SCOP Overview and Installation Guide

Version 1.5, 11th September 2006

David Ingram (`dmi1000@cam.ac.uk`)

Contents

1 Overview

1.1 What does it do?

SCOP consists of a user library and server process designed to make writing distributed applications (particularly in C, C++, Python and Java) very easy. It allows different processes (which may be written in different languages, on different machines and architectures — or the same machine of course) to communicate by means of sending each other *messages* or *events*, with very little effort from the programmer.

1.2 Why is it better than XYZ?

UNIX/Linux lacks a standard, built-in events system and this has been detrimental to the development of large scale integrated projects such as desktop environments and component embedding. Pipes and scripts do a good job for largely sequential command-line utilities; but there is no easy way to apply the UNIX philosophy of sharing data, automating and combining smaller tools to concurrent, possibly distributed, graphical programs.

There are plenty of existing events systems which you can install, but these tend to be either (i) huge and clunky, such as CORBA (frustration with which motivated this project), (ii) unpolished research projects, or (iii) special-purpose, for example almost every window system and graphics toolkit has an internal events system. SCOP is small, complete and general-purpose. The acronym stands rather arbitrarily for Systems COmmunication Protocol (by analogy with KDE's DCOP, which does something similar in the context of a desktop environment). It is pronounced "es-cop".

1.3 Features

The functionality offered by SCOP includes:

- Message passing
- Events
- Resource discovery
- RPC

Of course message passing and event delivery can be handled by the same mechanism, and indeed I shall make no further distinction between them. Resource discovery is facilitated by the ability to query which clients are connected to the server, and retrieve some state (a cookie) associated with each active connection. There are no problems with state left over from dead connections, unlike some file locking solutions.

RPC (Remote Procedure Call) is supported to the extent of allowing messages to have replies, and providing separate XML-based routines to package up (marshal) arguments and results. There is quite deliberately no IDL or automatic stub generation, so no preprocessing or machine-generated code is involved. I feel that the advantages of understanding exactly what is going on and not having a special build process imposed upon your project outweighs the alleged disadvantage of remote function calls not looking exactly like local ones. The extra work required to manually wrap up arguments is quite modest, and forces the programmer to think about efficiency a little bit whilst they are doing it. Although the system does not therefore provide *distribution transparency*, it is in another sense quite transparent by making it obvious to the reader of the code exactly when, where and what messages are being sent over the network and which format they are in.

1.4 Good things

Here are some good things about SCOP:

- Very simple to learn and use.
- Cleanly layered APIs allow you to use more rudimentary low-level interfaces if you don't want extra fancy high level features.
- Plain text wire format (with a basic XML subset if you are using the typed API), so if you don't know what is going on you can always inspect the messages being sent, and parse them yourself if you like.
- Memory management nightmares with C/C++ are avoided (without introducing error-prone pseudo-garbage-collection features) through a clearly defined and straightforward allocation/deallocation rule.
- Integrates with other file and socket based I/O through the standard `select` call.
- Doesn't take over your thread of control by imposing an event loop.
- No additional threads created (unless you do so explicitly yourself).

Don't be put off by the length of this manual, because simple things are still very, very simple to do. For example, here's how to multicast a message to a bunch of other processes:

```
int sock = scop_open("localhost", "myclient");
scop_send_message(sock, "groupname", "Hello, world!");
```

Fantastic `:-)`

1.5 Performance

SCOP is very lightweight, so although speed wasn't the main design goal, performance is reasonably good. My intuition is that the overhead due to the use of ASCII messages compares pretty well with the complex type system processing incorporated in most binary alternatives.

Typical figures for same-machine communication are a round trip time of 150μs on a 800 Mhz Pentium III (around 6700 sequential RPC's per second). Inter-machine communication was measured as 450μs per round trip, giving 2300 sequential RPC's per second.

1.6 Bad things

A current weakness of SCOP is that it has rather primitive security, and very little error checking. It's easy to crash the clients through incorrect use of the protocol, if you want to. It's almost certainly possible to crash the server likewise, although the design is supposed to make this less likely. The server is single-threaded so denial of service attacks are trivial. Of course, the source code is available so anyone can strengthen the security if this becomes necessary. Otherwise it is advisable to run it inside a firewall.

The two security features which *are* provided are privilege reduction and hostname access control. If you run `scopserver` as root it will automatically change itself to user and group `nobody` (assuming they exist on your system) as soon as it starts, which helps by limiting the mischief a runaway server could do quite a bit. `scopserver` will only allow access from clients running on hosts which are listed in its authorised host file. This means that any user on any permitted machine has unconstrained access to SCOP, but hosts you haven't explicitly included cannot connect at all.

2 Getting started

2.1 Requirements

Nothing specific, other than `gcc` and GNU `make`.

SCOP has been developed on Linux, but should be portable to UNIX environments with a little effort. It has been tested successfully on Slackware, Mandrake and Redhat Linux systems, on FreeBSD and on Solaris. I would be interested to hear what changes need to be made to get it to build on other UNIX variants.

