SCOP Programmer's Manual

Version 1.5, 11th September 2006
David Ingram (dmi1000@cam.ac.uk)

Contents

1 API Structure

This programmer's manual describes the SCOP API (for a general introduction to SCOP and what it can do, please see the separate overview and installation guide document). The API is structured in several layers:

- **Level 0** - Protocol
- **Level 1** - Library calls
- **Level 2** - Types (XML)
- **Level 3** - Reflection
- **Level 4** - Shell commands

If you want to get started programming quickly, just look at the documentation for level 1. Level 0 describes the low-level protocol, and levels 2 & 3 are just further refinement. In this manual §4 (which describes API level 1) contains all the functions you need to write a program using SCOP.

Level 3 is a work-in-progress, intended to provide type checking and introspective capabilities. Currently it is not shipped with SCOP so you can ignore it (unless you are using the cookie feature).

Level 4 is the command line interface, which is described in the installation guide. It is useful for trying things out and for debugging.

Note: it's not a good idea to use a mixture of levels, unless you know exactly what you are doing.

2 Language bindings

Bindings currently exist for C++, Python and Java, with contributed support for C and Scheme.

In principle any language can be used to access SCOP via API levels 0 and 4 (because you can write your own library routines to implement as much of the protocol as you need, or invoke the system command interpreter to call the shell utilities from your program).

Support for levels 1 to 3 requires a language binding. These are easy to create because only the client-side library needs to be ported (the server and command line utilities can still be written in C++).

The main manual describes the C++ binding; the Java and Python syntax are covered in separate annexes. I have chosen to present it this way rather than to describe the API in pseudo-code followed by bindings for all languages. This is because (i) in that case *everyone* suffers the inconvenience of cross-referencing the API, and (ii) C++ is better defined than pseudo-code, and I suspect most programmers have a fair idea of what the basic syntax means.

2.1 C++

C++ is fully supported at levels 1 and 2.

The interface actually isn't particularly object-oriented; it would arguably be better to create an SCOP connection class and make the function calls methods of this class, to avoid passing a socket parameter to every call.

2.2 Java

The Java bindings fully implement levels 1 and 2. The syntax is described in the separate annex to this document.

2.3 Python

The Python binding also provides full functionality at levels 1 and 2, and specifics are described in another annex.

2.4 C

Plain C is not officially supported. However, SCOP makes only light use of C++'s additional features and Sven Hartrumpf has now created a C port of SCOP.

2.5 Scheme

Sven has also completed a Scheme language binding for SCOP, which is maintained separately (not included in the main package).

3 Programming model

3.1 Connections

You may open the library multiple times from within the same program, thereby holding several connections to the `scopserver`. This is sometimes useful for listening to multiple event sources. It is also encouraged for clarity and to avoid errors when you are doing complex mixtures of message passing, receiving and RPC's (even when they could in principle be done on the same connection).

Connections are returned to you from the library as standard UNIX file descriptors, each of which identifies an open TCP/IP socket. You can therefore use the operating system's ordinary `select()` call to handle multiple connections at once or test if events have arrived, therefore integrating SCOP communications cleanly with the rest of your program's I/O. SCOP does not require you to hand control over to a built-in event loop, so you can organise your program's flow of control any way you see fit.

3.2 Endpoints

Every connection to the server has a name, which you specify when you open it. Actually the underlying protocol doesn't demand that you set a name, but until you do so your connection is labelled `Unnamed`. You can also specify that you wish to listen to events sent to a more general category name in which you express "interest".

Connection *names* and *interests* are collectively called *endpoints*. Messages can be sent to any endpoint. Effectively all clients "listen" to their own name, regardless of whether they have mentioned an interest as well. If multiple clients have the same name or interest they will all receive a copy of any messages sent there.

If no-one is listening to an endpoint the default behaviour is to silently discard messages. This allows you to start up event sources and sinks in any order, and to restart servers without affecting clients, since `scopserver` will rebind endpoints to the same name without disturbing other connections. This is in contrast to traditional systems where clients must register with services which are already running, and need to be restarted if the server dies. `scopserver` *decouples* event sources and sinks.

