NeuroRPC Documentation

Client–Server Library for RPC over TCP (Python-LabView)

Froylan ARAGON

2025 FEMTO-ST Institute

Table of Contents

1. NeuroRPC Documentation

Welcome to the **NeuroRPC** project documentation.

NeuroRPC is a Python-based framework that provides an interface for **remote procedure calls (RPC)** over **TCP/IP** in the context of distributed and embedded systems.
The library has been specifically developed to facilitate communication between Python applications and the **LabVIEW Actor Framework**, enabling reproducible, modular, and low-latency control architectures for experimental platforms.

1.1 Objectives

The primary objectives of **NeuroRPC** are:

- To offer a **transparent client–server model** for exchanging structured messages between heterogeneous environments (Python ↔ LabVIEW).
- To enable the design of **actor-based systems** in which commands and data streams are handled as serializable messages.
- To support **scientific instrumentation workflows**, such as real-time data acquisition, signal analysis, and closed-loop experimental control.
- To provide a foundation that can be **extended to other protocols or platforms** while preserving consistency and interoperability.

1.2 Key Features

- **TCP-based communication layer** ensuring reliable and ordered message delivery.
- **Serialization/deserialization** mechanisms compatible with LabVIEW binary flattening.
- **Actor-oriented modularity**, allowing processes to be encapsulated as autonomous units.
- **Cross-platform support** through Python and LabVIEW integration.
- **Documentation auto-generation** via MkDocs and mkdocstrings.

1.3 Structure of This Documentation

- **Conceptual Overview** – background on the RPC model, actor-based design principles, and system architecture.
- **API Reference** – detailed technical specification of modules, classes, and methods.
- **Examples and Workflows** – application to data acquisition, remote control, and message handling.
- **Integration Guidelines** – recommendations for extending NeuroRPC within experimental or industrial frameworks.

1.4 Quickstart

Minimal client example

```bash from neuro_rpc import Client

client = Client("127.0.0.1", port=2001) client.rpc("Display Text", {"Message": "Trying something :)", "exec_time": 0}, False)
```

2. API Reference

2.1 Client

TCP client for framed JSON-RPC-like communication.

This module implements a Python client that communicates with a LabVIEW/ CompactRIO server using a custom framed JSON message protocol. It manages socket lifecycle, message serialization, connection retries, and integration with the RPC stack.

Notes

- All socket operations are blocking.
- Background operation is achieved by running the client in a thread.

2.1.1 `Client(host='127.0.0.1', port=6363, encoding='UTF-8', endian='>I', timeout=10.0, max_retries=3, retry_delay=1.0, handler=None, no_delay=True)`

TCP Client for framed JSON messages.

Manages connection lifecycle, sending/receiving messages, background thread execution, and integration with RPC handlers and trackers.

Initialize a Client instance with connection parameters.

Parameters:

- **`host`** (`str`, default: `'127.0.0.1'`) – Target hostname or IP address.
- **`port`** (`int`, default: `6363`) – TCP port of the server.
- **`encoding`** (`str`, default: `'UTF-8'`) – Encoding for JSON messages.
- **`endian`** (`str`, default: `'>I'`) – Struct format for message length (e.g., `'>I'` big-endian).
- **`timeout`** (`float`, default: `10.0`) – Socket timeout in seconds.
- **`max_retries`** (`int`, default: `3`) – Maximum number of connection attempts.
- **`retry_delay`** (`float`, default: `1.0`) – Delay between retry attempts in seconds.
- **`handler`** – Optional RPC handler, defaults to `RPCMethods()`.
- **`no_delay`** (`bool`, default: `True`) – If True, disables Nagle’s algorithm and sets DSCP EF.

Source code in `python/neuro_rpc/Client.py`

```
def __init__(self, host: str = "127.0.0.1", port: int = 6363, encoding: str = 'UTF-8', endian: str = '>I', timeout: float = 10.0,
```

`connect(retry=True)`

Establish a TCP connection with retry support.

Parameters:
- `retry` (`bool`, default: `True`) – Whether to retry failed attempts.

Returns:
- `bool` (`bool`) – True if connected successfully.

Raises:
- `ConnectionError` – If all attempts fail.

Source code in `python/neuro_rpc/Client.py`

```
def connect(self, retry: bool = True) -> bool: """ Establish a TCP connection with retry support. Args: retry (bool): Whether to
```

`disconnect()`

Close the TCP connection.

Notes

Resets the socket and updates state to disconnected.

Source code in `python/neuro_rpc/Client.py`

```
def disconnect(self) -> None: """ Close the TCP connection. Notes: Resets the socket and updates state to disconnected. """ if se
```

`echo(message='test')`

Send an `echo` request and track its execution time.

Parameters:
- `message` (`str`, default: `'test'`) – String to send.

Source code in `python/neuro_rpc/Client.py`

```
def echo(self, message='test'): """ Send an ``echo`` request and track its execution time. Args: message (str): String to send. "
```

`echo_benchmark()`

Run a benchmark using echo requests with increasing payload sizes.

Iterates over multiple message sizes, repeating each size multiple times, and records metrics through the Benchmark tracker.

Source code in `python/neuro_rpc/Client.py`

```
def echo_benchmark(self): """ Run a benchmark using echo requests with increasing payload sizes. Iterates over multiple message s
```

`ensure_connected()`

Verify connection before sending/receiving.

Raises:

- `ConnectionError` – If not connected.

Source code in `python/neuro_rpc/Client.py`

```
def ensure_connected(self) -> None: """ Verify connection before sending/receiving. Raises: ConnectionError: If not connected. "
```

`receive_message(timeout=None, partial_timeout=None)`

Receive and parse a JSON message.

Parameters:

- `timeout` (`float | None`, default: `None`) – Optional override for socket timeout.
- `partial_timeout` (`float | None`, default: `None`) – Timeout for the remainder after the header.

Returns:

- `dict` (`Any`) – Parsed JSON response.

Raises:

- `ConnectionError` – If disconnected.
- `TimeoutError` – If operation times out.
- `MessageError` – If parsing fails.

Source code in `python/neuro_rpc/Client.py`

```
def receive_message(self, timeout: Optional[float] = None, partial_timeout: Optional[float] = None) -> Any: """ Receive and parse
```

`recv_packet()`

Receive a framed packet.

