The **CipUX** XML-RPC Server and Client

Communication Across Boundaries

Christian Külker

Version 3.4.0.7

This document addresses the basic **CipUX** XML-RPC server and client communication specification and its usage. The server API is explained in detail and examples for accessing the server are given in Perl. Other XML-RPC client developers should be in a position to acquire the important information through the given examples to seamlessly develop own applications. Sections about tools, debugging hints and existing free open source XML-RPC client projects complete this paper.

Contents

1 Preface

This document supports developers of XML-RPC clients and describes practical parts of the XML-RPC server calls and responses in detail while other parts like installing are skipped. Others might find to read this document useful too. In the beginning a section shows different parts of the software package and the configuration of the main server part. It also includes a part of how using the server in a secure way. The following section introduce briefly the session concept and a short section thereafter draw a picture representing the categories of XML-RPC server calls. For the impatient two example calls and responses are displayed in a section by itself. One example is a didactically sum call while the other is simple real life call from the task scope. The next two sections describe the construction of the client call and server response more specific. The most important section of this document specify all XML-RPC server scopes with their sub-commands in detail. The focus of this paper lies on the task scope. Afterwards the toolbox section introduce programs and hints for testing and debugging of XML-RPC clients. The last section is a summary of already developed administration tools. Those Free Open Source Software projects are provided there as a starting point for gathering more practical informations. Thanks to Jean-Charles Siegel and Jochen Breuer for a contribution in this section.

2 Parts of **CipUX** XML-RPC server

The CipUX-RPC-3.4.0.7.tar.gz package is a standard CPAN package. The latest package can be downloaded from http://release.cipux.org. This overview will not explain how a CPAN package works, but what this package provides and installs in the file system. It contains basically two approaches (A) + (B) which might (in the future) merge into one. (A) the **CipUX** XML-RPC sever and (B) security wrapper for stunnel4. Additional it contains (C) some utility tools. You need at least (A) to have a plain **CipUX** XML-RPC server. If you also install (B) you can use a secure connection over the stunnel4 wrapper. And finally if you also install and use (C) you can test some aspects of the XML-RPC server and use cipux_rpc_list as a convenient method to list all XML-RPC calls and task scope sub-commands. Scripts in (D) are provided in the source code upstream tar release for the purpose of study. Some of those scripts are referenced, explained or even included in this document.

```
(A)
/usr/share/perl5/CipUX/RPC.pm
/usr/share/perl5/CipUX/RPC/Server.pm
/usr/share/perl5/CipUX/RPC/Server/Daemon.pm
/usr/sbin/cipux_rpcd
/etc/init.d/cipux-rpcd
/usr/share/cipux/etc/cipux-rpc.ini
```

```
(B)
/etc/cipux/stunnel/readme.txt
/etc/cipux/stunnel-cert.conf
/etc/cipux/stunnel.conf
/etc/init.d/cipux-rpcdr
/usr/sbin/cipux_mkcertkey
/usr/sbin/cipux_rpcdr
```

```
(C)
/usr/bin/cipux_rpc_list
/usr/share/perl5/CipUX/RPC/Test/Client.pm
/usr/sbin/cipux_rpc_test_client
/usr/bin/cipux_rpc_test_repetition
```

```
(D)
doc/example/bin/expl_rpc_ping
doc/example/bin/expl_rpc_session
doc/example/bin/expl_rpc_task_create_destroy
doc/example/bin/expl_rpc_task_list
doc/example/bin/expl_rpc_task_member
doc/example/bin/expl_rpc_task_sum
```

3 Configuration of the server

The **`CipUX`** XML-RPC server is a Perl script called `cipux_rpcd` with its attached modules `CipUX::RPC*`. The server script can be found (depending on your installation) for example under `/usr/sbin/cipux_rpcd`. The default server configuration is for example at `/usr/share/cipux/etc/cipux-rpc.ini`. This default configuration can be overwritten by other files. This document will not explain the configuration space here. Just to give an idea, that it is not enough to parse a single file, overwriting configuration can be found for example in: `/etc/cipux/cipux-rpc.ini` or `~/.cipux/cipux-rpc.conf`.

Valid configuration directives - which are essential to know for programming XML-RPC clients - in this file are:

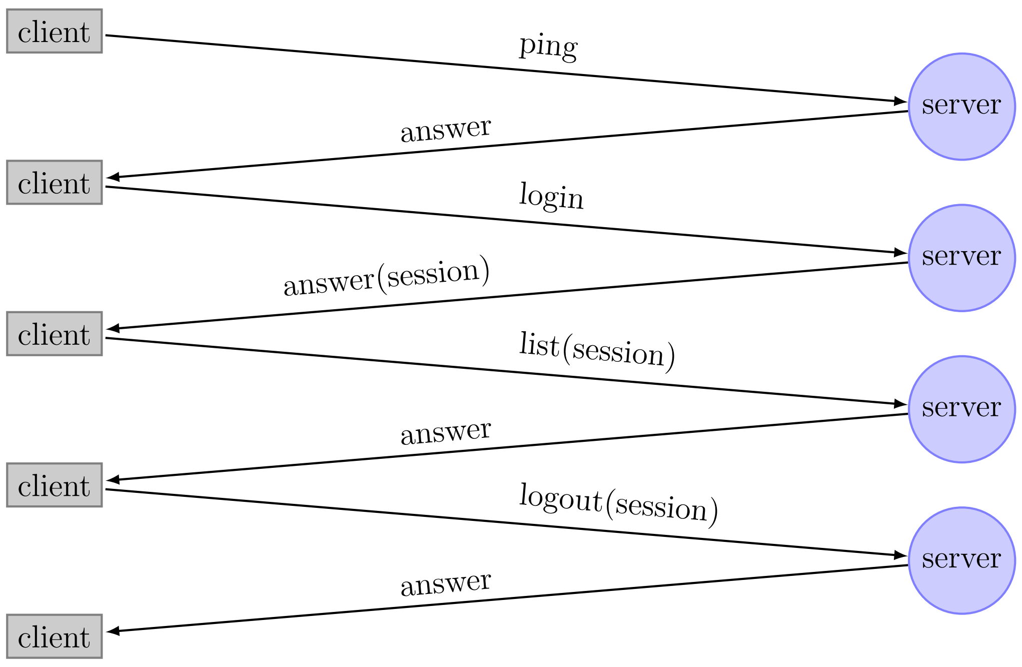
```
xml_rpc_port = 8001
xml_rpc_address = localhost
xml_rpc_proto = tcp
intern_admin_group = admin
```

Some of this default values are changeable today, some are planned to made changeable in the future. This set of values describes the plain CipUX XML-RPC server on port 8001. It is fine to use this port in socket mode on localhost. However you should not use this port for remote communication over network. The SSL encryption is handled by the stunnel4 wrapper for now, but might be integrated into the default server. If you plan to communicate over the network use the port defined in the stunnel4 configuration for CipUX. Per default this port is 8000.