You may have noticed that supporting both names and interests is almost redundant; multicast works to both types of endpoint and names are not used for anything other than to label endpoints for incoming messages. Indeed a previous version of SCOP only supported interests (and not names). The main problem with this is that when clients are listed using `scop list` they are all described according to the endpoint they are listening to, so it is difficult to tell them apart. It is therefore important to choose names which accurately describe the *client*, and interests which cover the kinds of *event source* you want to hear from.

The RPC commands do not allow multicast since their purpose is to contact a specific server by name.

3.3 Scenarios

Here are some typical communication scenarios, illustrating choices of endpoints:

	Message source name	Destination interest	Message target name[s]
1	Window Manager	– *not set* –	Panel dock
2	Weather station	Weather update	Alpha, Beta, Gamma (clients)
3	Beta	P2P file share	Alpha, Beta, Gamma (peers)
4	Alpha, Beta, Gamma	– *not set* –	Central log facility

The first example is a point-to-point message which is sent directly to the target's name (Panel dock), so no listening interest is necessary. The second example is a multicast to a set of different clients all of whom are listening to the "Weather update" endpoint. In the third case the sender also happens to be one of the clients. The final case demonstrates fan-in, with multiple clients of the central log facility sending messages to it.

In keeping with the layered design philosophy, SCOP does not provide *event filtering* services. You can easily build these on top of SCOP by writing your own event filtering server, which listens to streams of interest and outputs filtered versions.

4 API Level 1 - Library calls

4.1 General semantics

The level 1 API is simple because it only supports one datatype, namely null-terminated ASCII strings. It is up to you how you structure the information you send.

The following memory management strategy is consistently applied to all SCOP library functions. Buffers passed **to** the library must be *allocated and freed by you*. You may free them immediately after the call which uses them. Buffers returned to you **from** the library are *allocated by the library* and must be *freed by you* using the standard C++ `delete` keyword (which is why the library can't be called from plain C without some modification).

Endpoint names may include any printable character (including spaces) except for the exclamation mark.

4.2 Connection establishment and teardown

```
int scop_open(const char *remote_hostname, const char *name, int unique =
0);
void scop_listen(int sock, const char *interest, int unique = 0);
int close(int sock);
```

`scop_open` returns a file descriptor (referred to later as `sock`) on success, or -1 if there's an error (which usually means there's no `scopserver` process to connect to).

Specify "`localhost`" for `remote_hostname` if you aren't using a remote `scopserver`. UNIX domain sockets are not supported at present, only local-case TCP.

The `name` may be `NULL` if you wish to remain anonymous (you can't receive messages in this case). Note that it is encouraged to name endpoints even if you are only going to send messages, so that event sources can be listed with "`scop list`".

If you set the `unique` flag, any clients already connected with the same name are disconnected from `scopserver` by this call first. Leave `unique` set to zero if you wish to perform multicast transmissions to this endpoint.

Connections are terminated using the *normal* OS `close` routine for sockets (note the lack of `scop` prefix).

4.3 Message functions

```
int scop_send_message(int sock, const char *endpoint, const char
*message, int verify = 0);
char *scop_get_message(int sock);
```

Following the memory management conventions, it is safe to delete `message` as soon as `scop_send_message` returns.

If `verify` = `1` then `scop_send_message` returns the number of clients actually sent to, which may be zero, one or more. If `verify` = `0`, `scop_send_message` returns `0`. It is a good idea not to request verification unless you actually need it since this will reduce network traffic and speed up the call. If `scopserver` disconnects then `scop_send_message` returns `-1` regardless of the value of the `verify` flag.

`scop_get_message` blocks until a message arrives. You will never receive a partial fragment of a message. It either returns the message content or `NULL` if the connection has been closed for some reason. The library doesn't tell you who sent each message, so if this matters it is best to encode the name of the client in the message body.

Message passing example

At this point we have dealt with all the really important stuff, so we can present a fully working example. The rest of this document covers more advanced or specialised features.

Important note: all error-checking has been omitted from the examples, to keep them concise. In reality it would be extremely foolish not to check every return code from an `scop` function:

for example, `scop_open` will fail if the server isn't running, and `scop_get_message` will return `NULL` if the connection has been lost, which may happen at any time if somebody decides to kill the process, for instance. Failing to check for these conditions often causes unpleasant crashes, such as busy waiting in an infinite loop and filling up the disk with error log output...

