SCOP Manual Annex — Python Binding

Version 1.4, 12th November 2003
David Ingram (dmi1000@cam.ac.uk)

Contents

1 Introduction

This annex to the SCOP manual describes the Python language binding. The binding implements the complete SCOP API, and is itself written in Python. The API is very similar to the C++ one, but more concise because it is simplified by Python's type system and garbage collector.

You should read at least the *Programming model* section of the main manual before this annex — that information applies to all language bindings and is not repeated here. The main manual is also essential reading for discovering the precise semantics of each command; here they are documented by example only.

1.1 Using the library

The library consists of three files in the `python/` directory within the main distribution: `scop.py`, `scoplib.py` and `scopxml.py`. You can install it by copying these to an appropriate location for Python source on your system. Alternatively, just set your `PYTHONPATH` environment variable to wherever you have unpacked SCOP, for example:

```
export PYTHONPATH=$PYTHONPATH:$HOME/scop/python
```

To use the library from your Python programs, you only need to reference the SCOP module, as follows:

```
import scop
```

This provides access to all the functions, including the level 2 XML API, apart from the internal debugging routines.

1.2 Example programs

Python versions of the example programs are listed in this manual, and can also be found in the directory `python/examples`. They behave in an identical manner to their C++ and Java counterparts. You can of course talk to the C++/Java versions of the servers with the Python clients, and vice-versa.

There is one additional example, `status.py`, which simply shows how to exercise the `list` function (the `scop` utility itself serves this purpose for the C++ binding).

The `multiplex` wrapper for C's `select()` function is not needed, because Python's `select()` native interface is much cleaner than the C one.

2 General semantics

There is no need to free resources because buffers are dealt with by the garbage collector. The connection itself must be closed with the `scop_close()` method, however.

Python exceptions are not thrown by the library (at least, not intentionally) — errors are generally indicated by returning None.

2.1 Argument types

The following arguments are all of type *string*: `remote_hostname`, `name`, `endpoint`, `interest`, `message`, `method`, `text`, `request`, `reply`.

`log_level` is an integer. `unique` and `verify` are booleans represented by integers.

3 Level 1 API

As with the other language bindings, the level 1 API provides a single data type, namely strings.

The socket file descriptor used in the C++ binding is replaced by a Python socket object (although in fact you don't need to know what type it is since the `sock` object returned by `scop_open()` is simply passed back to the library so the other routines can identify which connection to use).

3.1 Connection establishment and teardown

`scop_open(remote_hostname, name, unique = 0)` - *Returns socket object or None*
`scop_close(sock)` - *No return value*
`scop_listen(sock, interest, unique = 0)` - *No return value*

3.2 Message functions

`scop_send_message(sock, endpoint, message, verify = 0)`
- *Returns an integer (-1 if not verifying)*
`scop_get_message(sock)`
- *Returns a tuple: (message string, rpc_flag boolean), or None on error*

Note that `scop_get_message()` returns a tuple of which the actual message is the first component (in the C++ API, `rpc_flag` is passed in as a pointer-to-int instead, and in Java there is a separate `reply_required()` method). This also applies to the Level 2 `scop_get_struct()` described below. Even if you don't intend to use the value of `rpc_flag` don't forget to assign the result to a tuple; not doing so is a common cause of error.

Message passing example

Right, that's enough calls for us to write our first example:

```
# sender.py - DMI - 7-11-03
# Usage: sender [ <message> ]   (default message is "Hello world!")

import scop, sys

if len(sys.argv) > 1:
   msg = sys.argv[1]
else:
   msg = "Hello world!"
sock = scop.scop_open("localhost", "sender")
if sock == None:
   print "Error on open"
   sys.exit()
scop.scop_send_message(sock, "receiver", msg)
scop.scop_close(sock)
```

```
# receiver.py - DMI - 29-10-03

import scop, sys

sock = scop.scop_open("localhost", "receiver")
if sock == None:
   print "Error on open"
   sys.exit()
while 1:
   msg, rpc_flag = scop.scop_get_message(sock)
   print "Received <" + msg + ">"
   if msg == "quit":
      break
scop.scop_close(sock)
```

Since this was the first example, I was careful to check that `scop_open()` didn't return an error. To keep things concise though, error returns from SCOP functions are not checked in the rest of the examples, except for `sos.py`.