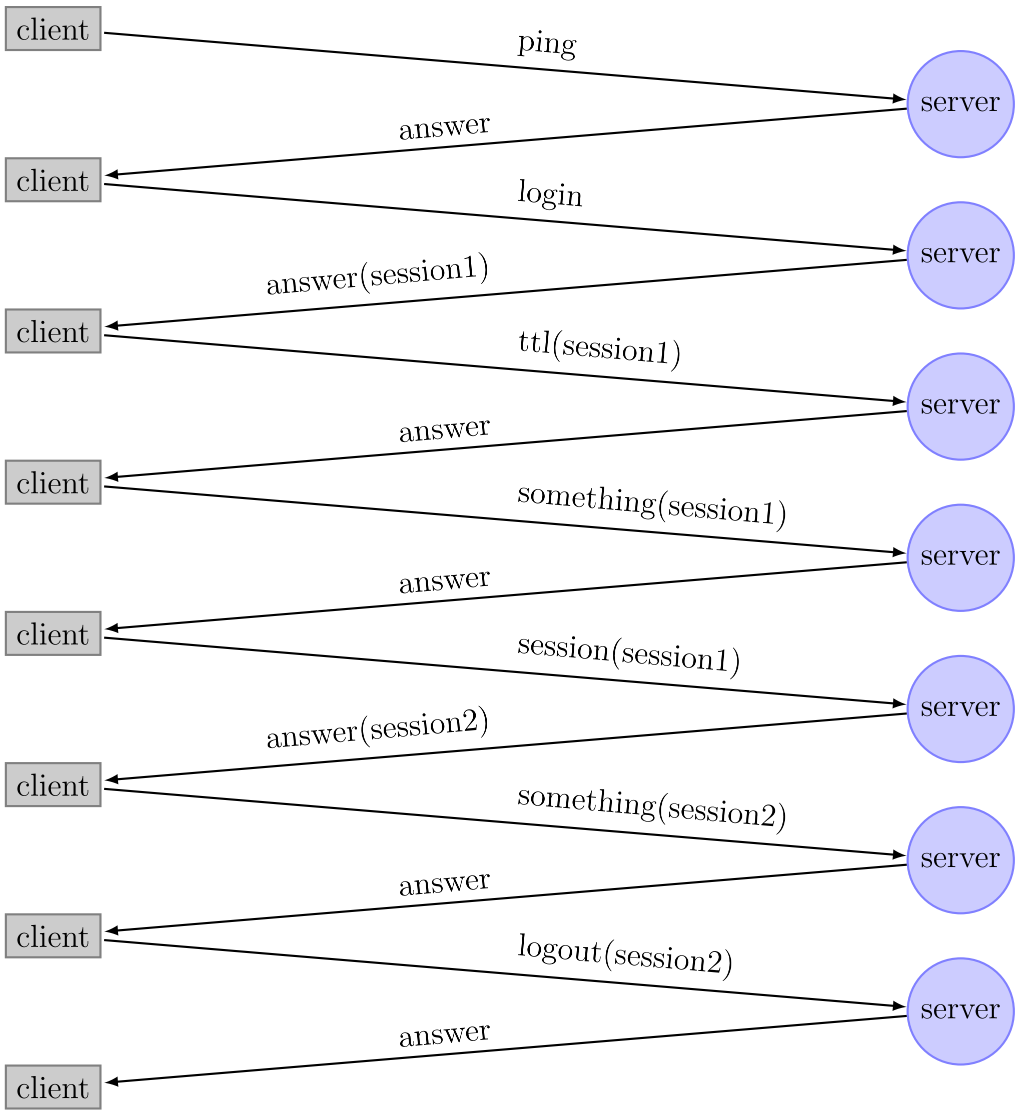
The **CipUX** XML-RPC server in version up to 3.4.0.7 do handle *http* requests but not *https* requests. To make the connection secure you have to use a different software. One possibility is to use stunnel4 which is the default solution for secure connections to the **CipUX** XML-RPC server so far. There is not much to say about this software. The Perl script `cipux_rpcdr` is a wrapper to stunnel4 with its start script `/etc/init.d/cipux-rpcdr` and a separate configuration file `stunnel.conf` and certificates. The default is to accept https requests on port 8000. For more information please refer to the stunnel4 documentation.

4 Communication via session

The communication with the XML-RPC server is straight forward. The following describes a simplified session.



If you run longer sessions with your clients - which is not advisable for web application through their fragile nature (lot of timeout traps) - you should take care of the session key. Here is an example of communication which renews the session key. The ttl call is not mandatory and can be used to examine how long as session can live.



5 Categories of RPC calls

The above calls - like `list` - are of course simplified. The list call will be introduced in detail later. To answer the question which calls are needed, one should know that they can be categorized as follows:

❶ simple calls (mostly for testing): `sum`

❷ availability calls: `ping`

❸ authentication calls: `login, logout`

❹ session management calls: `session, ttl`

❺ task calls with sub-commands: call: `task` (sub-commands: `cipux_task_*`)

If you write a CipUX RPC client, you need in most situations only the calls from ❷, ❸, ❹ and ❺. Occationally in the development phase you might also need ❶. Especially ❶ might be a good starting point, if you are new in the RPC development.

6 Example call and answer

Before more details are presented and explained this section will just provide two examples `sum` and `task` of a client call and the server response with discrete values for the impatient. The examples are taken from the `cipux_rpc_test_client` script.

6.1 Sum call example

Here is the client code for `'sum'` call with its `sum` command. A simple call and command that do not need a ticket but it need two parameters:

Discrete sum call example

```
    my $https_url = "https://localhost:8001/RPC2";
    my $server = Frontier::Client->new( url => $https_url );
    my $pay_hr = { # hash ref of payload
        header_hr => { # header is part 1 of payload
             cipux_version  => '3.4.0.0',
             client_name    => 'cipux_rpc_test_client',
             client_version => '3.4.0.0',
             rpc_version    => '2.0',
             client_key     => 'dummy',
             client_cred    => 'dummy',
             gmt_time       => '1195349030',
        },
        # some important key-value pairs
        login        => 'dummy',
        ticket       => 'dummy',
        cmd          => 'sum',
        param_hr   => { # second part of payload: parameter
            'summand2' => '4',
            'summand1' => '3',
        },
    };
    my $answer_hr = $server->call( 'sum' ), $pay_hr );
```

And here is the value of `$answer_hr` which comes from the server displayed in the Data::Dumper format:

```
$answer_hr = {
        'header_hr' => {
            'cipux_version' => '3.4.0.0',
            'server_key' => '',
            'server_cred' => '',
            'gmt_time' => '1195349220',
            'server_version' => '3.4.0.0',
            'server_name' => 'cipux_rpcd',
            'rpc_version' => '2.0'
         },
         'cmd' => 'sum',
         'ticket' => 'dummy',
         'login' => 'dummy',
         'status' => 'TRUE',
         'type'   => 'aref'
         'cmdres_r' =>  [ '7' ],
    };
```

6.2 Task call example

The task call sub-command `cipux_task_list_student_accounts` is a real world CipUX::Task XML-RPC example. It is is also a good example for a simple parameterless call. One part, which is also different from the last call is, that a login and ticket is required.

Discrete list call example

```perl
    my $https_url = "https://localhost:8001/RPC2";
    my $server = Frontier::Client->new( url => $https_url );
    my $pay_hr = {
        header_hr => {
             cipux_version  => '3.4.0.0',
             client_name    => 'cipux_rpc_test_client',
             client_version => '3.4.0.0',
             rpc_version    => '2.0',
             client_key     => 'dummy',
             client_cred    => 'dummy',
             gmt_time       => '1195349030',

        },
        login        => 'cipadmin',
        ticket       => '48df73accd8530af69f97cf1c847f29e',
        cmd          =>'cipux_task_list_student_accounts',
        param_hr   => { },
    };
    my $answer_hr = $server->call( 'task', $pay_hr);
```

Of course this was a static `$pay_hr` call. And the ticket is not valid any more. Therefore this example shows *what* data is transfered or programed not *how* it should be transfered. Hard coded tickets are not at all a solution.