All of the examples in this manual are also provided in the `examples/` subdirectory of the SCOP distribution. To compile them, first build and install SCOP from the toplevel directory in the usual way, as described in the installation guide. Then change into the `examples/` subdirectory. Look at the `Makefile` there; if you didn't install SCOP as root you should change the definitions near the top of it, as indicated by the comments. Then type `make` to compile the examples.

The first example consists of two programs, `sender` and `receiver`. They should be self-explanatory. You can of course run multiple copies at the same time, and the library will distribute messages to all the receivers. Make sure `scopserver` is running before trying them out.

```
// sender.cpp - DMI - 7-9-02

/* Usage: sender [ <message> ]   (default message is "Hello world!") */

#include <scop.h>

int main(int argc, char **argv)
{
   int sock;
   char *msg = argc > 1 ? argv[1] : (char *)"Hello world!";

   sock = scop_open("localhost", "sender");
   scop_send_message(sock, "receiver", msg);

   close(sock);
   return 0;
}
```

```
// receiver.cpp - DMI - 7-9-02

#include <scop.h>

int main()
{
   int sock;
   char *msg;
```

```
   sock = scop_open("localhost", "receiver");
   while(1)
   {
      msg = scop_get_message(sock);
      printf("Received <%s>\n", msg);
      if(!strcmp(msg, "quit")) break;
      delete[] msg;
   }

   close(sock);
   return 0;
}
```

4.4 Predefined event sources

```
void scop_set_source_hint(int sock, const char *endpoint);
int scop_emit(int sock, const char *message, int verify = 0);
```

Every connection may have a *source hint*. If set, you can omit the destination endpoint for messages by sending them with `scop_emit` instead of `scop_send_message`; they will go to the endpoint specified by the source hint.

The motivation for this is that while some message-based programs send messages to lots of different named targets, programs that behave as sources of general-purpose events typically direct them to a specific endpoint (which others can listen to). Specifying it in advance is not done to improve efficiency but to improve status reporting and the reflective power of the system. For example, `scop_list` reports source hints, which is useful as a reminder of the endpoint clients must listen to in order to receive events from a particular source.

Once the source hint has been set for a connection it can be changed at any time, and `scop_emit`'s may be mixed in with `scop_send_message`'s to arbitrary targets; so this does not restrict what you can do with the connection nor guarantee where it will actually send messages. Nevertheless, it is a useful concept in some cases.

`scop_emit` returns the same values as `scop_send_message`.

Event sources example

The programs `event_source` and `event_listener` demonstrate source hints and listening to endpoints. `event_source` emits a stream of events, one per second.

The third program below, `multi_listener`, is more interesting. This is a replacement for `event_listener` which listens to two different endpoints at once (`news` and `updates`). You can try running `multi_listener` and then starting two copies of `event_source`, one with the command line argument `updates` (which changes the name of the source). In this case `multi_listener` will display a stream of interleaved events from the two different sources. Observe the effect when you stop and start the event sources whilst `multi_listener` is running. If you run `scop list` at this point you will see the following useful summary of the communication in progress. Note that the first two connections are from a single process.

```
4 clients connected:
Client connection <multi_listener> listening to <news>, source hint <>
Client connection <multi_listener> listening to <updates>, source hint <>
Client connection <event_source> listening to <>, source hint <news>
Client connection <event_source> listening to <>, source hint <updates>
```

