Gearthonic Documentation

Release 0.2.0

Timo Steidle

August 11, 2016

Contents

A simple client for the API of Homegear.

Look at the documentation for detailed information.

CHAPTER 1

Quickstart

Install *gearthonic* via pip:

```
pip install gearthonic
```

Initialise the client:

```
from gearthonic import GearClient
# You only have to provide the host and port of the Homegear server
gc = GearClient('192.168.1.100', 2001, secure=False, verify=False)
```

Use the predefined methods to make requests to the API:

```
gc.system.list_methods()
gc.device.get_value(1, 4, 'ACTUAL_TEMPERATURE')
```

Alternatively you can call any method directly via the client:

```
gc.getValue(1, 4, 'ACTUAL_TEMPERATURE')
```

The default communication protocol is XML-RPC. If you want to use another protocol like JSON-RPC or a MQTT broker, see the full documentation.

CHAPTER 2

Contents:

2.1 Usage

2.1.1 Basic usage

To use Gearthonic in a project:

```
from gearthonic import GearClient
# Initialise the client with the host and port of your homegear server
gc = GearClient('192.168.1.100', 2003)
# Now you can make your requests
gc.system.list_methods()
gc.device.get_value(1, 4, 'ACTUAL_TEMPERATURE')
```

All methods are already implemented within the client to make it easy to use. You don't have to lookup all methods and their parameters, just look at the code or see Method Reference.

Alternatively you can call any method directly via the client or use the generic ''call''-method:

```
gc.getValue(1, 4, 'ACTUAL_TEMPERATURE')
gc.call('system.listDevices')
gc.call('getValue', 1, 4, 'ACTUAL_TEMPERATURE')
```

A full list of all available methods provided by the Homegear API can be found in the wiki of the Homegear project.

2.1.2 Communication protocols

Gearthonic supports different communication protocols to communicate with your Homegear server. Set the protocol while initialising the client:

```
from gearthonic import JSONRPC
gc = GearClient('192.168.1.100', 2003, protocol=JSONRPC)
```

The default protocol is `XMLRPC`.

Any protocol can accept additional parameters. You can supply them while initialising the client:

```
gc = GearClient('192.168.1.100', 2003, secure=False, verify=False)
```

For a complete list of available protocols and supported parameters look at *gearthonic.protocols*.

2.1.3 Security

XML RPC and JSON RPC

If you set `secure=True` while initialising the GearClient, the client tries to establish a secure connection to the server via SSL. It's highly recommended to use SSL for the network traffic. Otherwise the communication is not encrypted.

If you are using a self signed certificate, you can skip the verification of the certificate. Set `verify=False` while initialising the GearClient and the certificate won't be verified. But keep in mind: that's not suggested!

2.1.4 Additional information

- Documentation of all available data endpoints for all devices
- wiki of the Homegear project

2.2 API

2.2.1 API Reference

See Method Reference for all API methods.

GearClient

Client for the APIs provided by Homegear.

class `gearthonic.client.`**`GearClient`**(*host*, *port*, *protocol=0*, ***kwargs*)

Client for the APIs provided by Homegear.

Usage:

```
>>> gc = GearClient('localhost', 1234)
>>> gc.device.list_devices()
```

The default communication protocol is `XML RPC`. Additionally, `JSON RPC` is supported:

```
>>> from gearthonic import JSONRPC
>>> GearClient('localhost', 1234, protocol=JSONRPC)
```

Any protocol can accept additional parameters. Provide them while initialising the client:

```
>>> GearClient('localhost', 1234, protocol=JSONRPC, secure=False, username='ham')
```

For a full list of supported parameters, have a look at each protocol class or see the documentation for the protocols. :class:

`call`(*method_name*, **args*, ***kwargs*)

Call the given method using the instantiated protocol.