And here is the value of $answer_hr again in Data::Dumper format:

```
$answer_hr = {
          'msg' => '',
          'problem' => '0',
          'ltarget' => 'memberUid',
          'cmd' => 'cipux_task_list_student_accounts',
          'cmdres_r' => {
```

```
                           'students' => {
                                          'cn' => [
                                                    'students'
                                                  ],
                                          'memberUid' => [
                                                           'bilbo',
                                                           'frodo',
                                                           'mytest',
                                                         ]
                                        }
                         },
          'status' => 'TRUE',
          'header_hr' => {
                           'server_cred' => '',
                           'server_key' => '',
                           'cipux_version' => '3.4.0.0',
                           'gmt_time' => '1260715448',
                           'server_version' => '3.4.0.0',
                           'rpc_version' => '2.0',
                           'server_name' => 'cipux_rpcd'
                         },
          'type' => 'HASH',
          'ticket' => 'dc43ee1170d9514ad7f762f561b4382b',
          'login' => 'cipadmin'
        };
```

7 Construction of the client call

In general a XML-RPC call can be made by a simple evocation:

```
my $answer_hr = $server->call('RPC scope',$pay_hr);
```

This line is made from different parts. I. The `$answer_hr` which contains the server response explained in section 8 on page 13. II. The `$server` is a Perl object. See the example client in section 9.1 on page 16 how to get that. III. The Frontier::Client key word ***`call`***, which is a subroutine in Frontier::Client. You probably do not need to look that up. IV. The last part represents the parameters to ***`call`***. Every call contains two parameters first the `'RPC scope'` name (ping, login, task, ...) and second a hash reference `$pay_hr` to the payload.

7.1 RPC scope

The RPC scope is the name of the subroutine (***sub***) of CipUX::RPC Perl module. The following table shows different RPC scopes and their constrains.

RPC scope	require login	ticket check	ticket renew
ping	no	no	no
version	no	no	no
sum	no	no	no
login	yes	yes	no
logout	yes	yes	no
session	yes	yes	yes
ttl	yes	yes	no
task	yes (*)	yes (*)	no

(*) Only for the task `cipux_task_sum` the value is ”no”.

7.2 Payload

The payload can be assigned through a reference to a hash. This reference are constructed out of 5 mandatory parts (2-6):

General payload hash

```
$pay_hr = {
    header_hr => $HEADER_HR,
    login     => $login,
    ticket    => $ticket,
    cmd       => 'sub-command name',
    param_hr  => $param_hr,
};
```

header_hr The key `header_hr` demands a reference to a hash as its value: the so called ”header”. The header as of protocol version 2.0 remains static during the communication, the count of keys are fixed. The values are also more or less fixed. The `gmt_time` value received from the server however can change. The server uses only the `client_name`. Therefore the client should provide and use a registered name. A not registered client might be rejected. Since there is no prove of validity of a client name, the server do not trust a client just because it is registered. Therefore rejecting or not rejecting a registered client is not a matter of security it is just a matter of convenience for the client developer. This behavior might change in the future, but it is foreseen that this has to go along with using other (not jet used) header fields, like `client_key` and `client_cred`.

Since the header remains basically the same, this document will not print the header over and over again. It will use the following header taken from `cipux_rpc_test_client` whenever the hash reference `$HEADER_HR` occur.

General header hash

```
$HEADER_HR = {
     cipux_version  => '3.4.0.0',
     client_name    => '/usr/bin/cipux_rpc_test_client',
     client_version => '3.4.0.0', # can be choosen
     rpc_version    => '2.0',
     client_key     => $dummy,  # $dummy not used
     client_cred    => $dummy,
     gmt_time       => $epoche,
};
```

login is the user id (uid) of the logged in user. You may not use an empty string. Also keep in mind that this do not refer to numerical user id (uidNumber).

ticket is the valid session ticked of the logged-in user. You may not use an empty string, But some calls need a sting. So if the call do not need a ticket "dummy" or "test" is a good choice.

cmd name of the sub-command. Some RPC scopes do have only one command. Like the RPC scope "`ping`" which will have the `cmd` "`ping`". Other RPC scopes like "`task`" to have several hundreds of `cmd`s. You may not use an empty string here.

param_hr is a reference to a hash. It can be a empty reference to a hash but not a reference to an array. The expected content depends on the `cmd` value. And this is related to the call subroutine of CipUX::RPC and secondly to the task subroutine of CipUX::Task. Later parts of this document will provide more hints on how to determine which parameter keys are possible.

8 The server response

This section describes what will be send back from the server and how to interpret this. The client might have implemented this call to the server.

```
my $answer_hr = $server->call('RPC scope',$pay_hr);
```

In Perl the answer `$answer_hr` of the server is a reference to a hash and is made out of seven parts.

```
$answer_hr = {
     header_hr => $server_header_hr,     # 1
     login     => $login,                # 2
     ticket    => $ticket,               # 3
     cmd       => 'sub-command name',    # 4
     status    => 'TRUE|FALSE',          # 5
     type      => 'href|aref|string',    # 6
     cmdres_r  => $cmdres_r,             # 7
};
```

header_hr is a key in the payload. The value of this key is a hash reference to a "header", which (again) remains more or less the same during a session.

```
$server_header_hr = {
   cipux_version  => '3.4.0.0',
   server_name    => 'cipux_rpcd', # server name
   server_version => '3.4.0.0',    # not fixed
   rpc_version    => '2.0',
   server_key     => $dummy,       # reserved
   server_cred    => $dummy,       # reserved
   gmt_time       => time(),       # server time
};
```

See the following subsection for details about the header keys. This answer-header is basically the same as the header which is used in the client call in section 7.2.

cipux_version is a key with a scalar value. The **CipUX** version is given from the server. However only the first 3 digits are significant. The forth digit will probably not change.

server_name contains a scalar value with the name of the server. Its value is constant and as long nobody else develops a different server it will `cipux_rpcd`.

server_version has a scalar value of the server version. All four digits of this response are significant. The server will increase this version, if the server was updated and restarted.

rpc_version represents a scalar value of the **CipUX** XML-RPC protocol version. This is the version number of the RPC calls. The number is unlikely to be changed within **CipUX** 3.4.0.y but might change if needed in **CipUX** 3.4.x.y. Your client should only accept connections up to a discrete version number.

server_key Not used now.

server_cred Not used now.

gmt_time The time is set by the server.

login is the uid of the logged in user

ticket is the more or less valid ticked of the logged in user, which was used by the user request. This filed will never provide a new or renewed ticket. you have to use the session call for this. See section 9.8 on page 22.

cmd name of the sub-command

status is key with a boolean scalar value. The status can be `TRUE` or `FALSE`.

type returned data type: href, aref, string

cmdres_r is key which name derived from command (`cmd`) result (`res`) reference (`_r`). The value is a reference to an array or hash which is constructed like this

```
$cmdres_r = ['a','b','c']
```

or

```
$cmdres_r = { ttl=> 20 }
```

ltarget is a key which can be used to automatically parse the output of the `cmdres_r` key value. The value of `ltarget` *can* be a LDAP attribute like uidNumber if the target of the task scope sub-command is only one attribute. This feature is considered to be experimental.

At the moment some calls return other undocumented fields and there might be more fields added in the future. The fields which are documented here are considered to be a part of the version 2.0 of the protocol. You should use them. You could also use the undocumented fields. However they might vanish or change with out warning. If you are missing some fields or if you want that a field will be officially, you can join cipux-devel and make a existing field stable or develop a new field, which can be introduced in the next protocol version 2.2.

9 Calls and responses in detail

The CipUX::RPC call "ping" and CipUX::RPC "sum" can be used to test the RPC server without authentication.