The code for `multi_listener` uses `select()` to process messages from the two separate connections.

```
// event_source.cpp - DMI - 7-9-02

/* Usage: event_source [ <source> ]   (default source is "news") */

#include <scop.h>

int main(int argc, char **argv)
{
   int sock;
   int count = 1;
   char msg[80];

   sock = scop_open("localhost", "event_source");
   scop_set_source_hint(sock, argc > 1 ? argv[1] : (char *)"news");

   while(1)
   {
      sprintf(msg, "Item %d", count);
      scop_emit(sock, msg);
      count++;
      sleep(1);
   }

   return 0;
}
```

```
// event_listener.cpp - DMI - 7-9-02

#include <scop.h>

int main()
{
   int sock;
   char *msg;

   sock = scop_open("localhost", "event_listener");
   scop_listen(sock, "news");
   while(1)
   {
      msg = scop_get_message(sock);
      printf("Received <%s>\n", msg);
      delete[] msg;
   }

   return 0;
}
```

```
// multi_listener.cpp - DMI - 7-9-02

/* Usage: multi_listener [ <source-one> <source-two> ]
   (default sources are "news" and "updates") */

#include <sys/select.h>

#include <scop.h>

int main(int argc, char **argv)
{
   int sock[2];
   char *msg;
   fd_set read_fds;
   int max_fd;

   for(int i = 0; i < 2; i++)
      sock[i] = scop_open("localhost", "multi_listener");
   scop_listen(sock[0], argc == 3 ? argv[1] : (char *)"news");
   scop_listen(sock[1], argc == 3 ? argv[2] : (char *)"updates");

   while(1)
   {
      FD_ZERO(&read_fds);
      max_fd = 0;
```

```
      for(int i = 0; i < 2; i++)
      {
         FD_SET(sock[i], &read_fds);
         if(sock[i] > max_fd) max_fd = sock[i];
      }
      select(max_fd + 1, &read_fds, NULL, NULL, NULL);

      for(int i = 0; i < 2; i++)
      {
         if(FD_ISSET(sock[i], &read_fds))
         {
            msg = scop_get_message(sock[i]);
            printf("Received <%s> from %s\n", msg,
                  i == 1 ? "updates" : "news");
            delete[] msg;
         }
      }
   }

   return 0;
}
```

4.5 RPC functions

```
char *scop_rpc(int sock, const char *endpoint, const char *args);
char *scop_get_message(int sock, int *rpc_flag = NULL);
int scop_send_reply(int sock, const char *reply);
```

`endpoint` refers to the connection name of the target server. It must exist and be unique, otherwise this call fails and return `NULL`. `scop_rpc` also returns `NULL` if `scopserver` has disconnected. `scop_send_reply` returns `0` except when `scopserver` has disconnected, in which case it returns `-1`.

Servers *must* reply to RPC requests in FCFS order so that `scopserver` can match each reply up with the corresponding request. If you skip an RPC, callers will get incorrect answers (probably causing an imminent crash) and clients will remain blocked.

Note the extended version of `scop_get_message` (if `rpc_flag` isn't `NULL`, it is set to 1 or 0 to indicate whether an RPC request or an event message was received, respectively. In the former case, the `scop-rpc-call` protocol is removed and the remainder of the buffer is copied to a new area, so you can safely delete it as usual).

RPC example

This illustrates a server which takes a string argument and then echoes back each character twice (so the reply to the default query will be `HHeellllloo wwoorrlldd!!`).

```
// client.cpp - DMI - 7-9-02

/* Usage: client [ <query> ]   (default query is "Hello world!") */

#include <scop.h>

int main(int argc, char **argv)
{
   int sock;
   char *query = argc > 1 ? argv[1] : (char *)"Hello world!";
   char *reply;

   sock = scop_open("localhost", "client");
   reply = scop_rpc(sock, "server", query);
   printf("Query <%s>, Reply <%s>\n", query, reply);
   delete[] reply;

   close(sock);
   return 0;
}
```

```
// server.cpp - DMI - 7-9-02

#include <scop.h>

int main()
{
   int sock;
   char *query, *reply;
   int len;

   sock = scop_open("localhost", "server");
   while(1)
   {
      query = scop_get_message(sock);
      len = strlen(query);
      reply = new char[len * 2 + 1];
      for(int i = 0; i < len; i++)
```

```
            reply[i * 2] = reply[i * 2 + 1] = query[i];
         reply[len * 2] = '\0';
         scop_send_reply(sock, reply);
         delete[] reply;
         delete[] query;
      }

      close(sock);
      return 0;
}
```

4.6 Admin functions

```
struct list_node
{
   char *name, *interest, *src_hint;
   list_node *next;
};
```

```
int scop_query(int sock, const char *endpoint);
void scop_clear(int sock, const char *endpoint);
void scop_set_log(int sock, int log_level);
list_node *scop_list(int sock, int *count);
void scop_terminate(int sock);
void scop_reconfigure(int sock);
```

`scop_query` returns the number of clients connected to a given endpoint (which may be a name or an interest).

`scop_clear` disconnects all clients connected to a given endpoint (again, this could be due to their name or interest). After a `scop_clear` you can be sure that no-one is listening to the endpoint which you cleared.