Parameters

- **`method_name`** – method to be called
- **`args`** – arguments passed through
- **`kwargs`** – keyword arguments passed through

Returns return value of the call

device
Smart access to all device related API methods.

system
Smart access to all system related API methods.

Communication protocols

These classes are used to communicate with the different APIs provided by the Homegear server.

class gearthonic.protocols.**JsonRpcProtocol**(*host*, *port*, *secure=True*, *verify=True*, *username=None*, *password=None*)
Communicate with Homegear via JSON RPC.

```
>>> jp = JsonRpcProtocol('host.example.com', 2003)
>>> jp.call('listDevices')
[...]
```

Set `secure=False` to use `http` instead off `https`. Set `verify=False` to skip the verification of the SSL cert.

Provide credentials via `username` and `password` if the Homegear server is secured by basic auth. It's not possible to use authentication with an insecure connection (http)!

call(*method_name*, **args*, ***kwargs*)
Call the given method using the HTTPServer.

Parameters

- **method_name** – Method to be called
- **args** – Arguments passed through
- **kwargs** – Keyword arguments passed through

Returns Return value of the XML RPC method

class gearthonic.protocols.**XmlRpcProtocol**(*host*, *port*, *secure=True*, *verify=True*, *username=None*, *password=None*)
Communicate with Homegear via XML RPC.

```
>>> xp = XmlRpcProtocol('host.example.com', 2003)
>>> xp.call('listDevices')
[...]
```

Set `secure=False` to use `http` instead off `https`. Set `verify=False` to skip the verification of the SSL cert.

Provide credentials via `username` and `password` if the Homegear server is secured by basic auth. It's not possible to use authentication with an insecure connection (http)!

call(*method_name*, **args*, ***kwargs*)
Call the given method using the ServerProxy.

Parameters

- **method_name** – Method to be called
- **args** – Arguments passed through
- **kwargs** – Keyword arguments passed through

Returns Return value of the XML RPC method

gearthonic.protocols.**initialise_protocol**(*protocol*, *host*, *port*, ***kwargs*)
Factory method to initialise a specific protocol.

Parameters

- **protocol** (*int*) – ID of the protocol to initialise
- **host** (*str*) – host of the server
- **port** (*int*) – port of the server
- **kwargs** – will be used to initialise the protocol

Return type _ProtocolInterface

2.2.2 Method Reference

Includes all functions provided by the XML RPC API, split into logical entities.

System methods

class gearthonic.methods.**SystemMethodsCollection**(*caller*)
All XML RPC server provide a set of standard methods.

get_capabilities()
Lists server's XML RPC capabilities.

Example output:

```
{
    'xmlrpc': {'specUrl': 'http://example.com', 'specVersion': 1}
    'faults_interop': {'specUrl': 'http://example2.com', 'specVersion': 101}
    'introspection': {'specUrl': 'http://example3.com', 'specVersion': 42}
}
```

Returns A dict containing all information

Return type dict

list_methods()
Lists servers's XML RPC methods.

Returns A list of available methods

Return type list

method_help(*method_name*)
Returns the description of a method.

Parameters **method_name** (*str*) – The name of the method

Returns The description of the method

Return type str

method_signature(*method_name*)
Returns the signature of a method.

Parameters **method_name** (*str*) – The name of the method

Returns The signature of the method

Return type list

multicall(*methods*)
Call multiple methods at once.

Example list of `methods`:

```
[
    {'methodName': 'getValue', 'params': [13, 4, 'TEMPERATURE']},
    {'methodName': 'getValue', 'params': [3, 3, 'HUMIDITY']}
]
```

Return value of the multicall:

```
[22.0, 58]
```

Parameters **methods** (*list*) – A list of methods and their parameters

Returns A list of method responses.

Return type list

General methods

class gearthonic.methods.**GeneralMethodsCollection**(*caller*)
All general methods.

get_service_messages(*return_id*)
Return all service messages.

This method returns all service messages that are currently active in Homegear (device unreachable, config pending, low battery, sabotage, ...).

