

SPOCP Client Implementation Guide

by Roland Hedberg (Umeå University), Torbjörn Wiberg (Umeå University)

Table of contents

1 Introduction.....	2
2 Client library.....	2

1. Introduction

When you want to Spocpify an application, you not only have to decide on a query format (where/when you are going to pose queries to a Spocp server), but you also have to modify the source code for the application. This document details this process in an application that is written in C. There also exists client library implementations in Perl and Java, but they will not be dealt with here.

2. Client library

A more detailed description of the commands can be found in the doxygen documentation.

This text will describe some of the commands as they appear in the apache spocp authorization module. The full code can be found at the same place that you found this software, among the client implementations.

```
static sexpargfunc path_to_sexp;  
static sexpargfunc get_method;  
static sexpargfunc ac_mode;  
static sexpargfunc get_ip;  
static sexpargfunc get_host;  
static sexpargfunc get_inv_host;  
static sexpargfunc get_uid;  
static sexpargfunc get_authname;  
static sexpargfunc get_authtype;  
static sexpargfunc get_transpsec;  
static sexpargfunc get_ssl_vers;  
static sexpargfunc get_ssl_cipher;  
static sexpargfunc get_ssl_subject;  
static sexpargfunc get_ssl_issuer;  
static sexpargfunc get_authinfo;  
static sexpargfunc get_hostinfo;
```

First some functions that can be used to get information from the application: these are extremely application dependent, and probably cannot be used between applications, or sometimes not even between versions of the same application.

```
const sexparg_t transf[] = {  
    { "resource", path_to_sexp, 'l', TRUE },  
    { "method", get_method, 'a', FALSE },  
    { "acmode", ac_mode, 'a', FALSE },  
    { "ip", get_ip, 'a', FALSE },  
    { "host", get_host, 'a', FALSE },  
    { "invhost", get_inv_host, 'l', TRUE },  
    { "uid", get_uid, 'a', FALSE },  
    { "authname", get_authname, 'a', FALSE },  
};
```

SPOCP Client Implementation Guide

```
    { "authtype", get_authtype, 'a', FALSE },
    { "authinfo", get_authinfo, 'l', TRUE },
    { "hostinfo", get_hostinfo, 'l', TRUE },
#ifdef MOD_SSL
    { "ssl_vers", get_ssl_vers, 'a', FALSE },
    { "ssl_cipher", get_ssl_cipher, 'a', FALSE },
    { "ssl_subject", get_ssl_subject, 'a', TRUE },
    { "ssl_issuer", get_ssl_issuer, 'a', TRUE },
#endif
    { "transportsec", get_transpsec, 'l', TRUE }
} ;
```

This is about putting all the information gathering functions into a struct array which can later be used by the function that constructs S-expressions from a basic format definition and information pieces gathered by these functions.

```
char *httpquery =
"(4:http(8:resource%{resource}1:-)(6:action%{method}%{acmode})%{host\
info}%{authinfo})";
```

The format definition; every `%{...}` construct refers back to a line in the `transf` array. So, in the place occupied by `%{resource}` you would find that which the `path_to_sexp()` function returns.

```
httpq = parse_format(httpquery, transf, ntransf);
```

This is where the format definitions above are parsed into a more usable format.

```
    sa.r = r;
    sa.path = 0;
    sa.authz = authzt;
    sa.method = met;
#ifdef MOD_SSL
    sa.ssl = ssl;

    if (ssl)
        sexp = sexp_constr( (void *) &sa, a->sec_template ) ;
    else
#endif
        sexp = sexp_constr( (void *) &sa, a->template ) ;
```

And this is where the S-expression, the Spocp query, is constructed. The `sexp_constr` function takes two arguments. The first is a void pointer to something application specific; this is the information which the gathering function gets as argument. Secondly, there is a pointer to the parse S-expression format definition (which in this case could be `httpd`, if SSL/TLS is not used, or some user defined format). In this case different templates are used depending on whether SSL/TLS is used or not.

```
if (sc->spocp == 0)
    sc->spocp = spocpc_open( 0, sc->server, 1 ) ;
```

If there is no connection open, you will need to open one. In this case don't wait more than 1 second before declaring that the Spocp server was not reachable.

```
\
{
    if((rc = spocpc_send_query( spocp, 0, sexp, &qres )) == SPOCPC_OK )
    {
        if (qres.rescode == SPOCP_SUCCESS ) {
            if (qres.blob ) {
```

Send the query; if the operation went without any problems, check the result code that the Spocp server returned. If the result code indicated a success, look for the return info, if any.

```
if (sc->spocp != 0 ) { /* got a connection pose the question */
    if (pose_query( r, sc->spocp, sexp, &res ) ==
SPOCPC_CON_ERR)\
{
    spocpc_reopen( sc->spocp, 1 ) ;
    pose_query( r, sc->spocp, sexp, &res ) ;
}
}
```

If the query failed due to connection problems (the server wasn't there anymore) try to reconnect. If more than one server was defined in the server definition, all of them will be tried in a round-robin fashion before giving up. Again, the client will wait not more than 1 second before giving up on one server.