`scop_set_log` changes the `scopserver`'s log level whilst running. Valid log levels are currently 0 (only serious errors), or 1 (almost every operation).

`scop_list` returns a linked list, which should be deleted by the caller after use. This can be done just by deleting the first element of the list (the destructor will cascade down the list and delete all the embedded strings).

`scop_terminate` requests that `scopserver` shutdown, and `scop_reconfigure` causes it to re-read the authorised hosts file (but not to close any existing connections, even if they would no longer be accepted).

4.7 Cookies

```
void scop_set_plain_cookie(int sock, const char *text);
char *scop_get_plain_cookie(int sock, const char *name);
```

There is a maximum cookie size of 4 KB per connection.

It is recommended that the XML versions of these functions (see §5.7) are always used instead by applications.

Error-checking example

I'll close this chapter with one final example which actually includes error checking (just to show it isn't hard!) It's a server which simply logs all the messages it receives using the standard `syslog` facility. This is actually a real program which I used to get error messages from a process running on an embedded system where `syslog` wasn't available but TCP sockets were.

```
// sos.cpp - DMI - 3-1-2002

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <syslog.h>

#include <sys/types.h>
#include <sys/socket.h>

#include <scop.h>

int main(int argc, char **argv)
{
   int sock;
   char *buf;
   int rpc_flag;

   sock = scop_open("localhost", "sos", 1);
   if(sock == -1)
   {
      printf("Can't connect to scopserver.\n");
      exit(0);
   }
```

```
   if(fork() > 0) exit(0); // Detach

   while(1)
   {
      buf = scop_get_message(sock);
      if(buf == NULL)
      {
         syslog(LOG_INFO, "Lost connection to scopserver.\n");
         exit(0);
      }
      syslog(LOG_INFO, "%s\n", buf);
      delete[] buf;
   }

   close(sock);
   return 0;
}
```

4.8 Undocumented examples

There are a few more small programs in the examples directory which aren't discussed in this manual. They are `rtt_client` (used for performance measurements), `multiplex` and `multiplex_listener` (wrappers for `select()`). These can safely be ignored!

5 API Level 2 - Types with XML

5.1 General semantics

The level 2 API adds support for typed and structured data, which is encoded into strings for delivery via level 1 using a kind of XML format. Data is stored in a special type called `vertex`. The API provides functions to pack typed data into vertices (and to combine vertices to form larger data structures), a set of higher level SCOP functions which transport vertices rather than strings, and a third set of functions to extract the raw data from a `vertex` afterwards.

It is important to realise that the format of received messages and RPC call arguments is not checked for you by the library. This isn't because we don't believe in type checking but because it is seen as something which should be implemented as a higher level service (for example using IDL). Currently this is up to the application (but it could be supported by a new SCOP API layer in future). As usual we omit error checking from our examples; in practice you should be more careful unless you can be absolutely sure what type everything will be.

An XML `vertex` tree is freed by deleting the top node only. This deletes all the sub-vertices. Any `char *` or `byte *` buffers contained within the tree are deleted by this too, so you don't have to delete the results of the `extract_string` and `extract_bytes` functions. This also implies that you must make deep copies of them if you want to save the results beyond the `vertex` deletion.

To use level 2, add this to your source files:

```
#include <scopxml.h>
```

5.2 Marshalling

Marshalling primitive datatypes:

```
vertex *pack(int n);
vertex *pack(const char *s);
vertex *pack(double x);
vertex *pack(byte *buf, int bytes);
```

Constructing lists:

```
vertex *mklist();
vertex *append(vertex *list, vertex *v);

vertex *pack(vertex **vert_array, int n);

vertex *pack(vertex *v1, vertex *v2);
vertex *pack(vertex *v1, vertex *v2, vertex *v3);
...
vertex *pack(vertex *v1, ..., vertex *v6);
```

The last group of list construction calls are convenience functions for making short lists (between two and six items) without resorting to arrays or `append` calls.

All of the list functions "consume" their arguments, in the sense that they are linked into the overall tree and you don't have to worry about them beyond that point (recall that child vertices are deleted when the parent is).

