Integrating crystallographic structure determination and visualization:
remote control of Coot

Nathaniel Echols and Paul D. Adams
Lawrence Berkeley National Laboratory, Berkeley CA
correspondence: nathaniel.echols@gmail.com

Macromolecular structure building and refinement is typically an inherently visual task, requiring at the very least interactive validation of the final model, and ideally some degree of monitoring throughout the process. Most building and refinement programs, however, have traditionally been largely decoupled from direct visualization aside from the output of files suitable for 3D display. Although a few projects have dedicated molecular graphics viewers (Langer et al. 2013, Turk 2013), the overhead for both the programmer (who must develop and maintain the code) and the user (who must learn a separate interface) is substantial. An alternative, simpler approach is to leverage an existing viewer (see for instance Payne et al. 2005).

The ubiquitous crystallography model-building program Coot (Emsley et al. 2010) has several advantages which make it an obvious and attractive frontend for automated structure determination software:

- distribution under an open-source license (GPL)
- binary distributions for the most commonly used operating systems
- Python and Scheme scripting support

The latter feature facilitates the development of plug-ins which extend the functionality of Coot, allowing manipulation and augmentation of the graphical interface using either the PyGTK or GUILE scripting APIs, which wrap the GTK GUI library (http://www.gtk.org). Python (http://www.python.org) in particular provides a rich set of built-in functions, including support for commonly used network protocols such as HTTP. Importantly, the use of scripted plugins does not require modification of the source code or separate distribution of Coot itself.

The Phenix software (Adams et al. 2010) is a Python-based crystallography suite encompassing most common procedures in structure determination (with the exception of processing of raw diffraction images). We have placed an emphasis on ease of use and automation, including the development of a modern graphical interface (Echols et al. 2012). Although we have found it convenient to use a simple Python-based OpenGL molecular viewer within the Phenix GUI, the development of a full-fledged building program is far outside the scope of the project, and the existence of Coot has made this both unnecessary and undesirable. Inspired by an existing PyMOL feature[1] and the molecular dynamics viewer VMD (Humphrey et al. 1996), we therefore

[1] G. Landrum, unpublished; http://sourceforge.net/p/pymol/code/HEAD/tree/trunk/pymol/modules/pymol/rpc.py. Note that this module also now registers most of the function calls automatically by extracting their names from the cmd module.

implemented an extension to Coot which enables direct control from external programs using the XML-RPC protocol (http://xmlrpc.scripting.com/spec.html). Although the use of TCP/IP sockets raises several practical issues, the protocol is sufficiently flexible to make fully interactive communication possible on all three common desktop operating systems (Mac, Linux, and Windows), and is easily adapted to any program capable of running an XML-RPC client, including software written in Python, C++, and Java.

Overview of XML-RPC

The XML-RPC (eXtensible Markup Language Remote Procedure Call) protocol is a simple client-server framework for executing remote code in a platform- and language-independent manner. Other similar frameworks are available, the most popular being SOAP (Simple Object Access Protocol), but we chose XML-RPC primarily because it is installed by default with Python, and because the more advanced features of SOAP were deemed unnecessary for this application. The underlying communication is handled via HTTP, with the server handling POST queries containing formatted XML (example below), which encapsulates the method calls:

```
<?xml version='1.0'?>
<methodCall>
<methodName>read_pdb</methodName>
<params>
<param>
<value><string>/Users/nat/data/lysozyme.pdb</string></value>
</param>
</params>
</methodCall>
```

The use of XML imposes some obvious limitations on the type of information which may be exchanged. Any binary content or other special characters which are unacceptable to the XML parser are prohibited; this includes Python pickles (serialized objects). In practice, if a common filesystem is available it is usually possible to use intermediate files (PDB or MTZ), with the XML-RPC calls restricted to instructions to load files.

Definition of the remotely callable methods is left to the application developer; individual methods must be registered with the XML-RPC server. Typically the method names in the application will be identical to the remote calls, although this is not a requirement. Arguments are limited to objects which may be embedded in XML, which excludes binary data. A minimal example of paired client and server Python code is shown below.