Parameters **return_id** (*bool*) – Recommended. If true, Homegear returns the peer ID instead of the "address" (serial number and channel separated by a colon). By default, the address is returned for compatibility reasons.

get_version()
Return Homegear's version number.

log_level(*level=None*)
Get or set the current log level.

Valid values are:

- 0: Log nothing
- 1: Critical
- 2: Error
- 3: Warning
- 4: Info
- 5: Debug

Warning: Don't use "debug" for normal operation as it slows down Homegear.

Parameters **level** (*int*) – (optional) This is the log level that you want to set.

write_log(*message*, *log_level=None*)
Write a message to the Homegear log.

This method writes a message to Homegear's log. It's possible to set the log level. Valid values are:

- 1: Critical
- 2: Error
- 3: Warning
- 4: Info
- 5: Debug

Parameters

- **message** (*str*) – This is the message you want to write to the log file. The date is automatically prepended.
- **log_level** (*int*) – (optional) This is the log level of the message. If Homegear's log level value is lower than this value, no message is logged.

Device methods

class gearthonic.methods.**DeviceMethodsCollection**(*caller*)
All device related methods.

activate_link_paramset(*peer_id*, *channel*, *remote_peer_id*, *remote_channel*, *long_press=False*)
Simulate a remote key press.

This method can be used to simulate a remote key press. For most cases, *set_value()* should be sufficient. Use *activate_link_paramset* if you want to execute commands that can be configured only using the link parameter set.

Parameters

- **peer_id** (*int*) – The ID of the actuator
- **channel** (*int*) – The channel of the actuator
- **remote_peer_id** (*int*) – The ID of a peer linked to the actuator or "0" if you want to use a virtual peer
- **remote_channel** (*int*) – The channel of a remote peer linked to the peer; use the same "channel" value if a virtual peer is specified.
- **long_press** (*bool*) – (optional) Set to "True" to simulate a long key press. The default setting is "False".

add_link(*sender_id*, *sender_channel*, *receiver_id*, *receiver_channel*, *name=''*, *description=''*)
Create a link between two devices.

This method links two devices so that they can send commands to each other directly.

Parameters

- **sender_id** (*int*) – The ID of the sending peer (e.g. a remote)
- **sender_channel** (*int*) – The channel of the sending peer or "-1"
- **receiver_id** (*int*) – The ID of the receiving peer (e.g. a switch)
- **receiver_channel** (*int*) – The channel of the receiving peer or "-1"

- **name** (*str*) – (optional) A descriptive name for the link
- **description** (*str*) – (optional) A short description of the link

get_all_config(*peer_id=None*)
Return all peer configuration parameters and some additional metadata.

This method returns all configuration parameter values and information about all configuration parameters for one or all peers. Variables are not returned. To get all variables, call *get_all_values()*.

Parameters **peer_id** (*int*) – (optional) When specified, only variables of this peer are returned.

get_all_values(*peer_id=None*, *return_write_only_variables=False*)
Return all peer configuration parameters and some additional metadata.

This method returns all variable values and information about all variables for one or all peers. Configuration parameters are not returned. To get all configuration parameters, call *get_all_config()*.

Parameters

- **peer_id** (*int*) – (optional) When specified, only variables of this peer are returned.
- **return_write_only_variables** (*bool*) – (optional) When specifed, write only variables are also returned.

get_value(*peer_id*, *channel*, *key*, *request_from_device=False*, *asynchronous=False*)
Return the value of the device, specified by channel and key (parameterName).

Per default the value is read from the local cache of Homegear. If the value should be read from the device, use `request_from_device`. If the value should be read from the device, this can be done asynchronously. The method returns immediately and doesn't wait for the current value. The value will be sent as an event as soon as it's returned by the device.

