

Remote interface protocol

Introduction

Currently, Phenix uses bespoke interfaces to Coot and PyMol that are substantially different in their Phenix-facing APIs, leading to a lot of code duplication. Furthermore, the Coot (and PyMol? I haven't personally looked) interface is uni-directional: Phenix sends a command and receives an acknowledgement, but does not receive any detailed information about the command's effects. This limits the usability of the interface to quite simple tasks.

I would like to add ChimeraX/ISOLDE as an optional viewer for Phenix, and additionally to access various functions of Phenix directly from ISOLDE. Since various practical barriers make direct access through Python difficult, the next best option would seem to be a well-designed bi-directional socket interface. This represents my initial draft of a protocol and suggested implementation.

Protocol

Communication between client and server should be as flexible and platform-agnostic as possible, avoiding the use of non-standard Python libraries as much as possible. The simplest (and arguably most flexible) choice is via JSON strings: the client sends a command as a dict dumped to JSON:

```
send_dict['cmd'] = 'command_name'
send_dict['args'] = [arg1, arg2, arg3, ... argn]
send_dict['kwargs'] = { 'kwarg1': kwarg1,
                        'kwarg2': kwarg2,
                        ...
                        'kwargn': kwargn
}
```

```
send_request_to_server(json.dumps(send_dict))
```

... where args and kwargs are any Python types that can be serialised to JSON and reconstituted on the server side (i.e. string, int, float, bool, None, or lists/dicts thereof).

In response, the server returns a dict detailing the result(s) of the command. The return dict should have some required keys (e.g. a unique identifier for the model in response to a `load_model()` command), but allows flexibility to return extra server-specific information (e.g. ISOLDE would always return a 'log' entry containing any messages written to the ChimeraX log in the course of running the command).

Addition of new methods on the server side should **not** require any addition of new code to the client implementation. Rather, the client should be a fairly generic class which queries the server for available methods and auto-populates with client-side methods accordingly. The corollary to this is that the server should provide a JSON-formatted description of all available methods on request (I suggest serving this in response to a HTTP GET, while commands are handled by POST).

For true bi-directional communication both Phenix and the external package need to act as both "client" and "server". To my knowledge this is not possible on a single port using built-in Python libraries, but it can be done with the combination of `asyncio` and `websockets` library. This approach

has the disadvantage that an extra level would need to be added to the protocol to distinguish an incoming command from an incoming response to a previously-sent command. In my opinion it would be simpler in cases where full bi-directional interaction is needed to just use two ports, one for each direction of communication. This way the package which first connects as client can start its own server and send a command containing the associated port number to establish the reverse connection.

Implementation

My preliminary implementation for ChimeraX/ISOLDE is attached. While the server side uses some code specific to ChimeraX, the client side is generic and should run in any Python environment. Key points:

- Addition of new server methods should not require the developer to know nor care about the details of the server implementation – they should just have to write a method meeting the requirements for JSON-serialisable argument and return types, and call `server.register_server_method()` to make it available. The server should handle the rest – in particular, ensuring the call is thread-safe since you will almost certainly want to be running the server in its own Python thread. For this, you should define a function wrapper called `thread_safe()`. My ChimeraX-specific implementation is below – this will need to be adapted slightly for each specific server environment:

```
def thread_safe(func):
    import functools
    @functools.wraps(func)
    def thread_safe_func(session, *args, **kwargs):
        from queue import Queue
        q = Queue()
        def inner_func(session, *args, q=q, **kwargs):
            from chimeraX.core.logger import StringPlainTextLog
            with StringPlainTextLog(session.logger) as rest_log:
                ret = {}
                try:
                    ret.update(func(session, *args, **kwargs))
                except Exception as e:
                    import traceback
                    ret.update({
                        'error': str(e),
                        'traceback': traceback.format_exc()
                    })
                ret['log'] = rest_log.getvalue()
            q.put(ret)
        session.ui.thread_safe(inner_func, session, *args, **kwargs)
        return q.get()
    return thread_safe_func
```

- Here, the `session` argument refers to the main ChimeraX session object, and is excluded from the final method signature passed to the client. The `@functools.wraps(func)` decorator is extremely important, since it ensures that the wrapped function looks like the original one to the Python interpreter allowing introspection to faithfully recapitulate it on the client side.

To provide a concrete example: if, on the server side I define the following function:

```
def test_command(session, arg1:'whatever', arg2:'you', kwarg1:'like'=None):
    """
    Will simply echo back a dict of the provided arguments.
    """
```

```
return {'arg1': arg1, 'arg2':arg2, 'kwarg1': kwarg1}
```

... and registered with the server under the name “test” with `register_server_method(“test”, thread_safe(test_command))`, then the next HTTP GET call from the client (called automatically when the client first connects via its `get_available_methods()` method, and at will thereafter) causes `list_server_methods()` to be called. This uses Python’s inspect introspection tools to extract the method docstring and signature including hints and default arguments and excluding an optional list of black-listed argument names (such as `self` and `session`) to a JSON dict. The client side then uses this information to create a method on-the-fly with the pruned signature and docstring, but which internally simply sends the arguments on to the server, then waits for and returns the result. This new function is added as a class attribute, so it can be called like any other method:

```
client = IsoldeRestClient(address, port)
client.connect()
client.test(1, 3.1415, ['watermelon', 'elephant'])
> {'arg1': 1, 'arg2': 3.1415, 'kwarg1': ['watermelon', 'elephant'], 'log': ''}
```

For real-world applications it would be advisable to run the client in its own thread to avoid hanging your main function while waiting for results from the server.

There is at least one potential “gotcha” with this design. Since methods retrieved from the server become properties of the client **class**, then methods added to one instance will become methods of **every** instance of the class. To avoid this, each application-specific client should be a trivial subclass of the client base class, i.e.:

```
class RESTClientBase:
    ... actual client class code goes here

class IsoldeRESTClient(RESTClientBase):
    pass

class CootRESTClient(RESTClientBase):
    pass

class PyMolRESTClient(RESTClientBase):
    pass
```