent RPCs (from the same client).

close_rpc()
Closes the connection to the RPC server.

No further method calls should be done after this method is called.

get_local_host()
Returns the address of the local end of the connection.

get_rpc_id()
Returns a tuple (target_names, description) containing the identification information of the server.

get_selected_target()
Returns the selected target, or `None` if no target has been selected yet.

select_rpc_target(*target_name*)
Selects a RPC target by name. This function should be called exactly once if the object was created with `target_name=None`.

exception sipyco.pc_rpc.**IncompatibleServer**
Raised by the client when attempting to connect to a server that does not have the expected target.

class sipyco.pc_rpc.**Server**(*targets*, *description=None*, *builtin_terminate=False*, *allow_parallel=False*)
This class creates a TCP server that handles requests coming from *Client* objects (whether *Client*, *BestEffortClient*, or *AsyncioClient*).

The server is designed using `asyncio` so that it can easily support multiple connections without the locking issues that arise in multi-threaded applications. Multiple connection support is useful even in simple cases: it allows new connections to be be accepted even when the previous client failed to properly shut down its connection.

If a target method is a coroutine, it is awaited and its return value is sent to the RPC client. If `allow_parallel` is true, multiple target coroutines may be executed in parallel (one per RPC client), otherwise a lock ensures that the calls from several clients are executed sequentially.

Parameters

- **targets** – A dictionary of objects providing the RPC methods to be exposed to the client. Keys are names identifying each object. Clients select one of these objects using its name upon connection.
- **description** – An optional human-readable string giving more information about the server.

- **builtin_terminate** – If set, the server provides a built-in `terminate` method that unblocks any tasks waiting on `wait_terminate`. This is useful to handle server termination requests from clients.
- **allow_parallel** – Allow concurrent asyncio calls to the target's methods.

sipyco.pc_rpc.**simple_server_loop**(*targets*, *host*, *port*, *description=None*)
Runs a server until an exception is raised (e.g. the user hits Ctrl-C) or termination is requested by a client.

See *sipyco.pc_rpc.Server* for a description of the parameters.

1.3 sipyco.fire_and_forget module

class sipyco.fire_and_forget.**FFProxy**(*target*)
Proxies a target object and runs its methods in the background.

All method calls to this object are forwarded to the target and executed in a background thread. Method calls return immediately. Exceptions from the target method are turned into warnings. At most one method from the target object may be executed in the background; if a new call is submitted while the previous one is still executing, a warning is printed and the new call is dropped.

This feature is typically used by RPC clients to wrap slow and non-critical RPCs, when avoiding blocking is more important than reliability.

ff_join()
Waits until any background method finishes its execution.

1.4 sipyco.sync_struct module

This module helps synchronizing a mutable Python structure owned and modified by one process (the *publisher*) with copies of it (the *subscribers*) in different processes and possibly different machines.

Synchronization is achieved by sending a full copy of the structure to each subscriber upon connection (*initialization*), followed by dictionaries describing each modification made to the structure (*mods*, see *ModAction*).

Structures must be PYON serializable and contain only lists, dicts, and immutable types. Lists and dicts can be nested arbitrarily.

class sipyco.sync_struct.**ModAction**
Describes the type of incremental modification.

Mods are represented by a dictionary `m`. `m["action"]` describes the type of modification, as per this enum, serialized as a string if required.

The path (member field) the change applies to is given in `m["path"]` as a list; elements give successive levels of indexing. (There is no `path` on initial initialization.)

Details on the modification are stored in additional data fields specific to each type.

For example, this represents appending the value `42` to an array `data.counts[0]`:

```
{
    "action": "append",
    "path": ["data", "counts", 0],
    "x": 42
}
```

append = 'append'
Appends *x* to target list.

delitem = 'delitem'
Removes target's *key*.

init = 'init'
A full copy of the data is sent in *struct*; no *path* given.

insert = 'insert'
Inserts *x* into target list at index *i*.

pop = 'pop'
Removes index *i* from target list.

setitem = 'setitem'
Sets target's *key* to *value*.

class sipyco.sync_struct.**Notifier**(*backing_struct*, *root=None*, *path=[]*)
Encapsulates a structure whose changes need to be published.

All mutations to the structure must be made through the *Notifier*. The original structure must only be accessed for reads.