5.3 Unpacking

Extracting primitive data types:

```
int vertex::extract_int(int item = -1);
double vertex::extract_double(int item = -1);
char *vertex::extract_string(int item = -1);
void *vertex::extract_bytes(int item = -1);
int vertex::count_bytes(int item = -1);
```

Extracting from lists:

```
vertex *vertex::extract_item(int item);
int vertex::count_items();

vertex **vertex::extract_array();

char *vertex::extract_method();
vertex *vertex::extract_args();
```

The latter two calls are convenience methods to support RPC method names (see §5.6).

The optional `item` arguments to the primitive extraction functions are a short-cut for extracting them from lists. For example, `v->extract_int(3)` means the same thing as `v->extract_item(3)->extract_int()`; in both cases `v` must be a list, the fourth item of which is an integer.

5.4 XML Message functions

```
int scop_send_struct(int sock, const char *endpoint, vertex *args, const
char *method = NULL);
vertex *scop_get_struct(int sock, int *rpc_flag = NULL);
```

These calls are similar to `scop_send_message` and `scop_get_message`, but operate on vertices rather than character strings. We shall defer discussion of the `method` parameter to §5.6. The return value from `scop_send_struct` is always `0` unless `scopserver` has disconnected in which case it returns `-1`.

XML example

This is an example of sending structured data. I've chosen an associative array — an address book containing (name, address) tuples. It's a good idea to put the marshalling code in a function by itself, and I've created an `AddressBook` class to take care of this. The example consists of four files; both `xml_sender` and `xml_receiver` depend on the header file `address_book.h` and need to be linked with `address_book.o`

If you run `xml_receiver` with the command line option `-inspect` it will show the actual XML it receives from `xml_sender` across the network as well as the final unpacked data structure. The `pretty_print` function is used to achieve this (its output is the same as the real wire format, except for whitespace added to enhance readability).

```
// address_book.h - DMI - 7-9-02

class AddressBook
{
   public:

      char **name;
      char **address;
      int entries;

      AddressBook(int size);
      ~AddressBook();

      void set_entry(int i, char *n, char *a);
      void dump();

      vertex *marshall();
      AddressBook(vertex *v);
};
```

```
// address_book.cpp - DMI - 7-9-02

#include <stdlib.h>
#include <stdio.h>

#include <scop.h>
#include <scopxml.h>

#include "address_book.h"
```

```
AddressBook::AddressBook(int size)
{
   entries = size;
   name = new (char *)[entries];
   address = new (char *)[entries];
   for(int i = 0; i < entries; i++)
      name[i] = address[i] = NULL;
}

AddressBook::~AddressBook()
{
   for(int i = 0; i < entries; i++)
   {
      if(name[i]) delete[] name[i];
      if(address[i]) delete[] address[i];
   }
   delete[] name;
   delete[] address;
}

void AddressBook::set_entry(int i, char *n, char *a)
{
   name[i] = new char[strlen(n) + 1];
   address[i] = new char[strlen(a) + 1];
   strcpy(name[i], n);
   strcpy(address[i], a);
}

void AddressBook::dump()
{
   for(int i = 0; i < entries; i++)
      printf("Name %s, Address %s\n", name[i], address[i]);
}

vertex *AddressBook::marshall()
{
   vertex *list, *tuple;

   list = mklist();
   for(int i = 0; i < entries; i++)
   {
      tuple = pack(pack(name[i]), pack(address[i]));
      append(list, tuple);
   }
   return pack(pack(entries), list);
}

AddressBook::AddressBook(vertex *v)
```

```
{
   vertex *list, *tuple;
   char *n, *a;

   entries = v->extract_int(0);
   name = new (char *)[entries];
   address = new (char *)[entries];
   list = v->extract_item(1);
   for(int i = 0; i < entries; i++)
   {
      tuple = list->extract_item(i);
      n = tuple->extract_string(0);
      a = tuple->extract_string(1);
      name[i] = new char[strlen(n) + 1];
      address[i] = new char[strlen(n) + 1];
      strcpy(name[i], n);
      strcpy(address[i], a);
   }
}
```

```
// xml_sender.cpp - DMI - 7-9-02

#include <scop.h>
#include <scopxml.h>

#include "address_book.h"

int main()
{
   int sock;
   AddressBook ab(3);
   vertex *v;

   ab.set_entry(0, "Poirot", "Belgium");
   ab.set_entry(1, "Morse", "Oxford, UK");
   ab.set_entry(2, "Danger Mouse", "London, UK");
   v = ab.marshall();

   sock = scop_open("localhost", "xml_sender");
   scop_send_struct(sock, "xml_receiver", v);
   delete v;

   close(sock);
   return 0;
}
```

```
// xml_reciever.cpp - DMI - 7-9-02

/* Usage: xml_reciever [-inspect] */

#include <scop.h>
#include <scopxml.h>

#include "address_book.h"

int main(int argc, char **argv)
{
   int sock;
   AddressBook *ab;
   vertex *v;

   sock = scop_open("localhost", "xml_receiver");
   v = scop_get_struct(sock);
   if(argc == 2 && !strcmp(argv[1], "-inspect"))
   {
      char *c = pretty_print(v);
      printf("%s\n", c);
      delete[] c;
   }
   ab = new AddressBook(v);
   delete v;
   ab->dump();
   delete ab;

   close(sock);
   return 0;
}
```