9.1 ping

call:	ping
cmd:	ping
login parameter:	no
ticket parameter:	no
param_hr:	empty
cmdres_r type:	hash reference
cmdres_r value:	empty

The aim of this function is to see if the server is up. The return value will always be the status "`TRUE`". No value in `$cmdres_r`. So if you are getting a status other then "`TRUE`" the server is up but has a problem. If you get no answer the server is not up. Sorry due to the nature of logic it was not possible to implement an answer of "`DOWN`" from the server if the server is down.

```
$status => 'TRUE',
$cmdres_r = { }
```

This short program tests if the local server is up and running or not. However it does not tell you if the stunnel4 wrapper is working or not.

expl_rpc_ping

```perl
#!/usr/bin/perl -w
use strict;
use English qw( -no_match_vars);
use Frontier::Client;

my $header_hr = {
    cipux_version  => '3.4.0.0',
    client_name    => 'expl_rpc_ping',
    client_version => '0.1',
    rpc_version    => '2.0',
    client_key     => '',
    client_cred    => '',
    gmt_time       => time,
};

my $pay_hr = {
    header_hr => $header_hr,
    login     => 'dummy',
    ticket    => 'dummy',
    cmd       => 'ping',
    param_hr  => {},
};

my $http_url   = "http://localhost:8001/RPC2";
my $server     = Frontier::Client->new( url => $http_url );
my $answer_hr = {};
eval { $answer_hr = $server->call( 'ping', $pay_hr ); };

if ($EVAL_ERROR) {
    print "Server down\n";
}
elsif ( $answer_hr->{status} eq 'TRUE' ) {
    print "Sever up\n";
}
else {
    print "Server problem\n";
}
```

Example Answer:

```
$answer_hr = {
          'msg' => '',
          'ltarget' => 'NULL',
          'cmd' => 'ping',
          'cmdres_r' => {},
          'status' => 'TRUE',
          'header_hr' => {
                         'server_cred' => '',
                         'server_key' => '',
                         'cipux_version' => '3.4.0.0',
                         'gmt_time' => '1260436207',
                         'server_version' => '3.4.0.0',
                         'rpc_version' => '2.0',
                         'server_name' => 'cipux_rpcd'
                       },
          'type' => 'href',
          'ticket' => 'dummy',
          'login' => 'dummy'

}
```

9.2 version

call:	version
cmd:	version
login parameter:	no
ticket parameter:	no
param_hr:	empty
cmdres_r type:	hashref
cmdres_r value:	cipux_version, server_version, rpc_version

The aim of the call is to test the version of the CipUX::RPC server (without logging in) to be able do decide if a login might be possible or not. An other thing is that you can test with this function, if you are able to parse the returned hash reference. The relevant part of the answer looks as follows.

```
'cmdres_r' => {
                         'cipux_version' => '3.4.0.0',
                         'server_version' => '3.4.0.0',
                         'rpc_version' => '2.0'
                       },
```

9.3 sum

call:	sum
cmd:	sum
login parameter:	no
ticket parameter:	no
param_hr:	summand1, summand2
cmdres_r type:	array reference
cmdres_r value:	a scalar value

The CipUX::RPC server provides two sum function: The first is CipUX::RPC `sum` and the second is and CipUX::Task `cipux_task_sum`. This section is about the simple CipUX::RPC sum function. The aim of this function is to test, if you can send arguments via hash reference and if you can parse the returned array reference.

```
    param_hr        => {
                           summand1  => 3,
                           summand2  => 4,
                           },
```

The call will return an array reference with a scalar value. See section 6.1 on page 7 for a full example.

```
'cmdres_r' => [
                 '7'
],
```

9.4 cipux_task_sum

call:	task
cmd:	cipux_task_sum
login parameter:	no
ticket parameter:	no
param_hr:	summand1, summand2
cmdres_r type:	array reference
cmdres_r value:	a scalar value

The `cipux_task_sum` is similar to the `sum` call. Except that it is internally invoked in the task section of the XML-RPC server. You have already noticed the difference between the `cmd` and ***`call`*** section above. Therefore rather then testing your client this call can be used to test the rpc server. It is summarized here for the sake of completeness.

The call will return an array reference with a scalar value. See section 6.1 on page 7 for a similar example with the `'sum'` call with its `sum` sub command.

9.5 login

call:	login
cmd:	login
login parameter:	yes
ticket parameter:	no
param_hr:	password
cmdres_r type:	hash reference
cmdres_r value:	ttl, login, ticket

The login call request 2 parameters. One parameter is given as payload and one parameter will be supplied inside `param_hr`.

login call payload

```
$pay_hr = {
  param_hr   => { password => '**********' },
  cmd        => 'login',
  header_hr => {
    cipux_version  => '3.4.0.0',
    client_key     => '',
    client_cred    => '',
    gmt_time       => '1196466596',
    client_name    => 'cipux_rpc_test_client',
    client_version => '3.4.0.0',
    rpc_version    => '2.0'
  },
  ticket => 'dummy',
  login  => 'cipadmin'
};
```

An positive answer should contain:

```
$answer_hr = {
  'msg'       => '',
  'cmd'       => 'login',
  'cmdres_r'  => {
    'ttl'     => '20',
    'ticket'  => '1559cc7c463af4a5a28586e931fbf744',
    'login'   => 'cipadmin'
  },
  'status'     => 'TRUE',
  'header_hr'  => {
    'cipux_version'   => '3.4.0.0',
    'server_key'      => '',
    'server_cred'     => '',
    'gmt_time'        => '1196466106',
    'server_version'  => '3.4.0.0',
    'server_name'     => 'cipux_rpcd',
    'rpc_version'     => '2.0'
  },
  'type'    => 'href',
  'ticket'  => 'dummy',
  'login'   => 'cipadmin'
};
```

Be aware that you have to grab the cmdres_r ticket not the payload ticket! The example script expl_rpc_session contains a login call.

9.6 logout

call:	logout
cmd:	logout
login parameter:	yes
ticket parameter:	yes
param_hr:	empty
cmdres_r type:	hash reference
cmdres_r value:	empty

There is nothing much to tell about the logout call. If it is successfully issued it returns “TRUE” as its status back. The example script expl_rpc_session contains a logout call.

```
'status' => 'TRUE',
```

9.7 ttl

call:	ttl
cmd:	ttl
login parameter:	yes
ticket parameter:	yes
param_hr:	empty
cmdres_r type:	hash reference
cmdres_r value:	ttl

The relevant part of the answer:

```
'cmdres_r'  => {'ttl' => '20'},
```

See section 9.9.7 on page 37 for an example.