In addition to the list methods below, the *Notifier* supports the index syntax for modification and deletion of elements. Modification of nested structures can be also done using the index syntax, for example:

```
>>> n = Notifier([])
>>> n.append([])
>>> n[0].append(42)
>>> n.raw_view
[[42]]
```

This class does not perform any network I/O and is meant to be used with e.g. the *Publisher* for this purpose. Only one publisher at most can be associated with a *Notifier*.

Parameters **backing_struct** – Structure to encapsulate.

append(*x*)
Append to a list.

insert(*i*, *x*)
Insert an element into a list.

pop(*i=-1*)
Pop an element from a list. The returned element is not encapsulated in a *Notifier* and its mutations are no longer tracked.

raw_view = None
The raw data encapsulated (read-only!).

class sipyco.sync_struct.**Publisher**(*notifiers*)
A network server that publish changes to structures encapsulated in a *Notifier*.

Parameters **notifiers** – A dictionary containing the notifiers to associate with the *Publisher*. The keys of the dictionary are the names of the notifiers to be used with *Subscriber*.

class sipyco.sync_struct.**Subscriber**(*notifier_name*, *target_builder*, *notify_cb=None*, *disconnect_cb=None*)
An asyncio-based client to connect to a Publisher.

Parameters

- **notifier_name** – Name of the notifier to subscribe to.
- **target_builder** – A function called during initialization that takes the object received from the publisher and returns the corresponding local structure to use. Can be identity.

- **notify_cb** – An optional function called every time a mod is received from the publisher. The mod is passed as parameter. The function is called after the mod has been processed. A list of functions may also be used, and they will be called in turn.
- **disconnect_cb** – An optional function called when disconnection happens from external causes (i.e. not when `close` is called).

sipyco.sync_struct.**process_mod**(*target*, *mod*)
Apply a *mod* to the target, mutating it.

sipyco.sync_struct.**update_from_dict**(*target*, *source*)
Updates notifier contents from given source dictionary.

Only the necessary changes are performed; unchanged fields are not written. (Currently, modifications are only performed at the top level. That is, whenever there is a change to a child array/struct the entire member is updated instead of choosing a more optimal set of mods.)

1.5 sipyco.remote_exec module

This module provides facilities for experiment to execute code remotely on RPC servers.

The remotely executed code has direct access to the resources on the remote end, so it can transfer vlarge amounts of data with them, and only exchange higher-level, processed data with the client (and over the network).

RPC servers with support for remote execution contain an additional target that gives RPC access to instances of *RemoteExecServer*. One such instance is created per client connection and manages one Python namespace in which the client can execute arbitrary code by calling the methods of *RemoteExecServer*.

The namespaces are initialized with the following global values:

- `controller_driver` - the target object of the RPC server.
- `controller_initial_namespace` - a server-wide dictionary copied when initializing a new namespace.
- all values from `controller_initial_namespace`.

With ARTIQ, access to a controller with support for remote execution is done through an additional device database entry of this form:

```
"$REXEC_DEVICE_NAME": {
    "type": "controller_aux_target",
    "controller": "$CONTROLLER_DEVICE_NAME",
    "target_name": "$TARGET_NAME_FOR_REXEC"
}
```

Specifying `target_name` is mandatory in all device database entries for all controllers with remote execution support.

class sipyco.remote_exec.**RemoteExecServer**(*initial_namespace*)
RPC target created at each connection by controllers with remote execution support. Manages one Python namespace and provides RPCs for code execution.

add_code(*code*)
Executes the specified code in the namespace.

Parameters **code** – a string containing valid Python code

call(*function*, **args*, ***kwargs*)
Calls a function in the namespace, passing it positional and keyword arguments, and returns its value.

Parameters **function** – a string containing the name of the function to execute.

sipyco.remote_exec.**simple_rexec_server_loop**(*target_name*, *target*, *host*, *port*, *description=None*)
Runs a server with remote execution support, until an exception is raised (e.g. the user hits Ctrl-C) or termination is requested by a client.

sipyco.remote_exec.**connect_global_rpc**(*controller_rexec*, *host=None*, *port=3251*, *target='master_dataset_db'*, *name='dataset_db'*)
Creates a global RPC client in a RPC server that is used across all remote execution connections.

The default parameters are designed to connects to an ARTIQ dataset database (i.e. gives direct dataset access to experiment code remotely executing in controllers).

If a global object with the same name already exists, the function does nothing.

Parameters

- **controller_rexec** – the RPC client connected to the server's remote execution interface.
- **host** – the host name to connect the RPC client to. Default is the local end of the remote execution interface (typically, the ARTIQ master).
- **port** – TCP port to connect the RPC client to.
- **target** – name of the RPC target.
- **name** – name of the object to insert into the global namespace.