Returns:

- – tuple[int, bytes, int] | None: `(size, data_bytes, trailer_int)` or `None` on error.

Source code in `python/neuro_rpc/Client.py`

```
def recv_packet(self): """ Receive a framed packet. Returns: tuple[int, bytes, int] | None: ``(size, data_bytes, trailer_int)`` o
```

`rpc(method, params, response=True)`

Perform an RPC call using Proxy encoding.

Parameters:

- `method` (`str`) – RPC method name.
- `params` (`dict`) – Parameters.
- `response` (`bool`, default: `True`) – Whether to wait for and return a response.

Returns:

- – tuple[int, str, int] | None: `(size, json_str, tail)` if `response=True`, else `None`.

Source code in `python/neuro_rpc/Client.py`

```
def rpc(self, method, params, response=True): """ Perform an RPC call using Proxy encoding. Args: method (str): RPC method name.
```

`send_and_receive(message, timeout=None, retry_on_error=True)`

Convenience wrapper to send and immediately receive a message.

Parameters:

- `message` (`dict`) – Message to send.
- `timeout` (`float | None`, default: `None`) – Optional receive timeout.
- `retry_on_error` (`bool`, default: `True`) – Whether to retry on send errors.

Returns:

- `dict` (`Any`) – Parsed JSON response.

Source code in `python/neuro_rpc/Client.py`

```
def send_and_receive(self, message: Dict[str, Any], timeout: Optional[float] = None, retry_on_error: bool = True) -> Any: """ Co
```

`send_message(message, retry_on_error=True)`

Send a JSON message with retry support.

Parameters:

- `message` (`dict`) – JSON-compatible message.
- `retry_on_error` (`bool`, default: `True`) – Whether to retry on socket errors.

Returns:

- `bool` (`bool`) – True if sent successfully.

Raises:

- `ConnectionError` – If not connected.
- `MessageError` – If serialization or send fails.

Source code in `python/neuro_rpc/Client.py`

```
def send_message(self, message: Dict[str, Any], retry_on_error: bool = True) -> bool: """ Send a JSON message with retry support.
```

`send_packet(packet)`

Send a raw packet.

Parameters:

- **`packet`** (`bytes`) – Complete framed packet.

Returns:

- **`bool`** – True if sent successfully.

Source code in `python/neuro_rpc/Client.py`

```
def send_packet(self, packet): """ Send a raw packet. Args: packet (bytes): Complete framed packet. Returns: bool: True if sent s
```

`start()`

Start the client in a background thread.

Notes

Spawns a daemon thread that calls `connect()` and maintains the connection.

Source code in `python/neuro_rpc/Client.py`

```
def start(self): """ Start the client in a background thread. Notes: Spawns a daemon thread that calls ``connect()`` and maintain
```

`stop()`

Stop the client thread and disconnect.

Calls `disconnect()`, stops monitoring, and joins the thread.

Source code in `python/neuro_rpc/Client.py`

```
def stop(self): """ Stop the client thread and disconnect. Calls ``disconnect()``, stops monitoring, and joins the thread. """ if
```

2.1.2 `ConnectionError`

Bases: `Exception`

Raised for connection-related errors (e.g., failed connect or lost connection).

2.1.3 `MessageError`

Bases: `Exception`

Raised when a message cannot be serialized, sent, or parsed.

2.1.4 `TimeoutError`

Bases: `Exception`

Raised when a receive operation exceeds the configured timeout.

2.2 Client

TCP client for framed JSON-RPC-like communication.

This module implements a Python client that communicates with a LabVIEW/ CompactRIO server using a custom framed JSON message protocol. It manages socket lifecycle, message serialization, connection retries, and integration with the RPC stack.

Notes

- All socket operations are blocking.
- Background operation is achieved by running the client in a thread.

2.2.1 `Client(host='127.0.0.1', port=6363, encoding='UTF-8', endian='>I', timeout=10.0, max_retries=3, retry_delay=1.0, handler=None, no_delay=True)`

TCP Client for framed JSON messages.

Manages connection lifecycle, sending/receiving messages, background thread execution, and integration with RPC handlers and trackers.

Initialize a Client instance with connection parameters.

Parameters:

- **`host`** (`str`, default: `'127.0.0.1'`) – Target hostname or IP address.
- **`port`** (`int`, default: `6363`) – TCP port of the server.
- **`encoding`** (`str`, default: `'UTF-8'`) – Encoding for JSON messages.
- **`endian`** (`str`, default: `'>I'`) – Struct format for message length (e.g., `'>I'` big-endian).
- **`timeout`** (`float`, default: `10.0`) – Socket timeout in seconds.
- **`max_retries`** (`int`, default: `3`) – Maximum number of connection attempts.
- **`retry_delay`** (`float`, default: `1.0`) – Delay between retry attempts in seconds.
- **`handler`** – Optional RPC handler, defaults to `RPCMethods()`.
- **`no_delay`** (`bool`, default: `True`) – If True, disables Nagle’s algorithm and sets DSCP EF.

Source code in `python/neuro_rpc/Client.py`

```
def __init__(self, host: str = "127.0.0.1", port: int = 6363, encoding: str = 'UTF-8', endian: str = '>I', timeout: float = 10.0,
```

`connect(retry=True)`

Establish a TCP connection with retry support.

Parameters:
- `retry` (`bool`, default: `True`) – Whether to retry failed attempts.

Returns:
- `bool` (`bool`) – True if connected successfully.

Raises:
- `ConnectionError` – If all attempts fail.

Source code in `python/neuro_rpc/Client.py`

```
def connect(self, retry: bool = True) -> bool: """ Establish a TCP connection with retry support. Args: retry (bool): Whether to
```

`disconnect()`

Close the TCP connection.

Notes

Resets the socket and updates state to disconnected.

Source code in `python/neuro_rpc/Client.py`

```
def disconnect(self) -> None: """ Close the TCP connection. Notes: Resets the socket and updates state to disconnected. """ if se
```

`echo(message='test')`

Send an `echo` request and track its execution time.

Parameters:
- `message` (`str`, default: `'test'`) – String to send.

Source code in `python/neuro_rpc/Client.py`

```
def echo(self, message='test'): """ Send an ``echo`` request and track its execution time. Args: message (str): String to send. "
```

`echo_benchmark()`

Run a benchmark using echo requests with increasing payload sizes.