5.5 XML RPC functions

Here we present versions of the RPC functions which have `vertex` arguments and results.

Remember that SCOP views RPC's fundamentally as communications to which a *reply* is expected. If the procedure you are trying to call remotely has a `void` return value you should therefore treat it as an XML *message* and use `scop_send_struct` rather than these RPC calls.

```
vertex *scop_rpc(int sock, const char *endpoint, vertex *args, const char
*method = NULL);
vertex *scop_get_request(int sock);
int scop_send_reply(int sock, vertex *reply);
```

In strict adherence to the memory management conventions, `args` is *not* deleted for you by `scop_rpc`, and `reply` is *not* deleted by `scop_send_reply` (even though in both cases you almost always want to do so immediately thereafter):

The `method` parameter is discussed in §5.6.

`scop_rpc` returns `NULL` if the target endpoint does not exist or isn't unique.

`scop_send_reply` returns `0` except when `scopserver` has disconnected, in which case it returns `-1`.

If a message is sent to your connection whilst it is waiting in a `scop_rpc` call for a reply to a RPC, the message is thrown away. This is done because interrupting your wait would break the atomicity of the RPC API, and queueing the message for later would be quite a bit more complex to implement. Don't use the same connection for RPC's and as a message target at the same time.

If a message is sent to a server which is waiting in a `scop_get_request` call for an incoming RPC, the message is thrown away and the call returns `NULL`. You should use `scop_get_struct` instead if you need to check for messages as well as handling RPC's, or better still use separate connections. `NULL` is also returned by `scop_rpc` if `scopserver` has disconnected.

XML RPC example

The server in this case calculates binomial coefficients given arguments n and k (the number of ways of choosing k objects from n). I've no idea why you would want to do this in a different process, but it illustrates the principle well enough! You can pass the parameters to the client on the command line.

```
// xml_client.cpp - DMI - 7-9-02

/* Usage: xml_client [<n> <k>]    (default values n = 4, k = 2) */

#include <scop.h>
#include <scopxml.h>

int main(int argc, char **argv)
{
   int sock;
```

```
   int n, k;
   vertex *v, *w;

   if(argc != 3)
   {
      n = 4;
      k = 2;
   }
   else
   {
      n = atoi(argv[1]);
      k = atoi(argv[2]);
   }

   sock = scop_open("localhost", "xml_client");
   v = pack(pack(n), pack(k));
   w = scop_rpc(sock, "xml_server", v);
   delete v;
   printf("%d choose %d equals %d.\n", n, k, w->extract_int());
   delete w;

   close(sock);
   return 0;
}
```

```
// xml_server.cpp - DMI - 7-9-02

#include <scop.h>
#include <scopxml.h>

int combi(int n, int k);

int main()
{
   int sock;
   vertex *v, *w;

   sock = scop_open("localhost", "xml_server");
   while(1)
   {
      v = scop_get_request(sock);
      w = pack(combi(v->extract_int(0), v->extract_int(1)));
      delete v;
      scop_send_reply(sock, w);
      delete w;
   }
```

```
   close(sock);
   return 0;
}

int combi(int n, int k)
{
   int result = 1;
   if(k > n || k < 0)
      return 0;

   for(int i = 0; i < k; i++)
      result *= n - i;

   for(int i = 1; i <= k; i++)
      result /= i;

   return result;
}
```

5.6 XML RPC method names

SCOP does not directly support remote method names, because messages are pure ASCII or XML strings, without a remote address parameter. However there is no reason why you can't pack method names into the payload together with the arguments, using an XML two-element list. This is sufficiently useful that SCOP provides some convenience functions to help you with it, and optional `method` arguments to the XML versions of `scop_rpc` and `scop_send_struct` so that the method name doesn't have to be combined in a separate step beforehand.