Server:

```
from iotbx.pdb import hierarchy
import SimpleXMLRPCServer

def read_pdb (file_name) :
```

```
    pdb = hierarchy.input(file_name=file_name)
    pdb.hierarchy.show()

server = SimpleXMLRPCServer.SimpleXMLRPCServer(
    addr=("localhost", 40000))
server.register_function(read_pdb, "read_pdb")
server.serve_forever()
```

Client:

```
import xmlrpclib
import sys
remote = xmlrpclib.ServerProxy("http://localhost:40000/RPC2")
remote.read_pdb(sys.argv[1])
```

For interactive applications, it is necessary to handle the remote requests without blocking the main execution thread. This can be handled in several ways depending on the design of the program being extended. If the Python interpreter itself is executed, as opposed to being embedded as a library in a monolithic C or C++ executable, the threading module may be used to handle server requests separately from the main thread. However, the programmer must still exercise caution when calling methods which may not be thread-safe, for example when updating a GUI[2]. Toolkits such as GTK and wxWidgets (or in the Phenix case, PyGTK and wxPython) allow use of timers to execute function calls at regular intervals. Therefore, instead of calling `server.serve_forever()`, one can instead call `server.handle_request()` periodically with a timer.

Application to Coot

Because Coot has the ability to execute arbitrary Python code, with a large number of functions available for loading and manipulating molecule and map objects, application of XML-RPC is relatively straightforward. However, we found it preferable to avoid the requirement of registering each of the many functions in the Coot API individually, especially since this would not automatically track with changes in Coot. A convenient solution is to simply use getattr() to retrieve methods by name from the coot module. Additionally, by specifying a class containing additional custom methods, these too can be automatically made accessible via XML-RPC.

Although Coot will not run a continuous, separate, Python thread which only listens on a socket, objects which are not garbage-collected will remain persistent in memory. Therefore, we can use the timeout function from the gobject module (part of PyGTK) to handle incoming requests.

[2] The design of PyMOL actually allows the child thread used to handle server requests to run any API function without crashing the program, but this requires additional logic coded in C, and would not normally apply to programs using Python GUI toolkits. We note in passing that the same principles described in this article may apply to other Python-enabled molecular graphics such as UCSF Chimera or VMD. However, the problem of how to handle XML-RPC requests more or less continuously without crashing the GUI must be handled differently in each case (and a preliminary attempt to write a Chimera plugin was unsuccessful for this reason).

We have found 250ms to be a reasonable interval, as it is short enough to make the latency barely noticeable during interactive control of Coot from the Phenix GUI, but does not appear to affect the performance of Coot itself. (A timeout of 10ms, on the other hand, makes Coot nearly unusable.)

The core of the implementation, shown below, subclasses the `SimpleXMLRPCServer` class, overriding the `_dispatch` method in order to extract the named methods. Failure to locate the desired function raises an exception. If an exception is caught for any other reason, it is re-raised with the original traceback embedded in the message string. This permits the client code to present full debugging information to the user and/or developer. Otherwise, the return value of `_dispatch` will be received by the client.

```
import coot
import gobject
import SimpleXMLRPCServer
import sys

class coot_server(SimpleXMLRPCServer.SimpleXMLRPCServer) :
  def __init__ (self, interface, **kwds) :
    self._interface = interface
    SimpleXMLRPCServer.SimpleXMLRPCServer.__init__(self, **kwds)

  def _dispatch (self, method, params) :
    if not self._interface.flag_enable_xmlrpc :
      return -1
    result = -1
    func = None
    if hasattr(self._interface, method) :
      func = getattr(self._interface, method)
    elif hasattr(coot, method) :
      func = getattr(coot, method)
    if not hasattr(func, "__call__") :
      print "%s is not a callable object!" % method
    else :
      try :
        result = func(*params)
      except Exception, e :
        traceback_str = "\n".join(traceback.format_tb(
           sys.exc_info()[2]))
        raise Exception("%s\nOriginal traceback:%s" % (str(e),
           traceback_str))
      else :
        if result is None :
          result = -1
    return result
```

(Note that in the quite common case that the return value is None, the server instead returns -1, due to limitations inherent to Python's implementation of the protocol.)