9.8 session

call:	session
cmd:	session
login parameter:	yes
ticket parameter:	yes
param_hr:	empty
cmdres_r type:	hash reference
cmdres_r value:	ticket or empty

The session call can be used in two ways with the same syntax. First it can be used to check if the session was still valid. Second the call gives you a new ticket in the `cmdres_r`, so you can use the call to extent the session. The best is to uses both ways together. If the session is not valid, you should examine the `msg` payload field for more information. In that case `cmdres_r` is empty.

```
$answer_hr = \{
  'msg' => 'The ticket check was not successful.
This could have several reasons: a timeout, logout,
... Please log in again. is_ticket_bad: (ticket is
 bad: time login mismatch)',
  'cmd'        => 'session',
  'cmdres_r'   => {},
  'status'     => 'FALSE',
  'header_hr'  => {
    'cipux_version'   => '3.4.0.0',
    'server_key'      => '',
    'server_cred'     => '',
    'gmt_time'        => '1196466106',
    'server_version'  => '3.4.0.0',
    'server_name'     => 'cipux_rpcd',
    'rpc_version'     => '2.0'
  },
  'type'    => 'href',
  'ticket'  => '1559cc7c463af4a5a28586e931fbf744',
  'login'   => 'cipadmin'
};
```

If the session is still a valid session a new ticket will be given via the `cmdres_r` reference. In this case a hash reference with the key `ticket`.

```
'cmdres_r' => {
                  ticket => '285c9699e3f664fbfce5d04e6d0b98e0'
              },
```

You can grab this for example with this lines of Perl code.

```perl
my $new_ticket = undef;
if (      exists $answer_hr->{cmdres_r}->{ticket}
    and defined $answer_hr->{cmdres_r}->{ticket}
    and $answer_hr->{cmdres_r}->{ticket} )
{
    $new_ticket = $answer_hr->{cmdres_r}->{ticket};
}
```

The old session ticket, which you send to the server will also given back via the `ticket` key in the payload of the answer. To make the distinction clear you could grab this with this lines of Perl code.

```perl
my $old_ticket = undef;
if (      exists $answer_hr->{ticket}
    and defined $answer_hr->{ticket}
    and $answer_hr->{ticket} )
{
    $old_ticket = $answer_hr->{ticket};
}
```

Sometimes things get wrong. One real user scenario might be that the user waited too long and the session expired or might get wrong because of other reasons. In this case the answer from the server would look like this. Non important keys where omitted.

```
'msg' => 'Your ticket is not valid! This can have serveral reasons: (1) the
ticket expired. (2) an error in programming (3) you never logged in',
'cmdres_r' => {},
'status' => 'FALSE',
'ticket' => 'test',
'login' => 'test'
```

So you can expect to get a status of FALSE and an empty cmdres_r. The msg field might give a hint what is wrong. For now it refers to 3 problems. This might changed in the future to be more precise. This message might also be translated into other languages. For an example see the script expl_rpc_session, it contains a session call.

9.9 task

call:	task
cmd:	name of CipUX::Task command
login parameter:	yes
ticket parameter:	yes
param_hr:	depends
cmdres_r type:	hash reference
cmdres_r value:	depends

The RPC scope "task" is the most used part of the **CipUX** XML-RPC server calls. Because the task scope has a lot of sub-commands we will start with a simple example. We assume that you have some users in the role "student" or "students" on your system. To list those you have to use the sub-command cipux_task_list_student_accounts. Just to show you the expected output you can use the CipUX::Task layer directly as root.

```
~$ cipux_task_client -t cipux_task_list_student_accounts
```

If you have students on you system you might get a similar output like this:

```
students        bilbo    frodo    mytest
```

Here "students" is the role and the members of this role are: "bilbo", "frodo" and "mytest". Section 6.2 on page 9 contains a full example what parameter are used and what the expected output looks like if you code this by hand.

If you re-program this call by hand it can get longly homework. Programming ping, login, task and then logout will take a long time. To make this shorter in Perl you can use the helper module `CipUX::RPC::Client`. If you use an other language then Perl have a look at the source code and comments of the `sub extract_data_for_tpl { ... }` subroutine of `CipUX::RPC::Client`. This was originally written for **CAT** to give it a helper routine to parse the output of an XML-RPC answer. However you probably should implement a similar or even better routine in your application. Here comes a full fledged `CipUX::RPC::Client` example with the XML-RPC scopes ping, login, task (`cipux_task_list_student_accounts`) and of course logout.

expl_rpc_task_list

```
#!/usr/bin/perl -w
use strict;
use CipUX::RPC::Client;
use English qw( -no_match_vars);

# prep
my $rpc = CipUX::RPC::Client->new(
    {
        url     => 'http://localhost:8001/RPC2',
        client  => 'expl_rpc_task_list',
        version => '0.0.1',
    }
);

# start calling
eval { $rpc->rpc_ping; };
die "Server is down! $EVAL_ERROR" if $EVAL_ERROR;
my $ok = $rpc->rpc_login;
if ($ok) {
    my $cmd  = 'cipux_task_list_student_accounts';
    my $a_hr = $rpc->xmlrpc( { cmd => $cmd } );
    my $d_hr = $rpc->extract_data_for_tpl(
        { answer_hr => $a_hr, use_ltarget => 1 } );
    print "Students on the system:\n";
    foreach ( @{ $d_hr->{tpl_data_ar} } ) {
        print "\t$_->{$d_hr->{ltarget}}\n";
    }
    $rpc->rpc_logout;
}
```

```perl
else {
    print 'No access for ' . $rpc->get_login . "!\n";
    print "Wrong password?\n";
}
```

If you use this sample program you will get output like this.

```
expl_rpc_task_list$ Enter login: cipadmin
expl_rpc_task_list$ Enter password:
Students on the system:
        bilbo
        frodo
        mytest
```

One short word on the server response. The server response depends on the sub command and the data structure which come from the storage layer. This is the data which the `cipux_task_list_student_accounts` sub-command returns. The most relevant parts of this answer are `'ltarget'` and `'cmdres_r'`

```
$answer_hr = {
          'msg' => '',
          'ltarget' => 'memberUid',
          'cmdres_r' => {
                        'students' => {
                                      'cn' => [
                                                'students'
                                              ],
                                      'memberUid' => [
                                                       'bilbo',
                                                       'frodo',
                                                       'mytest',
                                                     ]
                                    }
                      },
          'cmd' => 'cipux_task_list_student_accounts',
          'status' => 'TRUE',
          'login' => 'cipadmin',
          'problem' => 0,
          'header_hr' => {
                         'cipux_version' => '3.4.0.0',
                         'server_key' => '',
                           'server_cred' => '',
                           'gmt_time' => 1260715448,
                           'server_version' => '3.4.0.0',
                           'server_name' => 'cipux_rpcd',
                           'rpc_version' => '2.0'
                         },
          'type' => 'HASH',
          'ticket' => '240bf8a03ed86a8f5b762f9cccdce73f'
        };
```

The expression

```
my $d_hr = $rpc->extract_data_for_tpl( { answer_hr => $a_hr } );
```

would convert this to the following quite similar output.

```
$d_hr ={
          'tpl_data_ar' => [
                              {
                                'cn' => 'students',
                                'memberUid' => 'bilbo, frodo, mytest'
                              }
                            ]
        };
```

You can use this directly. And the key `ltarget` with its value `'memberUid'` will be provided by the server response. The trick the former mentioned subroutine does is parse every output and transform this to a unified format. The additionally parameter `use_ltarget`

```
my $d_hr = $rpc->extract_data_for_tpl( {
    answer_hr   => $a_hr,
    use_ltarget => 1,
} );
```

will produce a very different data structure.

```
$d_hr = {
          'ltarget' => 'memberUid',
           'tpl_data_ar' => [
                                {
                                  'memberUid' => 'bilbo'
                                },
                                {
                                  'memberUid' => 'frodo'
                                },
                                {
                                  'memberUid' => 'mytest'
                                },
                              ]
         };
```

This look somewhat bigger, but is automatically processable.