Iterates over multiple message sizes, repeating each size multiple times, and records metrics through the Benchmark tracker.

Source code in `python/neuro_rpc/Client.py`

```
def echo_benchmark(self): """ Run a benchmark using echo requests with increasing payload sizes. Iterates over multiple message s
```

`ensure_connected()`

Verify connection before sending/receiving.

Raises:

- `ConnectionError` – If not connected.

Source code in `python/neuro_rpc/Client.py`

```
def ensure_connected(self) -> None: """ Verify connection before sending/receiving. Raises: ConnectionError: If not connected. "
```

`receive_message(timeout=None, partial_timeout=None)`

Receive and parse a JSON message.

Parameters:

- **`timeout`** (`float | None`, default: `None`) – Optional override for socket timeout.
- **`partial_timeout`** (`float | None`, default: `None`) – Timeout for the remainder after the header.

Returns:

- **`dict`** (`Any`) – Parsed JSON response.

Raises:

- `ConnectionError` – If disconnected.
- `TimeoutError` – If operation times out.
- `MessageError` – If parsing fails.

Source code in `python/neuro_rpc/Client.py`

```
def receive_message(self, timeout: Optional[float] = None, partial_timeout: Optional[float] = None) -> Any: """ Receive and parse
```

`recv_packet()`

Receive a framed packet.

Returns:

- – tuple[int, bytes, int] | None: `(size, data_bytes, trailer_int)` or `None` on error.

Source code in `python/neuro_rpc/Client.py`

```
def recv_packet(self): """ Receive a framed packet. Returns: tuple[int, bytes, int] | None: ``(size, data_bytes, trailer_int)`` o
```

`rpc(method, params, response=True)`

Perform an RPC call using Proxy encoding.

Parameters:

- `method` (`str`) – RPC method name.
- `params` (`dict`) – Parameters.
- `response` (`bool`, default: `True`) – Whether to wait for and return a response.

Returns:

- – tuple[int, str, int] | None: `(size, json_str, tail)` if `response=True`, else `None`.

Source code in `python/neuro_rpc/Client.py`

```
def rpc(self, method, params, response=True): """ Perform an RPC call using Proxy encoding. Args: method (str): RPC method name.
```

`send_and_receive(message, timeout=None, retry_on_error=True)`

Convenience wrapper to send and immediately receive a message.

Parameters:

- `message` (`dict`) – Message to send.
- `timeout` (`float | None`, default: `None`) – Optional receive timeout.
- `retry_on_error` (`bool`, default: `True`) – Whether to retry on send errors.

Returns:

- `dict` (`Any`) – Parsed JSON response.

Source code in `python/neuro_rpc/Client.py`

```
def send_and_receive(self, message: Dict[str, Any], timeout: Optional[float] = None, retry_on_error: bool = True) -> Any: """ Co
```

`send_message(message, retry_on_error=True)`

Send a JSON message with retry support.

Parameters:

- `message` (`dict`) – JSON-compatible message.
- `retry_on_error` (`bool`, default: `True`) – Whether to retry on socket errors.

Returns:

- `bool` (`bool`) – True if sent successfully.

Raises:

- `ConnectionError` – If not connected.
- `MessageError` – If serialization or send fails.

Source code in `python/neuro_rpc/Client.py`

```
def send_message(self, message: Dict[str, Any], retry_on_error: bool = True) -> bool: """ Send a JSON message with retry support.
```

`send_packet(packet)`

Send a raw packet.

Parameters:

- **`packet`** (`bytes`) – Complete framed packet.

Returns:

- **`bool`** – True if sent successfully.

Source code in `python/neuro_rpc/Client.py`

```
def send_packet(self, packet): """ Send a raw packet. Args: packet (bytes): Complete framed packet. Returns: bool: True if sent s
```

`start()`

Start the client in a background thread.

Notes

Spawns a daemon thread that calls `connect()` and maintains the connection.

Source code in `python/neuro_rpc/Client.py`

```
def start(self): """ Start the client in a background thread. Notes: Spawns a daemon thread that calls ``connect()`` and maintain
```

`stop()`

Stop the client thread and disconnect.

Calls `disconnect()`, stops monitoring, and joins the thread.

Source code in `python/neuro_rpc/Client.py`

```
def stop(self): """ Stop the client thread and disconnect. Calls ``disconnect()``, stops monitoring, and joins the thread. """ if
```

2.2.2 `ConnectionError`

Bases: `Exception`

Raised for connection-related errors (e.g., failed connect or lost connection).

2.2.3 `MessageError`

Bases: `Exception`

Raised when a message cannot be serialized, sent, or parsed.

2.2.4 `TimeoutError`

Bases: `Exception`

Raised when a receive operation exceeds the configured timeout.

2.3 Console

Interactive console wrapper for NeuroRPC client.

Provides a REPL-style interface to manually start, stop, and inspect the state of the TCP client. Useful for debugging and testing RPC connections interactively.

Notes

- Runs with Python's built-in InteractiveConsole.
- Available commands and modules are preloaded in the interactive namespace.

2.3.1 `Console(client_config=None)`

Interactive console for manual client control.

Encapsulates startup and shutdown logic for the NeuroRPC client and exposes convenience commands in an interactive REPL environment. Provides status checks and log outputs for debugging.

Initialize the interactive console with client configuration.

Parameters:

- `client_config` (`dict | None`, default: `None`) – Optional dictionary of client configuration parameters (host, port, etc.). If None, defaults from `Client` are used.

Source code in `python/neuro_rpc/Console.py`

```
def __init__(self, client_config=None): """ Initialize the interactive console with client configuration. Args: client_config (di
```

`clear_screen()`

Clear the console screen.

Notes

Uses OS-specific commands: `cls` on Windows, `clear` on Unix-like systems.

Source code in `python/neuro_rpc/Console.py`

```
def clear_screen(self) -> None: """ Clear the console screen. Notes: Uses OS-specific commands: ``cls`` on Windows, ``clear`` on
```

`client_status()`

Display current client status in the logger.

Shows information about the client object, connection status, active thread, and registered RPC methods (if available).

Source code in `python/neuro_rpc/Console.py`

```
def client_status(self): """ Display current client status in the logger. Shows information about the client object, connection s
```

`run()`

Run the interactive console.