The second half of the implementation actually starts the XML-RPC server, and defines additional methods which will automatically become part of the remotely callable API:

```
class coot_interface (object) :
  def __init__ (self, port=40000) :
    self.flag_enable_xmlrpc = True
    self.xmlrpc_server = coot_server(
      interface=self,
      addr=("127.0.0.1", port))
    self.xmlrpc_server.socket.settimeout(0.01)
    self.current_imol = None
    print "xml-rpc server running on port %d" % port
    gobject.timeout_add(250, self.timeout_func)

  def timeout_func (self, *args) :
    if (self.xmlrpc_server is not None) :
      self.xmlrpc_server.handle_request()
    return True

  def update_model (self, pdb_file) :
    if (self.current_imol is None) :
      self.current_imol = read_pdb(pdb_file)
    else :
      clear_and_update_molecule_from_file(self.current_imol,
        pdb_file)

  def recenter_and_zoom (self, x, y, z) :
    set_rotation_centre(x, y, z)
    set_zoom(30)
    graphics_draw()
```

In this example, the method `update_model` reloads a model as desired (updating the molecule object rather than creating a new one), for instance showing the progress of automated model-building or refinement. The `recenter_and_zoom` method is used in a number of contexts in the Phenix GUI, in particular the MolProbity interface (Chen et al. 2010), to associate tables and plots of residue properties with the Coot viewport.

A more complex, fully functional implementation of this method is distributed as part of CCTBX in the `cootbx` directory, with the file name `xmlrpc_server.py`. It may be invoked at launch time as:

```
coot --script xmlrpc_server.py
```

We have found it most convenient for the user if the remote control can be disabled and resumed at any time; therefore the actual implementation also adds a button to the Coot toolbar which toggles the connection.

Once Coot is started with the XML-RPC server running, other Python code can connect using xmlrpclib.ServerProxy and call the full range of API functions (built-in or custom), e.g.:

```
import xmlrpclib
coot = xmlrpclib.ServerProxy("http://127.0.0.1:40000/RPC2")
coot.read_pdb("model.pdb")
coot.auto_read_make_and_draw_maps("maps.mtz")
```

Other programming languages will work equally well provided an XML-RPC library is available.

Practical considerations, limitations, and workarounds

Although the above description is reasonably simple and foolproof, a number of problems became immediately apparent upon testing in real-world situations. Chief among these was the requirement that Coot be in a running state before remote calls can be initiated. Using the XML-RPC protocol exactly as intended meant that buttons intended to open the results of a program run would either cause the Phenix GUI to block for several seconds while Coot was launched, or would not actually perform the desired action until clicked a second time. As neither behavior was ideal for an intuitive user interface, it was necessary to find a way to deal with the lag time transparently.

Our eventual solution leaves much to be desired from a purist perspective, as it essentially reduces the XML-RPC communication to a one-way system. The client code is invoked as soon as Coot is started, but well before the XML-RPC server is started; when the remote calls inevitably fail due to connection refusal, the resulting exception is suppressed and the request cached by the modified client. A timer function in the Phenix GUI is used to attempt to flush the cache (again, every 250ms) until successful. In practice this appears nearly seamless to the end user (aside from the delay associated with launching Coot), but as there is no guarantee that any given query will be immediately successful, the return value from the server cannot be reliably obtained.

Several issues were only partially resolved. On many systems, especially running Linux, the use of sockets was (surprisingly) unreliable despite connecting only on the local host. This led to a number of connection errors even when Coot was running, which had to be specifically ignored. Socket errors also occur when the desired TCP/IP port is unavailable, either due to use by another program, or a previous instance of Coot. Therefore our module also chooses the port number from a large range at random, and communicates this to Coot via an environment variable.