3.3 Predefined event sources

`scop_set_source_hint(sock, endpoint)` - *No return value* `scop_emit(sock, message, verify = 0)` - *Returns an integer (-1 if not verifying)*

Event sources example

```
# event_source.py - DMI - 7-11-03
# Usage: event_source [ <source> ]   (default source is "news")

import scop, sys, time

count = 1
sock = scop.scop_open("localhost", "event_source")
if len(sys.argv) > 1:
   source = sys.argv[1]
else:
   source = "news"
scop.scop_set_source_hint(sock, source)
while 1:
   msg = "Item " + str(count)
   scop.scop_emit(sock, msg)
   count = count + 1
   time.sleep(1)
```

```
# event_listener.py - DMI - 7-11-03

import scop, sys

sock = scop.scop_open("localhost", "event_listener")
scop.scop_listen(sock, "news")
while 1:
   msg, rpc_flag = scop.scop_get_message(sock)
   print "Received <" + msg + ">"
```

```
# multi_listener.cpp - DMI - 7-11-03
# Usage: multi_listener [ <source-one> <source-two> ]
#    (default sources are "news" and "updates")
```

```
import scop, sys, select

sock = []
for i in range(2):
   sock.append(scop.scop_open("localhost", "multi_listener"))

if len(sys.argv) == 3:
   scop.scop_listen(sock[0], argv[1])
   scop.scop_listen(sock[1], argv[2])
else:
   scop.scop_listen(sock[0], "news")
   scop.scop_listen(sock[1], "updates")

while 1:
   read_fds = [sock[0], sock[1]]
   r, w, e = select.select(read_fds, [], [])
   for fd in r:
      msg, rpc_flag = scop.scop_get_message(fd)
      print "Received <" + msg + "> from ",
      if fd == sock[1]:
         print "updates"
      else:
         print "news"
```

3.4 RPC functions

`scop_rpc(sock, endpoint, request, method = None)`
- *Returns a string, or None on error*
`scop_get_message(sock)`
- *Returns a tuple: (message string, rpc_flag boolean), or None on error*
`scop_send_reply(sock, reply)` - *No return value*

The same `scop_get_message()` function is used to get RPC requests and plain messages. If an RPC is expected you should of course check that the second member of the tuple, `rpc_flag`, returns true.

RPC example

```
# client.py - DMI - 8-11-03
# Usage: client [ <query> ]   (default query is "Hello world!")

import scop, sys

if len(sys.argv) > 1:
   query = sys.argv[1]
else:
   query = "Hello world!"
sock = scop.scop_open("localhost", "client")
reply = scop.scop_rpc(sock, "server", query)
print "Query <" + query + ">, Reply <" + reply + ">"
scop.scop_close(sock)
```

```
# server.py - DMI - 8-11-03

import scop

sock = scop.scop_open("localhost", "server")
while 1:
   query, rpc_flag = scop.scop_get_message(sock)
   length = len(query)
   reply = ""
   for i in range(length):
      reply = reply + query[i] + query[i]
   scop.scop_send_reply(sock, reply)
scop.scop_close(sock)
```

3.5 Admin functions

`scop_query(sock, endpoint)` - *Returns an integer*
`scop_clear(sock, endpoint)` - *No return value*
`scop_set_log(sock, log_level)` - *No return value*
`scop_terminate(sock)` - *No return value*
`scop_reconfigure(sock)` - *No return value*
`scop_list(sock)` - *Returns a list of (name, interest, src_hint) tuples*

Status example

This illustrates the use of the `scop_list()` call.

```
# status.py - DMI - 8-11-03

import scop

sock = scop.scop_open("localhost", "status")
v = scop.scop_list(sock)
print len(v), "clients connected."
for tuple in v:
   name, interest, src_hint = tuple
   print "Client connection <" + name + "> listening to <" + interest + \
      ">, source hint <" + src_hint + ">"
scop.scop_close(sock)
```