Initializes the interactive console and blocks until exit. Handles Ctrl+C gracefully, stopping the client if necessary.

Source code in `python/neuro_rpc/Console.py`

```
def run(self): """ Run the interactive console. Initializes the interactive console and blocks until exit. Handles Ctrl+C gracefu
```

`start_client()`

Start the client in a background thread.

Notes

Instantiates `Client` if not yet created and calls `start()`.

Source code in `python/neuro_rpc/Console.py`

```
def start_client(self): """ Start the client in a background thread. Notes: Instantiates ``Client`` if not yet created and calls
```

`start_interactive_console()`

Start the REPL console with preloaded commands.

Provides start/stop/status/cls commands and access to Logger and LoggerConfig. A banner with usage instructions is displayed at startup.

Source code in `python/neuro_rpc/Console.py`

```
def start_interactive_console(self): """ Start the REPL console with preloaded commands. Provides start/stop/status/cls commands a
```

`stop_client()`

Stop the running client thread.

Notes

Calls `Client.stop()`. Logs error if client is uninitialized.

Source code in `python/neuro_rpc/Console.py`

```
def stop_client(self): """ Stop the running client thread. Notes: Calls ``Client.stop()``. Logs error if client is uninitialized.
```

2.4 Logger

Colorized, structured logging utilities for the NeuroRPC stack.

Provides a compact ANSI color formatter and a reusable Logger factory. All modules (Client, Benchmark, RPC stack, Console) use this centralized logging infrastructure to ensure consistent formatting and runtime readability.

Notes

- Uses colorama for cross-platform color handling.
- Verbosity can be adjusted dynamically.

2.4.1 `ColoredFormatter`

Bases: `Formatter`

Minimal ANSI color formatter.

Extends `logging.Formatter` to prepend log messages with ANSI color codes depending on the log level. Colors are reset automatically after each message.

`format(record)`

Apply color formatting to a log record.

Parameters:

- `record` (`LogRecord`) – Log record object to format.

Returns:

- `str` – Formatted string with ANSI colors.

Source code in `python/neuro_rpc/Logger.py`

```
def format(self, record): """ Apply color formatting to a log record. Args: record (logging.LogRecord): Log record object to form
```

2.4.2 `Logger(name, level=logging.DEBUG, verbose=True)`

Bases: `Logger`

Factory for module-level loggers with a shared format.

Ensures all modules share a consistent format and color scheme. Provides caching of Logger instances by name, synchronized levels, and verbosity toggling.

Initialize a Logger instance.

Parameters:

- `name` (`str`) – Name of the logger.
- `level` (`int`, default: `DEBUG`) – Log level.
- `verbose` (`bool`, default: `True`) – Whether verbose format is enabled.

Source code in `python/neuro_rpc/Logger.py`

```
def __init__(self, name: str, level=logging.DEBUG, verbose: bool = True): """ Initialize a Logger instance. Args: name (str): Nam
```

`get_logger(name='__neuro__', level=logging.DEBUG, verbose=True)` `staticmethod`

Get or create a configured logger.

Parameters:

- `name` (`str`, default: `'__neuro__'`) – Logger identifier (typically module or class name).
- `level` (`int`, default: `DEBUG`) – Log level (default `DEBUG`).
- `verbose` (`bool`, default: `True`) – Whether to include extended context (process/thread info).

Returns:

- `Logger` – Configured logger instance.

Notes

Attaches a StreamHandler on first creation.

Source code in `python/neuro_rpc/Logger.py`

```
@staticmethod def get_logger(name="__neuro__", level=logging.DEBUG, verbose=True): """ Get or create a configured logger. Args: n
```

`print_loggers()` `staticmethod`

Print currently registered loggers.

Useful for debugging which loggers are active and their configuration.

Source code in `python/neuro_rpc/Logger.py`

```
@staticmethod def print_loggers(): """ Print currently registered loggers. Useful for debugging which loggers are active and thei
```

`setLevel(level)`

Override setLevel to synchronize handler level.

Parameters:

- `level` (`int`) – New log level.

Source code in `python/neuro_rpc/Logger.py`

```
def setLevel(self, level) -> None: """ Override setLevel to synchronize handler level. Args: level (int): New log level. """ supe
```

`setVerbose(verbose)`

Dynamically update verbosity and re-apply formatter.

Parameters:

- `verbose` (`bool`) – True for detailed context, False for compact output.

Source code in `python/neuro_rpc/Logger.py`

```
def setVerbose(self, verbose: bool) -> None: """ Dynamically update verbosity and re-apply formatter. Args: verbose (bool): True
```

`test()`

Emit test messages at all levels.

Useful for verifying logger color and format configuration.

Source code in `python/neuro_rpc/Logger.py`

```
def test(self) -> None: """ Emit test messages at all levels. Useful for verifying logger color and format configuration. """ sel
```

2.4.3 `LoggerConfig`

Default logging configuration holder.

Provides presets for production, development, and per-component debugging.

`configure_for_debugging(component_name, level=logging.DEBUG, verbose=True)` `staticmethod`

Configure a specific component for debugging.

Parameters:

- `component_name` (`str`) – Name of the component/logger.
- `level` (`int`, default: `DEBUG`) – Log level (default `DEBUG`).
- `verbose` (`bool`, default: `True`) – Verbosity flag.

Notes

Creates a new logger if not already registered.

Source code in `python/neuro_rpc/Logger.py`

```
@staticmethod def configure_for_debugging(component_name, level=logging.DEBUG, verbose=True): """ Configure a specific component
```

`configure_for_development()` `staticmethod`

Configure all loggers for development.

Sets `DEBUG` level and enables verbose formatting.

Source code in `python/neuro_rpc/Logger.py`

```
@staticmethod def configure_for_development(): """ Configure all loggers for development. Sets ``DEBUG`` level and enables verbos
```

`configure_for_production()` `staticmethod`

Configure all loggers for production.

Sets `INFO` level and disables verbose formatting.

Source code in `python/neuro_rpc/Logger.py`

```
@staticmethod def configure_for_production(): """ Configure all loggers for production. Sets ``INFO`` level and disables verbose
```

2.5 Proxy

Conversion utilities between Python dictionaries/tuples and LabVIEW Clusters.