9.9.1 Taxonomy

This section deals with logic behind task scope sub-command names. You can derive the mandatory parameter(s) from this logic.

sub-command schematic	param_hr	return scope
∗_add_(M)_to_(G)	object=(G), value=(M)	
∗_change_(X)_(L)	object=(X), value=(L)	
∗_create_(X)	object=(X) [1]	
∗_deregister_(X)	object=(X)	
∗_destroy_(X)	object=(X)	
∗_disable_(X)	object=(X)	
∗_enable_(X)	object=(X)	
∗_list_(A)s		all objects
∗_list_(M)_of_(G)	object=(X)	all (M)
∗_obtain_(X)_(L)	object=(X)	one attribute
∗_register_(A)	object=(X)	
∗_remove_(M)_from_(G)	object=(G), value=(M)	
∗_retrieve_all_(A)_(L1)_..._(Ln)	object=(X)	all attributes
∗_search_all_(L)		all attributes
∗_sum	[2]	one sum

`*: cipux_task`

```
[1]: require sometimes some other parameter depending on the object
     require no additional parameter for *_account or *_share
[2]: only for testing
(A): object is coded in task name
(G): group object, line *_account or *_share
(L): LDAP attribute codes as English name (memberUid -> member)
(M): member, client
(Y): member, client (variable)
(X): ID, which is a variable ID to the object name (variable)
```

9.9.2 Object call parameter

The `'object'` parameter is an abstract `$param_hr` hash key to define the name of the thing the sub-command is operating on. As you learn in the RPC `task` scope sub-command taxonomy, not every sub-command do need an object, but some do. If the sub-command needs an object it can be provided like this:

Object parameter

```perl
my $param_hr = {
         object  => $some_object_scalar,
};
```

The example script `expl_rpc_task_create_destroy` plays with this object parameter. It creates a student account and delete it thereafter.

expl_rpc_task_create_destroy

```perl
#!/usr/bin/perl -w
use strict;
use CipUX::RPC::Client;
use English qw( -no_match_vars);

# prep
my $rpc = CipUX::RPC::Client->new(
    {
        url     => 'http://localhost:8001/RPC2',
        client  => 'cipux_rpc_task_create_destroy_test',
        version => '0.0.1',
    }
);
my $s = {}; my $id = shift;
```

```perl

# start calling
eval { $rpc->rpc_ping; };
die "Server is down! $EVAL_ERROR" if $EVAL_ERROR;
my $ok = $rpc->rpc_login;
if ($ok) {
    list_students();
    create_student($id) if not exists $s->{"\t$id"};
    list_students();
    destroy_student($id) if exists $s->{"\t$id"};
    list_students();
    $rpc->rpc_logout;
}
else {
    print 'No access for ' . $rpc->get_login . "!\n";
    print "Wrong password?\n";
}

sub list_students {
    my $cmd  = 'cipux_task_list_student_accounts';
    my $a_hr = $rpc->xmlrpc( { cmd => $cmd } );
    my $d_hr = $rpc->extract_data_for_tpl(
        { answer_hr => $a_hr, use_ltarget => 1 } );
    $s = {};
    foreach ( @{ $d_hr->{tpl_data_ar} } ) {
        $s->{"\t".$_->{$d_hr->{ltarget}}}=1;
    }
    print "Students on the system:",sort keys %{$s},"\n";
}

sub create_student {
    my $s    = shift;
    my $cmd  = 'cipux_task_create_student_account';
    my $p_hr = { object => $s, };
    my $a_hr = $rpc->xmlrpc({cmd=>$cmd, param_hr=>$p_hr});
}

sub destroy_student {
    my $s    = shift;
    my $cmd  = 'cipux_task_destroy_student_account';
    my $p_hr = { object => $s, };
    my $a_hr = $rpc->xmlrpc({cmd=>$cmd, param_hr=>$p_hr});
}
```

The script requires a parameter, the name of the account which should be created. In this case `mytestobj1` was provided as an object.

```
~$ perl expl_rpc_task_create_destroy mytestobj1
expl_rpc_create_destroy$ Enter login: cipadmin
expl_rpc_create_destroy$ Enter password:
Students on the system: bilbo   frodo   mytest
Students on the system: bilbo   frodo   mytest  mytestobj1
Students on the system: bilbo   frodo   mytest
```

9.9.3 Value call parameter

The `'value'` parameter is an abstract `$param_hr` hash key to define the data for a given `'object'`. Like `--object frbeutlin` or `-o frbeutlin` and `-x value=Frodo` giving on the command line the value of `-x` are most likely referring to the *first name* of the account `frbeutlin`. In this terms the meaning of the abstract `value` is determined by the sub-command name. As you learned in the RPC scope `task` taxonomy about sub-commands, not every sub command do need a `value`, but some do. If the sub-command needs a `value` it can be provided like this on the command line:

```
~$ cipux_task_client -t cipux_task_add_member_to_role_account \
                          -o tutor \
                          -x value=frbeutlin
```

Keep in mind that the role *tutor* might be different on your system. The same sub-command looks with its parameters in code like:

Abstract value parameter hash

```
        'param_hr' => {
                        'object' => 'tutor'
                        'value'  => 'frbeutlin',
                        },
```

9.9.4 Other call parameters

In most cases there are performed additional calculations on the abstract `object` and `value` parameters. Therefore it is the best practice to provide those parameters as abstract hash key. This was introduced in section 9.9.2 and 9.9.3 at page 29 and 31.

It is of course possible to provide other parameters too, not just `object` and `value`.

Create call with other parameters

```
my $pay_hr = {
  header_hr => $header_hr,
  login     => $login,
  ticket    => $ticket,
  cmd       => 'cipux_task_create_student',
  param_hr  => {
    object => 'bibeutin',
    cipuxFirstname  => 'Bilbo',                  # cipux.schema
    cipuxLastname   => 'Beutlin',                # cipux.schema
    userPassword    => $new_password,            # core.schema
    homeDirectory   => '/home/cipux0/bibeutlin', # nis.schema
  },
};
```

However, this example would work only on systems where those LDAP attributes are defined. In this example `cipuxFirstname` and `cipuxLastname` is defined in the cipux.schema. If you use those LDAP attributes directly you make your application dependent on the existence and usage of those schemata. If possible try to avoid the direct use of LDAP attributes. Unfortunately **CipUX** is not providing more abstract attributes. It would be a nice feature to have a dispatch list for this in some future version to avoid using LDAP attributes directly. Foreseen attributes might be `firstname`, `lastname`, `mailaddress`, ... But there is limited solution for this problems in the current version, which will be introduced now.

The above problem could be avoided if you call 4 task scope sub-commands which gives you the desired level of abstraction:

① `cipux_task_create_student_account`

② `change_user_account_firstname`

③ `change_user_account_lastname`

④ `change_student_account_password`

If you use this sub-commands you could avoid using discrete LDAP attributes. On other thing is that you should choose the smallest object scope as possible. So choosing the next list of commands is a little bit better, because your client would need less access rights to do it.

① `cipux_task_create_student_account`

② `change_student_account_firstname`

③ `change_student_account_lastname`

④ `change_student_account_password`

If some sub-commands are missing in in you installation or in **CipUX**, this sub-commands can be added to the configuration file `/etc/cipux/cipux-task.perl` or `/usr/share/cipux/etc/cipux-task.perl`[1] locally or better you write just your additional sub-commands in a newly created file with the ending `.perl` under `/usr/share/cipux/etc/cipux-task.d/your_name.perl`. This will be additionally read in by the **CipUX** configuration space. The best way is of course to share your new sub-commands on the mailing list cipux-devel. This will give you feedback. You might get hints on improvement or your commands will even be be included in the next CipUX-Task release.

9.9.5 More then one call parameter

It is possible to add other attributes to a call. However this version of CipUX do not track the possibility if such an attribute can be used or not. The LDAP server in the last instance will decide what is possible and what not. To get an idea what is possible it is a good idea to look at all cipux-task.perl configuration files. But also additional attributes are possible. It is planned that an introspective command of the RPC will be written to support a query, for each object, to see what is possible. For now the LDAP schemata are the reference.

