SCOP Manual Annex — Java Binding

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

Contents

1 Introduction

This annex to the SCOP manual describes the Java language binding. If your project is written in Java note that some of the SCOP functionality can and probably should be replaced by object serialization and Java RMI. SCOP Java is most useful as an events system, and for interoperability. Nevertheless the Java binding provides access to the *complete* SCOP API. The port is implemented in pure Java (not as a wrapper around the C++ functions).

The API works in a very similar way to the C++ one. You should therefore refer to the main manual for details of what each method does. In general the Java binding is a little easier to use because it is simplified by some features of the Java language.

1.1 Performance

Using interpreted Java (Sun's SDK 1.4) local-case performance was measured on an 800 Mhz Pentium III as peaking at 1000 RPC's per second (about seven times slower than with the C++ binding). Of course the `scopserver` process is still implemented in C++.

2 General semantics

The socket file descriptor is replaced by an instance of class `SCOP` for each connection to `scopserver`. All SCOP functions are methods in this class, which do not need the `scop_` prefix or the `sock` parameter. `scop_open` is replaced by the `SCOP` class constructor. You can call `connection_ok` after creating a new `SCOP` object to discover if the connection to `scopserver` was made successfully.

There is of course no need to free anything because buffers are dealt with by the garbage collector. The connection itself must be closed with the `close` method, however. Character arrays are obviously replaced by `String`'s and `boolean` values are used instead of `int`'s where appropriate.

The source code for the Java binding can be found in `java/SCOP.java` within the main distribution. The `Makefile` located in that directory will compile this to a single Java bytecode class file, `SCOP.class`.

Note that there is a new method `reply_required`, which can be checked after a `get_message` or `get_struct` to see if the message was an RPC.

2.1 Restrictions

The following restrictions apply, compared to the C++ binding:

- Log data cannot be written using `syslog`, nor is the `SCOP_LOGFILE` environment variable used. The value `~/.scoplog` is hard-coded for the log file name.
- The port number cannot be read from `/etc/services` (the default number is hard-coded instead).

2.2 Example programs

Java versions of the C++ example programs described in this manual can be found in the directory `java/examples`. They behave in an identical manner to their C++ counterparts. You can of course talk to the C++ versions of the servers with the Java clients, and vice-versa. The source code for the examples should make it clear how to use the Java API. Don't forget to set your classpath so that Java can find the library, for example:

```
java -cp .:..  Receiver
```

The examples provided are `Sender`, `Receiver`, `EventSource`, `EventListener`, `MultiListener`, `Client`, `Server`, `SOS`, `RTTClient`, `XMLSender`, `XMLReceiver`, `XMLClient`, `XMLServer`, `MethodClient` and `MethodServer`.

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

The Java version of the `SOS` program has little point because we cannot easily access `syslog`. The example provided just echoes the messages it receives to `stdout`, making it a standard listener (with error checking).

The undocumented C++ example consisting of `multiplex` and `multiplex_listener` has no Java equivalent because it is just a wrapper for C's `select()` function.

2.3 Multiplexing connections

For a long time Java had no equivalent to the `select()` function, which made non-blocking and multiplexed calls impossible without multiple threads. The NIO API does provide a `Selector` class now, however. A `channel()` method has been added to the SCOP API to retrieve the underlying `SocketChannel` object, so that NIO can be used.

NIO is meant to allow handling of *all* I/O in a non-blocking fashion, and unfortunately it doesn't support the SCOP library's slightly weaker semantics (selecting on multiple inputs until one becomes active, then issuing blocking I/O calls to read from it) very well. Essentially the problem is that a socket channel has to be in either blocking or non-blocking mode; one can't perform blocking calls in non-blocking mode or do a select in blocking mode, changing to blocking mode requires that the socket first be deregistered from the `Selector`, and to do a select it must of course be registered.

A Java version of the `MultiListener` example (with `select()` calls replaced by a `Selector`) is provided but because of this restriction has to be implemented in a cumbersome (and slow) manner at present.

3 Level 1 API

```
public class SCOP
{
   // Connection setup and teardown:
   public SCOP(String host, String name);
   public SCOP(String host, String name, boolean unique);
   boolean connection_ok();
   SocketChannel channel();
   void listen(String interest);
   void listen(String interest, boolean unique);
   void close();

   // Messaging:
   void send_message(String endpoint, String message);
   int send_message(String endpoint, String message, boolean verify);
   String get_message();

   // RPC:
   boolean reply_required();
   String rpc(String endpoint, String args);
   void send_reply(String reply);

   // Predefined event sources:
   void set_source_hint(String endpoint);
   void emit(String message);
   int emit(String message, boolean verify);

   // Admin:
   int query(String endpoint);
   void clear(String endpoint);
   void set_log(int log_level);
   void terminate();
   void reconfigure();

   // Cookies:
   void set_plain_cookie(String text);
   String get_plain_cookie(String name);
}
```

4 Level 2 API

4.1 Marshalling and Unpacking

```
public class Vertex
{
   /* Marshalling: */

   static Vertex pack(int n);
   static Vertex pack(String s);
   static Vertex pack(double x);
   static Vertex pack(byte[] buf);

   static Vertex mklist();
   static Vertex append(Vertex list, Vertex v);
   static Vertex pack(Vertex[] vert_array);

   // Convenience functions for making short lists:
```

```
   static Vertex pack(Vertex v1, Vertex v2);
   static Vertex pack(Vertex v1, Vertex v2, Vertex v3);
   static Vertex pack(Vertex v1, Vertex v2, Vertex v3, Vertex v4);
   static Vertex pack(Vertex v1, Vertex v2, Vertex v3, Vertex v4, Vertex v5);
   static Vertex pack(Vertex v1, Vertex v2, Vertex v3, Vertex v4, Vertex v5,
         Vertex v6);

   /* Unpacking */

   int extract_int();
   double extract_double();
   String extract_string();
   byte[] extract_bytes();
   int count_bytes();

   int extract_int(int item);
   double extract_double(int item);
   String extract_string(int item);
   byte[] extract_bytes(int item);
   int count_bytes(int item);

   Vertex extract_item(int item);
   int count_items();

   Vertex[] extract_array();

   String extract_method(); // Convenience
   Vertex extract_args();   // Convenience

   /* Parsing: */

   static String vertex_to_string(Vertex v);
   static Vertex string_to_vertex(String s);
   static String vertex_to_string(Vertex v, String method); // Convenience

   /* Debugging: */

   static String pretty_print(Vertex v);
}
```

4.2 XML transport functions

```
public class SCOPXML extends SCOP
{
   // XML messages:
   void send_struct(String endpoint, Vertex args);
   void send_struct(String endpoint, Vertex args, String method);
   Vertex get_struct();

   // XML RPC:
   Vertex rpc(String endpoint, Vertex args);
   Vertex rpc(String endpoint, Vertex args, String method);
   Vertex get_request();
   void send_reply(Vertex reply);

   // XML Cookies:
   Vertex get_cookie(String name);
   void set_cookie(Vertex data);
}
```