Provides serialization and deserialization helpers to encode/decode nested data structures into LabVIEW's Cluster representation. Also integrates with RPCRequest/RPCResponse to support actor-style communication with LabVIEW classes.

Notes

- Extends ClusterConverter from `python.labview_data.type_converters`.

2.5.1 `NpEncoder`

Bases: `JSONEncoder`

JSON encoder for NumPy data types.

Converts `numpy.integer`, `numpy.floating`, and `numpy.ndarray` into standard Python `int`, `float`, and `list` for JSON serialization compatibility.

`default(obj)`

Override JSON encoding for NumPy objects.

Parameters:

- `obj` (`Any`) – Object to encode.

Returns:

- `Any` – Encoded Python-native type.

Source code in `python/neuro_rpc/Proxy.py`

```
def default(self, obj): """ Override JSON encoding for NumPy objects. Args: obj (Any): Object to encode. Returns: Any: Encoded Py
```

2.5.2 `Proxy`

Bases: `ClusterConverter`

Proxy class to convert between Python dicts and LabVIEW Cluster bytes.

Implements bidirectional mapping of nested dict/tuple structures to LabVIEW Cluster format, supporting serialization for sending RPC requests and deserialization of responses.

`dict_to_tuple(d)`

Convert a dictionary into a (values, keys) tuple.

Recursively descends into nested dictionaries to preserve structure.

Parameters:
- `d` (`dict`) – Input dictionary.

Returns:
- `tuple[list, list]` – tuple[list, list]: (values, keys) representation of the dictionary.

Source code in `python/neuro_rpc/Proxy.py`

```
def dict_to_tuple(self, d: dict) -> tuple[list, list]: """ Convert a dictionary into a (values, keys) tuple. Recursively descends
```

`from_act(raw_bytes, hdr_tree)`

Convert LabVIEW Actor Cluster bytes back into an RPCResponse.

Parameters:
- `raw_bytes` (`bytes`) – Cluster flat buffer.
- `hdr_tree` (`dict`) – Metadata tree from serialization.

Returns:
- `dict` – RPCResponse serialized as dictionary.

Source code in `python/neuro_rpc/Proxy.py`

```
def from_act(self, raw_bytes: bytes, hdr_tree: dict): """ Convert LabVIEW Actor Cluster bytes back into an RPCResponse. Args: raw
```

`from_cluster_bytes_and_tree(raw_bytes, hdr_tree, sdata=None, encoding='ansi')`

Reconstruct a (values, keys) tuple from Cluster bytes and metadata tree.

Parameters:
- `raw_bytes` (`bytes`) – Flat buffer for the cluster.
- `hdr_tree` (`dict`) – Metadata tree including headers, keys, and children.
- `sdata` (`SerializationData`, default: `None`) – Deserialization context.
- `encoding` (`str`, default: `'ansi'`) – Encoding used for string payloads. Defaults to "ansi".

Returns:
- `tuple[list, list]` – tuple[list, list]: (values, keys) structure.

Source code in `python/neuro_rpc/Proxy.py`

```
def from_cluster_bytes_and_tree( self, raw_bytes: bytes, hdr_tree: dict, sdata: SerializationData = None, encoding: str = "ansi"
```

`to_act(Message)`

Convert an RPCRequest/Message dict into a LabVIEW Actor Cluster.

Parameters:
- `Message` (`dict | RPCRequest`) – Message to encode.

Returns:
- – tuple[bytes, dict]: (flat buffer, metadata tree).

Source code in `python/neuro_rpc/Proxy.py`

```
def to_act(self, Message): """ Convert an RPCRequest/Message dict into a LabVIEW Actor Cluster. Args: Message (dict | RPCRequest)
```

`to_cluster_bytes_with_tree(tup, sdata=None, encoding='ansi')`

Serialize a (values, keys) tuple into a LabVIEW Cluster flat buffer.

Parameters:
- `tup` (`tuple[list, list]`) – (values, keys) representation of the cluster.
- `sdata` (`SerializationData`, default: `None`) – Serialization metadata. Defaults to version=0.
- `encoding` (`str`, default: `'ansi'`) – Encoding to use for nested buffers. Defaults to "ansi".

Returns:
- `tuple[bytes, dict]` – tuple[bytes, dict]: (flat buffer, metadata tree).

Source code in `python/neuro_rpc/Proxy.py`

```
def to_cluster_bytes_with_tree( self, tup: tuple[list, list], sdata: SerializationData = None, encoding: str = "ansi" ) -> tuple[
```

`tuple_to_dict(values_keys)`

Convert a (values, keys) tuple back into a dictionary.

Recursively reconstructs nested dictionaries from tuple representations.

Parameters:
- `values_keys` (`tuple[list, list]`) – (values, keys) pair.

Returns:
- `dict` (`dict`) – Reconstructed dictionary.

Source code in `python/neuro_rpc/Proxy.py`

```
def tuple_to_dict(self, values_keys) -> dict: """ Convert a (values, keys) tuple back into a dictionary. Recursively reconstructs
```

2.6 RPCHandler

Registration and dispatch of RPC request/response methods.

Implements a handler for JSON-Message 2.0 (similar to JSON-RPC 2.0), providing: - Method registration via the @rpc_method decorator. - Creation of request, response, and error messages. - Processing of incoming messages (both requests and responses). - Integration with Benchmark to track latency and round-trip times.

Notes

- Acts as the bridge between raw JSON messages and Python method calls.

2.6.1 `RPCHandler()`

Bases: `RPCMessage`

Core handler for JSON-Message 2.0 operations.

Manages registration of request/response handlers, creation of message objects, and routing of incoming messages. Integrates with Benchmark to track requests and responses.

Initialize the RPCHandler.

Creates registries for request/response methods, sets up a Benchmark tracker, and initializes a logger.

Source code in `python/neuro_rpc/RPCHandler.py`

```
def __init__(self): """ Initialize the RPCHandler. Creates registries for request/response methods, sets up a Benchmark tracker,
```

`create_error(error_type, data=None, id=None)`

Create a JSON-Message error object.

Parameters:

- `error_type` (`str | dict`) – Error type (see RPCError constants).
- `data` (`Any`, default: `None`) – Additional error details.
- `id` (`str`, default: `None`) – ID of the related request.

Returns:

- `dict` – Serialized error response object.