[1]You can find example sub-commands in this file. You can add something or modify this. However this is the location of the CipUX::Task boostrap configuration and will be overwritten when you install a new version.

Using the `cipux_task_client` is also a quick method to find out what is possible when you add the `--debug` switch on the command line.

```
~$ cipux_task_client -t cipux_task_create_teacher_account -o boromir \
-x cipuxFirstnanme=Boromir -x cipuxLastname='Son of Denethor II.'\
 --debug
```

This is equivalent to the following `$param_hr` perl code.

```
\$param_hr = {
     object = 'boromir',
     cipuxFirstname = 'Boromir',
     cipusLastname  = 'Son of Denethor II.',
};
```

It is not possible to show the hole output here, but you will get an idea, what additional LDAP attributes are possible for the `$param_hr` keys if you read the output. This and the next outputs are slightly modified to fit on the page.

```
...
CipUX::Storage::add_node <461>: -> found obj.Class: objectClass
CipUX::Storage::add_node <470>: -> found obj.Class name: posixAccount
CipUX::Storage::add_node <486>: --> mandatory attr: uid
CipUX::Storage::add_node <486>: --> mandatory attr: gidNumber
CipUX::Storage::add_node <486>: --> mandatory attr: homeDirectory
CipUX::Storage::add_node <486>: --> mandatory attr: uidNumber
CipUX::Storage::add_node <486>: --> mandatory attr: cn
CipUX::Storage::add_node <499>: --> auxiliary attr: description
CipUX::Storage::add_node <499>: --> auxiliary attr: gecos
CipUX::Storage::add_node <499>: --> auxiliary attr: loginShell
CipUX::Storage::add_node <499>: --> auxiliary attr: userPassword
...
```

This section for example logs the output of the subroutine `add_node`. It tells in details which attributes are mandatory or optional. Keep in mind that `cipux_task_client` will throw an exception when a mandatory option is missing. This options are mandatory in terms of LDAP and sometimes not mandatory for the client to provide. So you can imagine that CipUX tries hard to automatically calculate many mandatory and auxiliary values. Some of them can be overwritten on the fly. A better approach might be to overwrite them in a second call or they should be changed globally in a configuration file if you intend to always overwrite them. This is because a automatically calculated value might be used as an input to a second automatically calculated value. So you have to try or to understand what you do and what effect do this have. If unsure changing later is the better approach.

```
2009-12-17+23:36:24 CipUX::Storage::add_node <407>: BEGIN
2009-12-17+23:36:24 CipUX::Storage::add_node <408>: > obj: boromir
2009-12-17+23:36:24 CipUX::Storage::add_node <409>: > type: cipux_account.u
2009-12-17+23:36:24 CipUX::Storage::add_node <410>: > attr_hr: $VAR1 = {
          'cipuxHardQuota' => 200000,
          'cipuxIsAccount' => 'TRUE',
          'cipuxInternetStatus' => 'accept',
          'cn' => [ 'Boromir Son of Denethor II.'                  ],
          'cipuxLastname' => [ 'Son of Denethor II.' ],
          'uidNumber' => [ 10157 ],
          'cipuxRemark' => 'CipUX task layer',
          'cipuxSkeletonUid' => [ 'none' ],
          'cipuxMail' => [ 'boromir@tjener' ],
          'cipuxStatus' => 'idle',
          'cipuxRole' => 'teachers',
          'gecos' => [  'boromir' ],
          'userPassword' => [ '{crypt}GyDkuuhqJi/rU' ],
          'cipuxSoftQuota' => 100000,
          'uid' => [ 'boromir' ],
          'homeDirectory' => ['/skole/tjener/home0/boromir'],
          'objectClass' => [ 'posixAccount',
                             'top',
                             'shadowAccount',
                             'imapUser',
                             'cipuxAccount' ],
          'gidNumber' => [ 10157 ],
          'cipuxFirstname' => 'Boromir',
          'cipuxCreationDate' => [ '2009-12-17T23:36:23' ],
          'mailMessageStore' => ['/var/lib/maildirs/boromir'],
          'loginShell' => '/bin/bash'
        };
```

If you read further in the debug output you can find hole hash trees for LDAP object like this one. But be aware of just copying those structures to your XML-RPC client, because this is only on small part of what happens if a user account is created.

9.9.6 Exceptions

If you create a user account (eg. "frbeutlin") and try to create this twice which is of course not possible you will most likely get an exception similar like this one.

```
(EXCEPTION) Can not add entry \
            [cn=frbeutlin,ou=Group,dc=skole,dc=skolelinux,dc=no]! \
            (Already exists)
```

This exception are triggered from the object layer and are given back to the client over the XML-RPC server. You have to catch those exceptions.

9.9.7 Access to a task

Every task hast its own access check when requested via RPC. If we for example code this `'ttl'` call:

Call ttl as a task

```
my $pay_hr = {
    header_hr  => $header_hr,
    login      => 'cipadmin',
    ticket     => $some_valid_ticket,
    cmd        => 'ttl',
    param_hr   => {},
};
my $answer_hr = $server->call( 'task', $pay_hr);
```

Then we will be astonished because `cipadmin` has *no right* to execute the task `'ttl'` and the answer will be harsh.

```
$answer_hr = {
          'msg'        => 'No access for [cipadmin] to [ttl]',
          'ltarget'    => 'NULL',
          'cmd'        => 'ttl',
          'cmdres_r'   => {},
          'status'     => 'FALSE',
          'header_hr'  => $sever_header_hr,
          'type'       => 'href',
            'ticket'     => 'c73ce0812fd68a2345254634dab64e24',
            'login'      => 'cipadmin'
          };
```

We learn earlier that the `'ttl'` command can be accessed by everyone. Why can the access then be denied? Well, a common mistake is to think that `'ttl'` is a CipUX task (to be correctly a task scope sub command). It is not! The "`ttl`" is a RPC scope by itself. The call which generates this "access denied" called the "`task`" scope with the cmd `'ttl'`. Instead we should call the `'ttl'` command with the right scope `ttl`.

Call ttl as a ttl

```perl
my $pay_hr = {
    header_hr  => $header_hr,
    login      => 'cipadmin',
    ticket     => $some_valid_ticket,
    cmd        => 'ttl',
    param_hr   => {},
};
my $answer_hr = $server->call( 'ttl', $pay_hr);
```

This will give us the correct answer from the server.

```
$answer_hr = {
            'msg'         => '',
            'problem'     => '0',
            'ltarget'     => 'NULL',
            'cmd'         => 'ttl',
            'cmdres_r'    => {
                                 'ttl' => '1200'
                             },
            'status'      => 'TRUE',
            'header_hr'   => $HEADER_HR,
            'type'        => 'href',
            'ticket'      => '1223457de8afce5c0bcae9092833d082',
            'login'       => 'cipadmin'
          };
```

This was an example of calling an unknown sub-command like `'ttl'` in the RPC scope `task` and getting an "access denied" as an answer. If you are calling a real `task` sub-command (for example `'cipux_task_list_student_accounts'`) you have the chance that the user has access to this. Because *cipadmin* is in the *admin* group he should have access to all `task` sub-commands. For other users this depends on the role based access control (RBAC) module which can not be explained here in detail. Basically you can assume that if the RPC client is registered and has registered its `task` sub-commands and a group is assigned to that client and a user is assigned to that group the user has access to that tasks.

10 Toolbox

This section describes some programs coming with the XML-RPC server which might be useful for testing and debugging.