3.6 Cookies

`scop_set_plain_cookie(sock, text)` - *No return value* `scop_get_plain_cookie(sock, name)` - *Returns a string*

Error-checking example

Here's one final example, this time with proper error checking. It's a server which simply logs all the messages it receives using `syslog`.

```
# sos.py - DMI - 8-11-03

import scop, sys, syslog, os

sock = scop.scop_open("localhost", "sos", 1)
if sock == None:
   print "Can't connect to scopserver."
   sys.exit()

if os.fork() > 0:
   sys.exit() # Detach
```

```
while 1:
   buf, rpc_flag = scop.scop_get_message(sock)
   if buf == None:
      syslog.syslog(syslog.LOG_INFO, "Lost connection to scopserver.")
      sys.exit()
   syslog.syslog(syslog.LOG_INFO, buf)
scop.scop_close(sock)
```

---

4 API Level 2 - Types with XML

The Level 2 API adds support for non-string data types, including structured data.

4.1 General semantics

The Python Level 2 API is completely different from the C++ and Java versions, because it does not use any `pack()` or `extract()` functions and there is no `vertex` data type. This is possible because Python provides a list data type and the means to determine the type of objects at runtime, which yields a radically simpler interface.

All you need to do is assemble your own data type using the Python facilities for constructing lists and including any combination of integers, floats and strings; then pass this object to SCOP. The library will marshall it for you. Likewise, when receiving data it is extracted from the XML automatically and handed to you as the appropriate native Python object.

There is no explicit support for SCOP's binary types, but fortunately this isn't necessary since Python strings can include NULLS. The solution adopted by the library to handle strings is to check if it has been passed entirely printable characters. If so it is packed as an XML string, otherwise it is sent as binary (using the SCOP protocol's hex encoding). The Python binding can also receive binary types sent by programs written in other languages, but these are always returned as Python strings.

4.2 Argument types

The following parameters are all *polymorphic*: `data`, `args`, `request`, `reply`, `v`. In the restricted sense of SCOP this means they may either be integers, floats, strings, or lists (containing the same types). Some of the function return values are also polymorphic in the same sense.

Note that parameters named `request` and `reply` are encoded without using XML if they happen to be strings (see the level 1 API).

4.3 XML Message functions

`scop_send_struct(sock, endpoint, args, method = None)` - *No return value*
`scop_get_struct(sock)` - *Returns a tuple: (polymorphic, rpc_flag boolean)*

XML example

This address book example is a lot simpler than the same thing in C++, because we can use Python's dictionary type for data storage. SCOP doesn't support dictionaries directly though, so we need marshalling functions to convert it to something it does understand (lists of lists, in this case). In fact this is so useful I might add dictionary support to the library at some point, although then this example would be rather pointless!

```
# xml_sender.py - DMI - 11-11-03

import scop

def marshall(dict):
   l = []
   keys = dict.keys()
   for k in keys:
      l.append([k, dict[k]])
   return l

ab = { "Poirot": "Belgium",
       "Morse": "Oxford, UK",
       "Danger Mouse": "London, UK" }
v = marshall(ab)
sock = scop.scop_open("localhost", "xml_sender")
scop.scop_send_struct(sock, "xml_receiver", v)
scop.scop_close(sock)
```

```
# xml_reciever.py - DMI - 11-11-03
# Usage: xml_reciever [-inspect]

import scop, sys
from scopxml import pretty_print

def extract(v):
   d = {}
   for pair in v:
```

```
      key, value = pair
      d[key] = value
   return d

sock = scop.scop_open("localhost", "xml_receiver")
v, rpc_flag = scop.scop_get_struct(sock)
if len(sys.argv) == 2 and sys.argv[1] == "-inspect":
   s = pretty_print(v)
   print s
ab = extract(v)
print ab
scop.scop_close(sock)
```

4.4 XML RPC functions

`scop_rpc(sock, endpoint, request, method = None)`
- *Returns polymorphic, or None on error*
`scop_send_reply(sock, reply)` - *No return value*
`scop_get_request(sock)` - *Returns polymorphic or None if not an incoming RPC*

Notice that the same `scop_rpc()` and `scop_send_reply()` functions are used as for non-XML RPC's. These functions perform different operations based on the types of their `request` and `reply` arguments. If these are plain strings then XML is not used, otherwise it is.