Source code in `python/neuro_rpc/RPCHandler.py`

```
def create_error(self, error_type, data=None, id=None): """ Create a JSON-Message error object. Args: error_type (str | dict): Er
```

`create_request(method, params=None, request_id=None)`

Create a JSON-Message request object.

Parameters:

- **`method`** (`str`) – Method name to call.
- **`params`** (`dict | list`, default: `None`) – Parameters for the request.
- **`request_id`** (`str`, default: `None`) – Custom request ID (UUID by default).

Returns:

- `dict` – Serialized request object.

Source code in `python/neuro_rpc/RPCHandler.py`

```
def create_request(self, method, params=None, request_id=None): """ Create a JSON-Message request object. Args: method (str): Met
```

`create_response(result, request_id)`

Create a JSON-Message response object.

Parameters:

- **`result`** (`Any`) – The result to return.
- **`request_id`** (`str`) – ID of the original request.

Returns:

- `dict` – Serialized response object.

Source code in `python/neuro_rpc/RPCHandler.py`

```
def create_response(self, result, request_id): """ Create a JSON-Message response object. Args: result (Any): The result to retur
```

`next_request_id()`

Generate a new request ID.

Returns:

- **`int`** (`int`) – Incremental request ID.

Source code in `python/neuro_rpc/RPCHandler.py`

```
def next_request_id(self) -> int: """ Generate a new request ID. Returns: int: Incremental request ID. """ self._request_id += 1
```

`process_message(message)`

Process an incoming JSON-Message.

Parses input (string/dict/RPCMessage), converts to RPCRequest or RPCResponse, and dispatches to the appropriate handler.

Parameters:
- **`message`** (`dict | str | RPCMessage`) – Incoming message.

Returns:
- `Optional[Dict[str, Any]]` – dict | None: Response dict if request, None if response.

Raises:
- `RPCError` – If message is invalid or cannot be parsed.

Source code in `python/neuro_rpc/RPCHandler.py`

```
def process_message(self, message: Union[Dict[str, Any], str, RPCMessage]) -> Optional[Dict[str, Any]]: """ Process an incoming J
```

`register_methods(instance)`

Register decorated methods from an instance.

Scans instance methods and registers those annotated with `@rpc_method`.

Parameters:
- **`instance`** (`Any`) – Object instance containing decorated methods.

Source code in `python/neuro_rpc/RPCHandler.py`

```
def register_methods(self, instance) -> None: """ Register decorated methods from an instance. Scans instance methods and registe
```

`register_request(method_name, method)`

Register a request handler.

Parameters:
- **`method_name`** (`str`) – Name of the RPC method.
- **`method`** (`Callable`) – Function to call when this request is received.

Raises:
- `ValueError` – If the provided method is not callable.

Source code in `python/neuro_rpc/RPCHandler.py`

```
def register_request(self, method_name: str, method: Callable) -> None: """ Register a request handler. Args: method_name (str):
```

`register_response(method_name, method)`

Register a response handler.

Parameters:
- `method_name` (`str`) – Name of the RPC method.
- `method` (`Callable`) – Function to call when a response is received.

Raises:
- `ValueError` – If the provided method is not callable.

Source code in `python/neuro_rpc/RPCHandler.py`

```
def register_response(self, method_name: str, method: Callable) -> None: """ Register a response handler. Args: method_name (str)
```

2.6.2 `rpc_method(method_type='both', name=None)`

Decorator to mark methods for RPC registration.

Annotates a function so that `RPCHandler.register_methods()` can discover and register it as a request and/or response handler.

Parameters:
- `method_type` (`str`, default: `'both'`) – One of {"request", "response", "both"} (default "both").
- `name` (`str`, default: `None`) – Optional alias under which the method is registered.

Returns:
- `Callable` – The decorated function.

Source code in `python/neuro_rpc/RPCHandler.py`

```
def rpc_method(method_type: str = "both", name: Optional[str] = None): """ Decorator to mark methods for RPC registration. Annota
```

2.7 RPCMessage

Message and error classes for JSON-Message 2.0 protocol.

Provides base classes for JSON-Message 2.0 communication (similar to JSON-RPC 2.0): - RPCError: structured error objects. - RPCMessage: base class for all messages. - RPCRequest: request with method, params, and id. - RPCResponse: response with result or error.

Notes

- Ensures compatibility with NeuroRPC stack (Client, RPCHandler, Benchmark).

2.7.1 `RPCError(error_type=None, data=None)`

Bases: `Exception`

Exception class for JSON-Message 2.0 errors.

Encapsulates standard and implementation-specific error codes as structured dictionaries, used for request/response validation.

Initialize RPCError with a given type and optional metadata.

Parameters:

- **`error_type`** (`str | dict`, default: `None`) – One of the error constants or a full error dict.
- **`data`** (`Any`, default: `None`) – Additional metadata attached as "metadata" field.

Source code in `python/neuro_rpc/RPCMessage.py`

```
def __init__(self, error_type=None, data: Any = None): """ Initialize RPCError with a given type and optional metadata. Args: err
```

2.7.2 `RPCMessage()`

Base class for JSON-Message 2.0 messages.

Defines the `jsonrpc` version and common serialization/deserialization helpers.

Initialize with version '2.0'.

Source code in `python/neuro_rpc/RPCMessage.py`

```
def __init__(self): """Initialize with version '2.0'.""" self.jsonrpc = "2.0"
```

`from_dict(data)` `classmethod`

Validate and create a message from dictionary.

Parameters:
- `data` (`dict`) – Dictionary to parse.

Returns:
- `RPCMessage` (`RPCMessage`) – Instance of the base class.

Raises:
- `RPCError` – If input is not a dict or version is invalid.

Source code in `python/neuro_rpc/RPCMessage.py`

```
@classmethod def from_dict(cls, data: Dict[str, Any]) -> 'RPCMessage': """ Validate and create a message from dictionary. Args: d
```

`from_json(json_str)` `classmethod`

Create message from JSON string.

Parameters:
- `json_str` (`str`) – Input JSON string.

Returns:
- `RPCMessage` (`RPCMessage`) – Parsed object.

Raises:
- `RPCError` – If parsing fails.

Source code in `python/neuro_rpc/RPCMessage.py`

```
@classmethod def from_json(cls, json_str: str) -> 'RPCMessage': """ Create message from JSON string. Args: json_str (str): Input
```

`to_dict()`

Serialize the message to a dictionary.