Filesystem-based alternatives

Although we believe that the flexibility of the XML-RPC approach makes it a compelling choice for inter-program communication, it is not necessarily suitable for applications where Coot must accept input from a remote host - for instance when the external software is being run on a managed cluster. In such situations, or any other application where the use of sockets is either undesirable or unnecessary, a simpler method based on monitoring of file modification times can be equally effective. Again, the timeout function is key; a minimal example is shown below:

```
import os.path

class watch_model (object) :
  def __init__ (self, file_name, timeout=2000,model_imol=None) :
    self.file_name = file_name
    self.model_imol = model_imol
    self.last_mtime = 0
    import gobject
    gobject.timeout_add(timeout, self.check_file)

  def check_file (self) :
    import coot
    if os.path.exists(self.file_name) :
      file_mtime = os.path.getmtime(self.file_name)
      if file_mtime > self.last_mtime :
        if self.model_imol is not None :
          clear_and_update_molecule_from_file(self.model_imol,
            self.file_name)
        else :
          self.model_imol = read_pdb(self.file_name)
        self.last_mtime = file_mtime
```

To use in the context of a program which regularly updates a PDB file, the calling code must simply write a script invoking the watch_model class with the appropriate file name. When run in Coot, the script will then reload the PDB file whenever the modification time changes (with a minimum frequency of every 2 seconds). As above, care must be taken when operating on NFS mounts. An example of this approach is also available in the `cootbx` directory as `watch_file.py`, including a more complex class which also reads a file containing map coefficients with standard labels. The applicability is not limited to PDB (or map) files; the target file could just as easily be a dynamically generated Python (or Scheme) script instead. Note that because this method does not require the use of any APIs in the underlying program beyond the ability to write a script file, it is suitable for use in software based on FORTRAN or other environments where the XML-RPC protocol is unavailable or unwieldy.

Availability

With the exception of features which are specific to the Phenix GUI, the majority of the implementations detailed above are part of the CCTBX, and we encourage their use and redistribution without restrictions as long as the original copyright is included (see http://sourceforge.net/p/cctbx/code/HEAD/tree/trunk/cctbx/LICENSE_2_0.txt for full license terms).

Acknowledgments

We are grateful to Joel Bard for suggesting the use of the timeout function, Paul Emsley and Bernhard Lokhamp for technical advice, and William Scott and Charles Ballard for debugging issues with Coot binary distributions. Funding was provided by the NIH (grant # GM063210) and the Phenix Industrial Consortium.

References

Adams, P. D., Afonine, P. V., Bunkóczi, G., Chen, V. B., Davis, I. W., Echols, N., Headd, J. J., Hung, L.-W., Kapral, G. J., Grosse-Kunstleve, R. W., et al. (2010). Acta Crystallographica. Section D, Biological Crystallography. 66, 1–9.

Echols, N., Grosse-Kunstleve, R. W., Afonine, P. V., Bunkóczi, G., Chen, V. B., Headd, J. J., McCoy, A. J., Moriarty, N. W., Read, R. J., Richardson, D. C., et al. (2012). J. Appl. Cryst (2012). 45, 581-586.

Emsley, P., Lohkamp, B., Scott, W. G., & Cowtan, K. (2010). Acta Crystallographica D. 66, 486–501.

Humphrey, W., Dalke, A., & Schulten, K. (1996). J. Molec. Graphics 14, 33-38.

Langer, G. G., Hazledine, S., Wiegels, T., Carolan, C., & Lamzin, V. S. (2013). Acta Crystallographica D. 69, 635–641.

Payne, B. R., Owen, G. S. & Weber, I. (2005). Computational Science - Iccs 2005, Pt 1, Proceedings 3514, 451-459.

Turk, D. (2013). Acta Crystallographica D. 69, 1–16.