If you are using this convention, it is possible that some of your methods may require no parameters at all (i.e. the method name alone is enough). In this case you may pass `NULL` for the `args` parameter.

Multiple RPC methods example

This illustrates the support for naming remote procedure calls. The server here implements three methods. `ctof` converts temperatures from centigrade to fahrenheit, `ftoc` does the reverse, and `stats` reports the number of times the server's main two methods have been invoked since it was started.

```
// method_client.cpp - DMI - 7-9-02

#include <scop.h>
#include <scopxml.h>

double cent_to_faren(int sock, double c);
double faren_to_cent(int sock, double f);
int count_uses(int sock);

int main()
{
   int sock;

   sock = scop_open("localhost", "method_client");

   printf("%g deg C = %g deg F.\n", 0.0, cent_to_faren(sock, 0.0));
   printf("%g deg C = %g deg F.\n", 20.0, cent_to_faren(sock, 20.0));
   printf("%g deg F = %g deg C.\n", 60.0, faren_to_cent(sock, 60.0));
   printf("The server has been accessed %d times.\n", count_uses(sock));

   close(sock);
   return 0;
}

double cent_to_faren(int sock, double c)
{
   vertex *v, *w;
   double f;

   v = pack(c);
   w = scop_rpc(sock, "method_server", v, "ctof");
   delete v;
   f = w->extract_double();
   delete w;
   return f;
}

double faren_to_cent(int sock, double f)
{
   vertex *v, *w;
   double c;

   v = pack(f);
   w = scop_rpc(sock, "method_server", v, "ftoc");
   delete v;
   c = w->extract_double();
   delete w;
   return c;
```

```
}

int count_uses(int sock)
{
   vertex *v, *w;
   int n;

   v = pack(0); // Dummy argument
   w = scop_rpc(sock, "method_server", v, "stats");
   n = w->extract_int();
   delete v;
   delete w;
   return n;
}
```

```
// method_server.cpp - DMI - 7-9-02

#include <scop.h>
#include <scopxml.h>

int invocations = 0;

double cent_to_faren(double c);
double faren_to_cent(double f);

int main()
{
   int sock;
   vertex *v, *w, *args;
   char *method;

   sock = scop_open("localhost", "method_server");
   while(1)
   {
      v = scop_get_request(sock);
      method = v->extract_method();
      args = v->extract_args();
      if(!strcmp(method, "ctof"))
         w = pack(cent_to_faren(args->extract_double()));
      else if(!strcmp(method, "ftoc"))
         w = pack(faren_to_cent(args->extract_double()));
      else if(!strcmp(method, "stats"))
         w = pack(invocations);
      else
         exit(1);
      delete v;
```

```
      scop_send_reply(sock, w);
      delete w;
   }

   close(sock);
   return 0;
}

double cent_to_faren(double c)
{
   invocations++;
   return (9.0 * c / 5.0) + 32.0;
}

double faren_to_cent(double f)
{
   invocations++;
   return (f - 32.0) * 5.0 / 9.0;
}
```

5.7 XML Cookies

```
vertex *scop_get_cookie(int sock, const char *name);
void scop_set_cookie(int sock, vertex *data);
```

5.8 Conversion functions

These are internal functions which you should never need to call directly (documented only for the curious):

```
char *vertex_to_string(vertex *v);
vertex *string_to_vertex(const char *s);
char *vertex_to_string(vertex *v, const char *method);
```

The latter call is a convenience function to support RPC method names.

5.9 Debugging

```
char *pretty_print(vertex *v);
```

6 API Level 3 - Reflection

This level is not implemented yet. To be forward compatible with it, you (hopefully!) only need to observe a convention for the data stored in cookies. These should always be formatted using the XML routines, and consist at the top-level of a two-element list. The first branch of the list is reserved for system data, whereas the second is available for free-form user data.