Returns:
- `dict` (`Dict[str, Any]`) – Dictionary containing the `jsonrpc` version.

Source code in `python/neuro_rpc/RPCMessage.py`

```
def to_dict(self) -> Dict[str, Any]: """ Serialize the message to a dictionary. Returns: dict: Dictionary containing the ``jsonrp
```

`to_json()`

Serialize the message to a JSON string.

Returns:
- `str` (`str`) – JSON string with message content.

Source code in `python/neuro_rpc/RPCMessage.py`

```
def to_json(self) -> str: """ Serialize the message to a JSON string. Returns: str: JSON string with message content. """ return
```

2.7.3 `RPCRequest(method, id=None, params=None)`

Bases: `RPCMessage`

JSON-Message 2.0 Request.

Contains method name, parameters, and identifier (id). Supports both positional (list) and named (dict) parameters.

Parameters:

- **`method`** (`str`) – Method name to call.
- **`id`** (`Any`, default: `None`) – Identifier for correlation (None for notifications).
- **`params`** (`dict | list`, default: `None`) – Parameters for the call.

Source code in `python/neuro_rpc/RPCMessage.py`

```
def __init__(self, method: str, id: Any = None, params: Optional[Union[Dict, List]] = None): """ Args: method (str): Method name
```

`is_notification` `property`

Check if this request is a notification.

Notifications do not have an id and therefore do not expect a response.

Returns:

- **`bool`** (`bool`) – True if id is None.

`from_dict(data)` `classmethod`

Create a request from dictionary.

Parameters:

- **`data`** (`dict`) – Input dictionary.

Returns:

- **`RPCRequest`** (`RPCRequest`) – Parsed request.

Raises:

- `RPCError` – If validation fails.

Source code in `python/neuro_rpc/RPCMessage.py`

```
@classmethod def from_dict(cls, data: Dict[str, Any]) -> 'RPCRequest': """ Create a request from dictionary. Args: data (dict): I
```

`to_dict()`

Serialize the request to a dictionary.

Returns:

- `dict` (`Dict[str, Any]`) – Request with jsonrpc, method, id, and params.

Source code in `python/neuro_rpc/RPCMessage.py`

```
def to_dict(self) -> Dict[str, Any]: """ Serialize the request to a dictionary. Returns: dict: Request with jsonrpc, method, id,
```

2.7.4 `RPCResponse(id, result=None, error=None, exec_time=None)`

Bases: `RPCMessage`

JSON-Message 2.0 Response.

Contains either a `result` or an `error`, but never both. Optionally includes execution time (exec_time) for benchmarking.

Parameters:

- `id` (`Any`) – ID of the original request.
- `result` (`Any` , default: `None`) – Result of the request.
- `error` (`dict` , default: `None`) – Error object.
- `exec_time` (`int` , default: `None`) – Execution time (µs, provided by server).

Raises:

- `RPCError` – If both result and error are provided.

Source code in `python/neuro_rpc/RPCMessage.py`

```
def __init__( self, id: Any, result: Any = None, error: Optional[Dict[str, Any]] = None, exec_time: Optional[int] = None, ): """
```

`is_error` `property`

Check if this response is an error response.

Returns:

- `bool` (`bool`) – True if error is not None.

`is_success` `property`

Check if this response is a success response.

Returns:

- `bool` (`bool`) – True if error is None.

`from_dict(data)` `classmethod`

Create a response from dictionary.

Parameters:
- `data` (`dict`) – Input dictionary.

Returns:
- `RPCResponse` (`RPCResponse`) – Parsed response.

Raises:
- `RPCError` – If validation fails.

Source code in `python/neuro_rpc/RPCMessage.py`

```
@classmethod def from_dict(cls, data: Dict[str, Any]) -> 'RPCResponse': """ Create a response from dictionary. Args: data (dict):
```

`to_dict()`

Serialize the response to a dictionary.

Returns:
- `dict` (`Dict[str, Any]`) – Response with jsonrpc, id, and either result or error.

Source code in `python/neuro_rpc/RPCMessage.py`

```
def to_dict(self) -> Dict[str, Any]: """ Serialize the response to a dictionary. Returns: dict: Response with jsonrpc, id, and ei
```

2.8 RPCMethods

Example RPC methods built on top of RPCHandler.

Provides a container of request/response methods for testing and demonstration. Includes echo, add, subtract, and a default response handler. Can be extended with custom RPC logic as needed.

Notes

- Uses the @rpc_method decorator to auto-register methods with RPCHandler.

2.8.1 `RPCMethods(auto_register=True)`

Bases: `RPCHandler`

Container for RPC methods.

Extends RPCHandler and defines example request/response methods that are automatically registered at initialization if `auto_register=True`.

Initialize the RPCMethods container.

Parameters:

- **`auto_register`** (`bool`, default: `True`) – If True, automatically registers decorated methods.

Source code in `python/neuro_rpc/RPCMethods.py`

```
def __init__(self, auto_register: bool = True): """ Initialize the RPCMethods container. Args: auto_register (bool): If True, aut
```

`add(a, b)`

RPC request method: add two numbers.

Parameters:

- **`a`** (`float`) – First number.
- **`b`** (`float`) – Second number.

Returns:

- **`float`** (`float`) – Sum of a and b.

Source code in `python/neuro_rpc/RPCMethods.py`

```
@rpc_method(method_type="request") def add(self, a: float, b: float) -> float: """ RPC request method: add two numbers. Args: a (
```

`default_response_handler(id=None, result=None, error=None)`

Default response handler.

Invoked if no specific handler is registered for a response.

Parameters:

- `id` (`Any` , default: `None`) – ID of the response.
- `result` (`Any` , default: `None`) – Result payload if success.
- `error` (`Any` , default: `None`) – Error payload if failure.

Source code in `python/neuro_rpc/RPCMethods.py`

```
@rpc_method(method_type="response", name="default") def default_response_handler(self, id: Any = None, result: Any = None, error:
```

`echo(message)`

RPC request method: echo a message.

Parameters:

- `message` (`str`) – String to echo back.

Returns:

- `str` (`str`) – The same message received.

Source code in `python/neuro_rpc/RPCMethods.py`

```
@rpc_method(method_type="request") def echo(self, message: str) -> str: """ RPC request method: echo a message. Args: message (st
```

`handle_add_response(id=None, result=None, error=None)`

Response handler for add.