Error codes:

•Returns `-2` when the device or channel is unknown

•Returns `-5` when the key (parameter) is unknown

Parameters

- **peer_id** (*int*) – ID of the device
- **channel** (*int*) – Channel of the device to get the value for
- **key** (*str*) – Name of the parameter to get the value for
- **request_from_device** (*bool*) – If true value is read from the device
- **asynchronous** (*bool*) – If true value is read asynchronously

Returns The value of the parameter or error code

Return type unknown

list_devices()
Return a list of devices.

Returns List of devices

Return type list

set_value(*peer_id*, *channel*, *key*, *value*)
Set the value of the device, specified by channel and key (parameterName).

Parameters

- **peer_id** (*int*) – ID of the device
- **channel** (*int*) – Channel of the device to set the value for
- **key** (*str*) – Name of the parameter to get the value for
- **value** (*unknown*) – The value to set

Returns

- `None` on success
- `-2` when the device or channel is unknown
- `-5` when the key (parameter) is unknown
- `-100` when the device did not respond
- `-101` when the device returns an error

Pairing methods

class gearthonic.methods.**PairingMethodsCollection**(*caller*)

All pairing related methods.

add_device(*serial_number*, *family_id=None*)

Pair a device without enabling pairing mode.

This method pairs a device by its serial number, but this does not work for all devices.

Parameters

- **serial_number** (*str*) – The serial number of the device to be paired
- **family_id** (*int*) – (optional) ID of the family you want to add the device to; if not specified, "addDevice" is executed for all device families that support it.

create_device(*family_id*, *device_type*, *serial_number*, *address*, *firmware_version*)

Create a device manually.

This method manually creates a new device. It is not supported by all device families, and it is also not supported for all devices. createDevice can be used to create virtual devices in the family "Miscellaneous".

Parameters

- **family_id** (*int*) – This is the ID of the family you want to create the device in. See: *list_families()*.
- **device_type** (*int*) – The type ID of the device as specified in the device's XML file
- **serial_number** (*str*) – The serial number of the new device
- **address** (*int*) – This is the physical address of the new device. Depending on the device family, this parameter might be optional. If it is not needed, set it to "-1".
- **firmware_version** (*int*) – This is the firmware version of the new device. Depending on the device family, this parameter might be optional. If the firmware version is "1.2", set this variable to 0x12 = 18. If it is not needed, set it to "-1".

get_install_mode(*family_id=None*)

Return the time left in pairing mode.

This method returns the remaining amount of time the central will be in pairing mode.

Parameters **family_id** (*int*) – (optional) This is the ID of the family for which you want to get the remaining time in pairing mode. If not specified, the remaining time in pairing mode of the first central for which pairing mode enabled is returned.

get_pairing_methods (*family_id*)
Return the pairing methods supported by a device family.

This method returns all pairing methods supported by the specified device family.

Parameters **family_id** (*int*) – The ID of the family for which you want to get the supported pairing methods

search_devices (*family_id=None*)
Start a device search for all supported device families.

When you use this method, Homegear searches for new devices in all device families that support the method.

Parameters **family_id** (*int*) – (optional) This is the ID of the family that you want to search for devices.

set_install_mode (*on*, *family_id=None*, *duration=60*)
Enable pairing mode.

This method enables or disables pairing mode for all device families if it is supported by the device family.

Parameters

- **on** (*bool*) – When this is true, pairing mode is enabled. Otherwise, pairing mode is disabled.
- **family_id** (*int*) – (optional) This is the ID of the family for which you want to enable pairing mode. If it is not specified, pairing mode will be enabled for all device families.
- **duration** (*int*) – (optional) This is the duration in seconds that the central should remain in pairing mode. The minimum duration is 5 seconds, and the maximum duration is 3600 seconds. The default duration is 60 seconds.

Family methods

class gearthonic.methods.**FamilyMethodsCollection** (*caller*)
All pairing related methods.

list_families ()
Return information about all device families (ID, name, pairing methods).

This method returns information about all available device families. Use this method to get the ID of a family if you have only the name or only the ID. You can also use this method to get the pairing methods supported by the family.