2.2 Where to get SCOP

You can download it from

`http://www.srcf.ucam.org/~dmi1000/scop/`

2.3 Compilation

```
tar xzvf scop.tar.gz
cd scop
make
```

2.4 Installation as root

If you have root access on your chosen machine, you should install SCOP like this. It is perfectly possible to install it without being root, but slightly less convenient in use.

```
su -c make install
su -c echo >>/etc/services "scop 51234/tcp"
```

Note: you can choose a different port number in the last line if you like, as long as you pick the same one for each machine which needs to communicate. The services entry must exist even on machines which only run clients and don't need the server process installed.

2.5 Non-root installation

If you don't have root access to the machine, SCOP can be installed like this.

```
mkdir ~/include ~/lib ~/bin
make userinstall
```

You should also add the following to your shell startup script (`.bashrc`, for example):

```
export PATH=$PATH:~/bin
export SCOP_LOGFILE=~/.scoplog
```

Since you can't update `/etc/services`, scop will use the standard port number (51234) when it finds it can't look up the service entry. This value can be changed at compile time by editing the constant `FALLBACK_PORT` in `scop.h`

2.6 Configuration

The only configuration task is to define the set of machines which should be allowed to connect to SCOP. This is done by editing the host access control file, `/etc/scophosts` (root installations) or `~/.scophosts` (non-root installations). The format consists of fully-qualified host names, e.g. `machinename.foo.bar.com` or whatever, one per line.

The default file just contains the special name `localhost`, which allows connections from the same machine that you are going to run the `scopserver` process on. If you have trouble connecting to the server you might need to add the name of your machine explicitly (not doing this is a common cause of mysterious failure). The SCOP server process only reads this file when it starts up, so you must kill and then restart the server if you subsequently modify the list of authorised hosts.

2.7 Starting the server

If necessary add `/usr/local/sbin` to your `$PATH`, then simply run

```
scopserver
```

In a non-root installation, run this instead:

```
scopserver -f ~/.scophosts
```

The server will detach and run in the background automatically. You probably want to add this command to the system's `/etc/rc.d/rc.local` file so the server starts automatically at boot time, for root installations. It is not possible to run more than one copy of the server per machine. In fact neither message sources nor sinks need a locally running server, since they can be directed to route via a server running on a third machine.

Check the server is running by looking for `scopserver` in the system's process list. If it doesn't seem to have started correctly, you should look for error messages in the SCOP log (see the next section). All problems are reported there, and not to the console.

Log file location

If you have set the `SCOP_LOGFILE` environment variable, log messages are appended to the file you specified.

If `SCOP_LOGFILE` isn't set, an `scopserver` running as root will use the `syslog` facility to issue log messages instead. They will therefore generally appear in a system log file such as `/var/log/messages`. You can change `/etc/syslog.conf` if this isn't the case.

In cases where `SCOP_LOGFILE` isn't set but `scopserver` is running as a normal user (not root), messages are logged to a default location which is `/.scoplog` instead. The reason we don't use `syslog` for non-root installations is that it typically isn't readable by ordinary users. Don't forget you can override all this by setting `SCOP_LOGFILE` yourself (perhaps to `/tmp/scoplog`, for example).

Server options

As well as the use of `scopserver -f <hostfile>` to specify the location of the access-control file, there are two other options you can pass to `scopserver`:

`scopserver --allhosts` deactivates access control altogether, allowing incoming connections from any machine.

`scopserver -p <port-number>` tells the server to listen for incoming connections on the port specified. This overrides any `/etc/services` entry and the default port number.

2.8 Testing it

Once the server is running, you can easily test your installation using the command line tools `scop` and `listen`, which will also have been installed. These clients should be run as a normal user; they don't require any special privilege. You are encouraged to work through the following tests since they also help illustrate SCOP's capabilities.

If you get "`Broken pipe`" error messages or things don't seem to be working, look at the SCOP log file, as described in §2.7. Any problems discovered in the SCOP library cause explanatory messages to be logged there, so it provides a single place to look for errors both from `scopserver` and the client programs.

Incidentally, you can instruct the server to be more verbose in its logging (typically this means logging successful commands as well as errors) by issuing the command `scop log 1`. You need to do this after `scopserver` has started, but before the behaviour you wish to monitor. It will generate a lot of output if your system is handling a high rate of events. A return to normal log verbosity is achieved with `scop log 0`.

For a quick and effective demonstration of multiple endpoints, you should open several shell windows on the machine which is running the server; lets say five windows for this test. Resize and shuffle them around until you can see all of them. In this section commands to be typed are prefixed by the window number into which you should type them. Then proceed as follows (don't be concerned if some of the commands appear to pause indefinitely – they are waiting for events):

```
1: scop --help
1: scop list
2: listen --help
2: listen magic
```

```
3: listen spell
4: listen magic
1: scop list
1: scop send spell "hello world"
1: scop send magic "foo"
1: scop send rhubarb "foo"
1: scop verify magic "bar"
1: scop verify rhubarb "bar"
1: scop rpc spell "backwards"
1: ping localhost | stream magic
5: scop list
5: listen -u magic
1: CTRL-C
1: scop list
1: scop clear spell
1: scop list
5: CTRL-C
1: scop list
```

Pay attention to the output you see in each window; it should be self-explanatory. The communication endpoints we set up in this example are called “magic” and “spell”. Note that multicast is performed automatically if several clients connect with the same name, and messages sent to non-existant endpoints are silently dropped. The `ping` command is used as an example stream source because on most systems it will generate a new line of output every second.