1.6 sipyco.common_args module

sipyco.common_args.**verbosity_args**(*parser*)
Adds *-v*/*-q* arguments that increase or decrease the default logging levels. Repeat for higher levels.

1.7 sipyco.asyncio_tools module

class sipyco.asyncio_tools.**AsyncioServer**
Generic TCP server based on asyncio.

Users of this class must derive from it and define the `_handle_connection_cr()` method/coroutine.

start(*host*, *port*)
Starts the server.

The user must call *stop()* to free resources properly after this method completes successfully.

This method is a *coroutine*.

Parameters

- **host** – Bind address of the server (see `asyncio.start_server` from the Python standard library).
- **port** – TCP port to bind to.

stop()
Stops the server.

1.8 sipyco.logging_tools module

class sipyco.logging_tools.**LogForwarder**(*host*, *port*, *reconnect_timer=5.0*, *queue_size=1000*, ***kwargs*)

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

class sipyco.logging_tools.**MultilineFormatter**

format(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class sipyco.logging_tools.**Server**
Remote logging TCP server.

Log entries are in the format:

```
source:levelno<total_lines>:name:message
continuation...
...continuation
```

CHAPTER

TWO

REMOTE PROCEDURE CALL TOOL

This tool is the preferred way of handling simple RPC servers. Instead of writing a client for simple cases, you can simply use this tool to call remote functions of an RPC server.

- Listing existing targets

  The `list-targets` sub-command will print to standard output the target list of the remote server:

  ```
  $ sipyco_rpctool hostname port list-targets
  ```

- Listing callable functions

  The `list-methods` sub-command will print to standard output a sorted list of the functions you can call on the remote server's target.

  The list will contain function names, signatures (arguments) and docstrings.

  If the server has only one target, you can do:

  ```
  $ sipyco_rpctool hostname port list-methods
  ```

  Otherwise you need to specify the target, using the `-t target` option:

  ```
  $ sipyco_rpctool hostname port list-methods -t target_name
  ```

- Remotely calling a function

  The `call` sub-command will call a function on the specified remote server's target, passing the specified arguments. Like with the previous sub-command, you only need to provide the target name (with `-t target`) if the server hosts several targets.

  The following example will call the `set_attenuation` method of the Lda controller with the argument `5`:

  ```
  $ sipyco_rpctool ::1 3253 call -t lda set_attenuation 5
  ```

  In general, to call a function named `f` with N arguments named respectively `x1, x2, ..., xN` you can do:

  ```
  $ sipyco_rpctool hostname port call -t target f x1 x2 ... xN
  ```

  You can use Python syntax to compute arguments as they will be passed to the `eval()` primitive. The numpy package is available in the namespace as `np`. Beware to use quotes to separate arguments which use spaces:

  ```
  $ sipyco_rpctool hostname port call -t target f '3 * 4 + 2' True '[1, 2]'
  $ sipyco_rpctool ::1 3256 call load_sample_values 'np.array([1.0, 2.0],␣
  ↪dtype=float)'
  ```

(continues on next page)

(continued from previous page)

If the called function has a return value, it will get printed to the standard output if the value is not None like in the standard python interactive console:

```
$ sipyco_rpctool ::1 3253 call get_attenuation
5.0
```

Command-line details:

ARTIQ RPC tool

```
usage: sipyco_rpctool [-h]
                      SERVER PORT {list-targets,list-methods,call,interactive}
                      ...
```

2.1 Positional Arguments

SERVER	hostname or IP of the controller to connect to
PORT	TCP port to use to connect to the controller
action	Possible choices: list-targets, list-methods, call, interactive

2.2 Sub-commands:

2.2.1 list-targets

list existing targets

```
sipyco_rpctool list-targets [-h]
```

2.2.2 list-methods

list target's methods

```
sipyco_rpctool list-methods [-h] [-t TARGET]
```

Named Arguments

-t, --target	target name

2.2.3 call

call a target's method

```
sipyco_rpctool call [-h] [-t TARGET] METHOD ...
```

Positional Arguments

METHOD	method name
ARGS	arguments

Named Arguments

-t, --target target name

2.2.4 interactive

enter interactive mode (default)

```
sipyco_rpctool interactive [-h] [-t TARGET]
```

Named Arguments

-t, --target target name

PYTHON MODULE INDEX

S

INDEX

S

U

V