Event Server methods

class gearthonic.methods.**EventServerMethodsCollection** (*caller*)
All EventServer related methods

client_server_initialized (*interface_id*)
Check if an RPC client's RPC server was successfully registered and if it still is registered.

This method checks if an RPC client's RPC server is registered and connected to Homegear. You can register your RPC "event" server by calling *init()*.

Parameters **interface_id** (*str*) – The interface ID as specified in *init()*

init(*url*, *interface_id*, *flags=None*)
Register a client's RPC server with Homegear to receive events.

This method is used to register or unregister an RPC event server with Homegear. After calling this method, Homegear's RPC client starts sending events and device updates to the registered server. It is not necessary to call "init" for MQTT or WebSockets.

It's possible to configure the communication between Homegear and the client's RPC server by using *flags*. The following (bitmask) flags are available:

•0x01: keepAlive: Do not close the connection after each packet.

•0x02: binaryMode: Send RPC data in binary format. Equivalent to "binary://" or "binarys://".

•0x04: newFormat: (Recommended) Send device's ID in broadcast methods instead of the serial number and activates variable types ARRAY and STRUCT. This is recommended because serial numbers are not necessarily unique.

•0x08: subscribePeers: If this is set, Homegear will send events only for peers subscribed with *subscribePeers* to the event server.

•0x10: jsonMode: Send RPC data in JSON format.

So if you want to enable *binaryMode* and *subscribePeers*, you have to provide *10*. If you want to set *newFormat* additionally, provide *14*. And if you want to enable *jsonMode* additionally, provide *30*.

Parameters

- **url** (*str*) – The URL of the event server that you want to register, including "http://" and the port. If you use "binary://", RPC data is sent in binary format. If you pass "https://" or "binarys://", SSL is enabled.
- **interface_id** (*str*) – This is an arbitrary name for the interface. To unregister an event server, pass an empty string to interfaceId.
- **flags** (*int*) – (optional) Used to configure the communication between Homegear and the registered server.

list_client_servers(*interface_id=None*)
Return information about all RPC servers registered with Homegear by clients.

This method returns an array with one entry for each RPC server registered with Homegear.

Parameters **interface_id** (*str*) – (optional) This is the interface ID of the RPC server as it was passed to *init()*. If it is specified, only the information for this server is returned.

subscribe_peers(*event_server_id*)
Subscribe peer events that are to be sent to an event server.

This method is used to subscribe peer events after calling *init()* with the *subscribe_peers* flag set.

Parameters **event_server_id** (*str*) – This is either the url specified in *init()* or the WebSocket client ID.

trigger_rpc_event(*event_method*)
Send an RPC event to all RPC event servers.

This method manually calls an RPC event method on all RPC event servers. Currently supported methods are *deleteDevices*, *newDevices* and *updateDevice*.

Parameters **event_method** (*str*) – This is the method you want to call.

unsubscribe_peers(*event_server_id*)
Unsubscribe peer events.

This method is used to unsubscribe peer events after *subscribe_peers()* has been called.

Parameters **event_server_id** (*str*) – This is either the url specified in *init()* or the WebSocket client ID.

Physical Interface methods

class gearthonic.methods.**PhysicalInterfaceMethodsCollection**(*caller*)
All methods related to the physical interface.

list_bidcos_interfaces()
Exist only for compatibility reasons.

This method exists only for reasons of backward compatibility with the CCU and has no real function.

list_interfaces(*family_id=None*)
List all physical interfaces with status information.

This method returns a list of all physical interfaces. It can be used to determine if an interface is available.

Parameters **family_id** (*int*) – (optional) The ID of the family for which you want to get interfaces

set_interface(*peer_id*, *interface_id*)
Set the physical interface Homegear uses to communicate with a peer.

This method sets the physical interface that Homegear is to use to communicate with a peer.