Parameters:

- `id` (`Any` , default: `None`) – ID of the corresponding request.
- `result` (`Any` , default: `None`) – Result (sum).
- `error` (`Any` , default: `None`) – Error object if the request failed.

Source code in `python/neuro_rpc/RPCMethods.py`

```
@rpc_method(method_type="response", name="add") def handle_add_response(self, id: Any = None, result: Any = None, error: Any = No
```

`handle_echo_response(id=None, result=None, error=None)`

Response handler for echo.

Parameters:

- `id` (`Any` , default: `None`) – ID of the corresponding request.
- `result` (`Any` , default: `None`) – Result content.
- `error` (`Any` , default: `None`) – Error object if the request failed.

Source code in `python/neuro_rpc/RPCMethods.py`

```
@rpc_method(method_type="response", name="echo") def handle_echo_response(self, id: Any = None, result: Any = None, error: Any =
```

`handle_subtract_response(id=None, result=None, error=None)`

Response handler for subtract.

Parameters:

- `id` (`Any` , default: `None`) – ID of the corresponding request.
- `result` (`Any` , default: `None`) – Result (difference).
- `error` (`Any` , default: `None`) – Error object if the request failed.

Source code in `python/neuro_rpc/RPCMethods.py`

```
@rpc_method(method_type="response", name="subtract") def handle_subtract_response(self, id: Any = None, result: Any = None, error
```

`subtract(a, b)`

RPC request method: subtract b from a.

Parameters:

- `a` (`float`) – Minuend.
- `b` (`float`) – Subtrahend.

Returns:

- `float` (`float`) – Result of a - b.

Source code in `python/neuro_rpc/RPCMethods.py`

```
@rpc_method(method_type="request") def subtract(self, a: float, b: float) -> float: """ RPC request method: subtract b from a. An
```

2.9 RPCTracker

Message tracking utility for JSON-Message 2.0 protocol.

Tracks outgoing/incoming requests and responses, monitors for timeouts, keeps statistics, and runs an optional background monitoring thread. Designed for integration with RPCHandler and Benchmark to provide runtime visibility of pending and completed RPC calls.

Notes

- Thread-safe using locks.
- Intended for long-running client/server sessions.

2.9.1 `RPCTracker(monitor_interval=1, cleanup_interval=60, autostart=True)`

Tracks request/response lifecycle for RPC messages.

Maintains dictionaries of outgoing/incoming requests and responses, updates statistics, and detects timeouts via a background thread.

Initialize RPCTracker.

Parameters:

- `monitor_interval` (`int`, default: `1`) – Interval in seconds to check for timeouts.
- `cleanup_interval` (`int`, default: `60`) – Interval in seconds to clean old entries.
- `autostart` (`bool`, default: `True`) – Whether to immediately start monitoring.

Source code in `python/neuro_rpc/RPCTracker.py`

```
def __init__(self, monitor_interval=1, cleanup_interval=60, autostart=True): """ Initialize RPCTracker. Args: monitor_interval (i
```

`clean_tracking_data(max_age_seconds=3600)`

Remove old tracking entries.

Parameters:

- `max_age_seconds` (`int`, default: `3600`) – Max age in seconds to keep entries.

Returns:

- `int` – Number of entries cleaned.

Source code in `python/neuro_rpc/RPCTracker.py`

```
def clean_tracking_data(self, max_age_seconds=3600): """ Remove old tracking entries. Args: max_age_seconds (int): Max age in sec
```

`get_statistics()`

Get current statistics snapshot.

Returns:

- `dict` – Copy of statistics counters.

Source code in `python/neuro_rpc/RPCTracker.py`

```
def get_statistics(self): """ Get current statistics snapshot. Returns: dict: Copy of statistics counters. """ with self._trackin
```

`monitor_messages()`

Inspect current requests for timeouts and pending states.

Returns:

- `dict` – Dictionary with lists of timed-out and pending requests.

Source code in `python/neuro_rpc/RPCTracker.py`

```
def monitor_messages(self): """ Inspect current requests for timeouts and pending states. Returns: dict: Dictionary with lists of
```

`start_monitoring(timeout_callback=None)`

Start the background monitoring thread.

Parameters:

- `timeout_callback` (`Callable`, default: `None`) – Callback called with a list of timed-out requests.

Returns:

- `bool` – True if started, False if already running.

Source code in `python/neuro_rpc/RPCTracker.py`

```
def start_monitoring(self, timeout_callback=None): """ Start the background monitoring thread. Args: timeout_callback (Callable,
```

`stop_monitoring()`

Stop the background monitoring thread.

Returns:

- `bool` – True if stopped cleanly, False otherwise.

Source code in `python/neuro_rpc/RPCTracker.py`

```
def stop_monitoring(self): """ Stop the background monitoring thread. Returns: bool: True if stopped cleanly, False otherwise. "
```

`track_incoming_request(request)`

Track an incoming request from server.

Parameters:
- `request` (`RPCRequest`) – Request object received.

Source code in `python/neuro_rpc/RPCTracker.py`

```
def track_incoming_request(self, request: RPCRequest): """ Track an incoming request from server. Args: request (RPCRequest): Rec
```

`track_incoming_response(response)`

Track an incoming response from server.

Parameters:
- `response` (`RPCResponse`) – Response object received.

Source code in `python/neuro_rpc/RPCTracker.py`

```
def track_incoming_response(self, response: RPCResponse): """ Track an incoming response from server. Args: response (RPCResponse
```

`track_outgoing_request(request, timeout=60)`

Track an outgoing request.

Parameters:
- `request` (`RPCRequest`) – Request object being sent.
- `timeout` (`int` , default: `60`) – Timeout in seconds for this request.

Source code in `python/neuro_rpc/RPCTracker.py`

```
def track_outgoing_request(self, request: RPCRequest, timeout=60): """ Track an outgoing request. Args: request (RPCRequest): Rec
```

`track_outgoing_response(response)`

Track an outgoing response.

Parameters:
- `response` (`RPCResponse`) – Response object being sent.

Source code in `python/neuro_rpc/RPCTracker.py`

```
def track_outgoing_response(self, response: RPCResponse): """ Track an outgoing response. Args: response (RPCResponse): Response
```