A subtle consequence of this is that it is not possible to send an XML message containing a single value of type string using the Python binding. Fortunately there is no particularly good reason to do this.

XML RPC example

```
# xml_client.py - DMI - 11-11-03
# Usage: xml_client [<n> <k>]    (default values n = 4, k = 2)

import scop, sys

if len(sys.argv) != 3:
   n, k = 4, 2
else:
   n, k = int(sys.argv[1]), int(sys.argv[2])
```

```
sock = scop.scop_open("localhost", "xml_client")
v = [n, k]
w = scop.scop_rpc(sock, "xml_server", v)
print str(n) + " choose " + str(k) + " equals " + str(w) + "."
scop.scop_close(sock)
```

```
# xml_server.cpp - DMI - 11-11-03

import scop

def combi(n, k):
   result = 1;
   if k > n or k < 0:
      return 0
   for i in range(k):
      result *= n - i
   for i in range (1, k + 1):
      result /= i
   return result

sock = scop.scop_open("localhost", "xml_server")
while 1:
   v = scop.scop_get_request(sock)
   w = combi(v[0], v[1])
   scop.scop_send_reply(sock, w)
scop.scop_close(sock)
```

4.5 XML RPC method names

There are no `extract_method()` and `extract_args()` functions in the Python binding, since these can be achieved simply by assigning the returned object to a tuple.

Multiple RPC methods example

```
# method_client.py - DMI - 11-11-03

import scop
```

```
def cent_to_faren(sock, c):
   return scop.scop_rpc(sock, "method_server", c, "ctof")

def faren_to_cent(sock, f):
   return scop.scop_rpc(sock, "method_server", f, "ftoc")

def count_uses(sock):
   return scop.scop_rpc(sock, "method_server", None, "stats")

sock = scop.scop_open("localhost", "method_client")
print str(0.0) + " deg C = " + str(cent_to_faren(sock, 0.0)) + " deg F."
print str(20.0) + " deg C = " + str(cent_to_faren(sock, 20.0)) + " deg F."
print str(60.0) + " deg F = " + str(faren_to_cent(sock, 60.0)) + " deg C."
print "The server has been accessed " + str(count_uses(sock)) + " times."
scop.scop_close(sock)
```

```
# method_server.py - DMI - 11-11-03

import scop, sys

invocations = 0

def cent_to_faren(c):
   global invocations
   invocations += 1
   return (9.0 * c / 5.0) + 32.0

def faren_to_cent(f):
   global invocations
   invocations += 1
   return (f - 32.0) * 5.0 / 9.0

sock = scop.scop_open("localhost", "method_server")
while 1:
   v = scop.scop_get_request(sock);
   method, args = v
   if method == "ctof":
      w = cent_to_faren(args)
   elif method == "ftoc":
      w = faren_to_cent(args)
   elif method == "stats":
      w = invocations
   else:
      sys.exit()
   scop.scop_send_reply(sock, w)
scop.scop_close(sock)
```

4.6 XML Cookies

`scop_get_cookie(sock, name)` - *Returns polymorphic or None*
`scop_set_cookie(sock, data)` - *No return value*

4.7 Debugging

These internal functions are defined in the module `scopxml`, and may occasionally be helpful if you want to access the XML parser directly for some reason.

`vertex_to_string(v, method = None)` - *Returns a string*
`string_to_vertex(s)` - *Returns polymorphic*
`pretty_print(v)` - *Returns a string*

A Performance

The local-case performance was measured on an 800 Mhz Pentium III as 890 RPC's per second. This is slightly slower than the Java binding (1000 round trips per second) and 7.5 times slower than C++ (6700 per second). Of course the `scopserver` process is implemented in C++ either way.

It's interesting to note the effect on code size using Python: in C++, the main library is 782 lines and the XML conversion routines are 963 lines. In Python, the library is only 332 lines and the XML routines are 144 lines.