Parameters

- **peer_id** (*int*) – The ID of the peer you want to set the interface for
- **interface_id** (*str*) – This is the ID of the physical interface as defined in the family interface settings. If it is empty, the physical interface is reset to the default interface.

Metadata methods

class gearthonic.methods.**MetadataMethodsCollection**(*caller*)
All metadata related methods.

delete_metadata(*peer_id*, *data_id=None*)
Delete previously stored metadata.

Parameters

- **peer_id** (*int*) – The ID of the peer for which the metadata is stored.
- **data_id** (*str*) – (optional) The dataId

get_all_metadata(*peer_id*)
Return all the metadata of one peer.

Parameters **peer_id** (*int*) – The ID of the peer for which you want to get metadata

get_metadata(*peer_id*, *data_id*)
Retrieve previously stored metadata.

This method returns metadata that was previously stored with `setMetadata()`.

Parameters

- **peer_id** (*int*) – The ID of the peer for which you want to get the metadata
- **data_id** (*str*) – The data ID

set_metadata(*peer_id*, *data_id*, *value*)
Store metadata for a peer.

This method can be used to store metadata for devices in Homegear's database. You can retrieve this metadata later by calling *get_metadata()*.

Parameters

- **peer_id** (*int*) – The ID of the peer for which you want to store metadata
- **data_id** (*str*) – A name of your choice
- **value** – The value you want to store

System variables methods

class gearthonic.methods.**SystemVariableMethodsCollection**(*caller*)
All system variables related methods.

delete_system_variable(*name*)
Delete a system variable.

This method deletes a system variable created with *set_system_variable()*.

Parameters **name** (*str*) – The name of the system variable to be deleted

get_all_system_variables()
Return all system variables.

get_system_variable(*name*)
Get the value of a system variable.

This method returns a system variable's value that was previously stored with *set_system_variable()*.

Parameters **name** (*str*) – The name of the system variable

set_system_variable(*name*, *value*)
Create or update a system variable.

This method can be used to store arbitrary data in Homegear's database. You can retrieve this data later by calling `getSystemVariable()`.

Parameters

- **name** (*str*) – A name of your choice
- **value** – The value to be stored

2.3 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.3.1 Types of Contributions

Report Bugs

Report bugs at https://gitlab.com/wumpitz/gearthonic/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the Gitlab issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the Gitlab issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Gearthonic could always use more documentation, whether as part of the official gearthonic docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://gitlab.com/wumpitz/gearthonic/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.3.2 Get Started!

Ready to contribute? Here's how to set up *gearthonic* for local development.

1. Fork the *gearthonic* repo on Gitlab.
2. Clone your fork locally:

```
$ git clone git@gitlab.com:your_name_here/gearthonic.git
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

4. When you're done making changes, check that your changes pass style and unit tests, including testing other Python versions with tox:

```
$ tox
```

To get tox, just pip install it.

5. Commit your changes and push your branch to Gitlab:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the Gitlab website.

2.3.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7 and 3.5, and for PyPy. Run the `tox` command and make sure that the tests pass for all supported Python versions.

2.3.4 Tips

To run a subset of tests:

```
$ py.test test/test_gearthonic.py
```

2.4 Changelog

2.4.1 0.2.0 (2016-08-08)

- Introduced different communication protocols (XML RPC and JSON RPC)

2.4.2 0.1.2 (2016-07-13)

- Fixed broken setup.py.

2.4.3 0.1.1 (2016-07-13)

- Added documentation.

2.4.4 0.1.0 (2016-07-13)

- Initial release to pypi.

CHAPTER 3

Feedback

If you have any suggestions or questions about **gearthonic** feel free to email me at mumpitz@wumpitz.de.

If you encounter any errors or problems with **gearthonic**, please let me know! Open an Issue at the Gitlab http://gitlab.com/wumpitz/gearthonic main repository.

g

A

C

D

E

F

G

I

J

L