2.9 Networked usage

When you wish to communicate between several different machines, you should only run `scopserver` on one of them (separate instances of `scopserver` are separate worlds with their own endpoint namespaces, which don’t talk to each other).

You must tell the clients which are running on the other machines where the `scopserver` is located. You can do this with the `-r <hostname>` option to `scop` and `listen`. If you miss out the `-r` argument it is as if you said `-r localhost`, which is the standard alias expanding to the name of the machine the process is running on.

2.10 Linking your application with SCOP

Shared library support is being added to future versions, but currently the library has to be statically linked (fortunately it is rather small). Here’s how to do it.

In your source code, include the line

```
#include <scop.h>
```

Then build your application with a Makefile like this one:

```
foo: foo.cpp
   g++ -o foo foo.cpp -lscop
```

What could be easier? `:-)`

If SCOP has *not* been installed as root you need some extra flags, so use a Makefile like this instead:

```
foo: foo.cpp
   g++ -I ${HOME}/include -L ${HOME}/lib -o foo foo.cpp -lscop
```

3 API Level 4 - Command Line Interface

SCOP provides several layers of API, the highest of which is the command line interface ("layer 4"). Here is a brief summary of the shell commands available, for reference.

3.1 listen

```
Usage: listen [option...] <interest>

Options: -r <hostname>   remote scopserver
         -u              unique endpoint
         -p              persistent
         -e <command>    execute command with message arguments
```

The `listen` program displays the messages sent to an endpoint which you specify on the command line. For the sake of example it tries to guess if they are in XML format (pretty-printing them if so), and illustrates replying to RPCs by reversing the argument, treated as a string. More realistic servers would of course know what format messages to expect!

The `-e` option is intended to support quick and dirty remote-control functionality via shell scripts (not recommended for serious work — use the C API instead!) If you run

```
listen -e "shell command" <endpoint>
```

Then whenever a message with content `foo bar` is sent to `<endpoint>`, listen will execute the following as if you had typed it into the shell:

```
$ shell command foo bar
```

The `-p` option allows `listen` to continue running if the `scopserver` it is connected to exits. The `listen` process will then periodically attempt to reconnect to `scopserver`. It can also wait for `scopserver` to start if it isn't running initially.

The algorithm at present tries reconnecting once per second for a minute, then once every 2 seconds for 2 minutes, then once every 4 seconds for 4 minutes and so on. After trying once every 32 seconds for 32 minutes the program retries every 64 seconds indefinitely.

This feature together with the `-p` option for `stream`, below, enable one to create very reliable streams which automatically restart without failing when the `scopserver` process is killed (or, more probably, someone reboots the machine it is on). If you have cron jobs on the respective machines to restart the `scopserver`, the `stream` source and the `listen` process then you know everything will correctly restart itself regardless of the type of failure experienced.

3.2 stream

```
Usage: stream [option...] <endpoint>
Options: -r <hostname>   remote scopserver
         -p              persistent
```

The `stream` program reads lines of text from standard input, and sends each one in turn as a message to the endpoint specified. Piping the output of another command into `stream` is a very quick and easy way of converting an external event source into SCOP events.

The `-p` option works the same way as for the `listen` client, above. Whilst `scopserver` is down, `stream` will still keep reading from `stdin` to prevent a stall in the pipeline, but it will discard the incoming messages until `scopserver` reappears.

Note that a more sophisticated system would probably offer the option of buffering the messages and replaying them after `scopserver` is restarted in order not to lose any information. We can't do this in a straightforward way because it's unlikely the listeners would have restarted as soon as the `scopserver` is back, hence the replayed messages would probably be dropped by `scopserver` for want of a receiver anyway. A correct implementation of buffered messages would also have to execute `scop query` on the endpoint in question until the required listener[s] were back, before initiating message replay.

3.3 scop

```
Usage: scop [<opts>] send {<endpoint> <message>}+
       scop [<opts>] verify <endpoint> <message>
       scop [<opts>] list
       scop [<opts>] log 0|1
       scop [<opts>] query <endpoint>
       scop [<opts>] clear <endpoint>
       scop [<opts>] rpc <name> <message>
       scop [<opts>] xmlrpc <name> <method or "-"> <arg>+
       scop [<opts>] xmlsend <name> <method or "-"> <arg>+
       scop [<opts>] terminate
       scop [<opts>] reconfigure

Options: -r <hostname>
```

`scop` is a general-purpose command line client. It is mainly used for sending messages and listing the active connections. See the walkthrough in §2.8 for some examples of its use.

A License notice