10.1 cipux_rpcd

The most important tool is the server by itself. Normally started you might get some information as log messages through you system log facility. However this is limited. To get more information the sever has a debug mode. Of course you have to stop the running server for this.

```
~$ /usr/sbin/cipux_rpcd --debug
```

This will print a lot of messages on the console! Of course this is not a good way to operate that server in production mode - it slows everything down, but it helps developing a XML-RPC client or to see if something is wrong. **CipUX** uses an advanced logging facility Log::Log4perl. This was developed for Java and adopted for Perl. More information can be found at CPAN. You can find the configuration in log4perl.conf. Be aware that **CAT** uses its own file. Sometimes this will generate too much messages. You can use the Log::Log4perl configuration to switch on and off messages, filter them, redirect them. In principle you can restrict logging to certain **CipUX** Perl modules or even subroutines. After the logging configuration has changed the server has to be restarted before it takes effect.

10.2 cipux_rpc_list

This script uses the same configuration space as the XML-RPC server. Its only duty is to list all calls and all sub-commands in a long list. Even though this a good thing to have in this output calls and sub-commands are not distinguished. Therefore this is a very simple method for checking what is available in terms of shell scripts. If the server gets an introspection mechanism in the future this command should be enhanced/replaced accordingly.

```
~$ cipux_rpc_list|grep -v cipux_task
```

Since all sub-commands in the RPC scope `task` starting with `cipux_task` the trick with the `grep -v` do the trick to list all scopes.

```
ping
version
sum
login
logout
session
ttl
task
```

10.3 cipux_rcp_test_client

The test client can be used to test a locally installed **CipUX** XML-RPC server. The usage is straight forward and self explanatory. Prepare yourself with the login of `cipadmin` and the password. The test script will create and delete some test accounts.

The `cipux_rcp_test_client` runs several RPC calls and try to figure out if the answers are valid or not. It can therefore used to test if the XML-RPC behaves correctly in some ways. Because the `cipux_rcp_test_client` execute real tasks like creating and deleting users and groups on the system there is the possibility to delete accounts that should not be deleted. Even if this is not very likely it is more likely to corrupt the LDAP database in terms of data validity and because of this risk it is advised to uses this command after a LDAP backup and not on productive systems unless you can assure that the LDAP data before running and after running is the same in terms of validity. So keep in mind until someone volunteers to enhance it, this program is not bullet-proof.

```
~$ cipux_rcp_test_client
```

This will outputs hundreds of lines. The last lines might look like this ones:

```
  Test 0305: CipUX::Task 03 create netgroup exec                 SUCSESS
  Test 0306: CipUX::Task 04 create netgroup list                 SUCSESS
  Test 0307: CipUX::Task 05 create netgroup result               SUCSESS
  Test 0308: CipUX::Task 06 add client to netgroup exec          SUCSESS
  Test 0309: CipUX::Task 07 add client to netgroup list          SUCSESS
  Test 0310: CipUX::Task 08 add member to netgroup result        SUCSESS
  Test 0311: CipUX::Task 09 erase client 's of netgroup list     SUCSESS
  Test 0312: CipUX::Task 10 destroy netgroup exec                SUCSESS
  Test 0313: CipUX::Task 11 destroy netgroup list                SUCSESS
  Test 0314: CipUX::Task 12 destroy netgroup result              SUCSESS
  ----------------------------------------------------------------------
                                            SUCSESS: 314    FAILURE: 0
  ----------------------------------------------------------------------
```

An other test of the script is also very useful: testing if the LDAP database is more or less the same before and after this script was executed. To perform this test you have to be root or at least you have to have the execution rights for slapcat.

```
~$ slapcat > before
~$ cipux_rpc_test_client
~$ slapcat > after
~$ diff --ignore-matching-lines '^modifyTimestamp' \
     --ignore-matching-lines '^entryCSN' before after
```

If you have no output of the `diff` command the LDAP is more or less the same as before.

10.4 cipux_rpc_test_repetition

The `cipux_rpc_test_repetition` script can be called simply

```
~$ cipux_rpc_test_repetition --time 2000
```

It will run for the specified time in seconds. To test the default ticket life span of 1200 seconds a value larger then 1200 for example 1800 (30 minutes) should be used. For 1 minute we get this output at the end of the job.

```
======================================================================
pay ticket 13e777a207fb292f3a1e3d94d8fcedc9
default channel ticket 13e777a207fb292f3a1e3d94d8fcedc9
cipux_task_list_user_accounts at 1260214026 status TRUE \
(13e777a207fb292f3a1e3d94d8fcedc9)
pay ticket 13e777a207fb292f3a1e3d94d8fcedc9
default channel ticket adf89d46c9e167b5758adbaf405110ac
explicit channel ticket adf89d46c9e167b5758adbaf405110ac
======================================================================
pay ticket adf89d46c9e167b5758adbaf405110ac
default channel ticket adf89d46c9e167b5758adbaf405110ac
cipux_task_list_user_accounts at 1260214026 status TRUE \
(adf89d46c9e167b5758adbaf405110ac)
pay ticket adf89d46c9e167b5758adbaf405110ac
default channel ticket d9f3a0acce41b72a5c4dbff23f2e8c9c
explicit channel ticket d9f3a0acce41b72a5c4dbff23f2e8c9c
Summary:
start at: 1260213967
stop  at: 1260214027
seconds:  60
true    : 291
false   : 0
```

11 CAT - other CipUX XML-RPC clients

A CipUX Administration Tool (**CAT**) do not have to be a XML-RPC client. At the time of this writing all of them are. One reason is that a XML-RPC clients for **CipUX** to not have to have root rights to access the XML-RPC server. Some of the existing clients are Web clients, where it is out of question running this clients with root rights. Another reason is that **CipUX** XML-RPC hide most of the complexity of the storage layer from the client and that the objects which are created are the same among all clients. Because **CipUX** functions as an abstraction layer between the clients and the operation system additional features can be implemented and used among all clients.

This section tries to summarize all **CAT**s to give an overview about the current development as well as a starting point regarding the search for further informations about different clients. The order of this sections represented the age of the tool.

11.1 CAT-Web

The development of CAT-Web started as CAT in the year 2000 when there was a need for a **CipUX** Administration Tool. The first university internal release 1.0 was a standalone CGI program for the apache web server. For the version 2.0.0 CAT was converted to a standard webmin module and used up to version 3.2.16 with root rights. This approach was obsolete when webmin was expelled from Debian. The name changed from CAT ot CAT-Web and is basically a rewrite from scratch. Some pre-releases that already uses the XML-RPC server where made public in 2007 - 2009. The next version 3.4.0.0 will be released ending 2009 or beginning 2010 and is a CGI application for apache (and probably other web servers) written in object orientated Perl that supports template driven skinning.

Home page: http://www.cipux.org (English)

11.2 CipuxPHP

CipuxPHP, is a web interface for managing users on Skolelinux. It's a module for the CMS Moodle, but it was also written to work in a stand-alone mode in the case of Moodle isn't installed.

The starting point of this software where actually before the CipUX XML-RPC server by itself where developed.

2005: The first version was developed by Benjamin Sonntag in a real programming session with the French Skolelinux Team in Forbach, France in July 2005.

2006: Jean-Charles Siegel took over the development in Summer 2006 at the great summer meeting at Carreau Wendel's Mines Museum (Petite-Rosselle, France) and developed the second version that was able to access CipUX 3.2.6, which was deployed at the French Add-On CD for Skolelinux.

A second big dev-camp in Extramadura's Summer University (Spain's area) was the theater of an important development of CipUX.

2007: The third major version was developed by Jean-Charles Siegel and Christophe Gossen for an university project that supports a non official release of CipUX in 2007. It was written in PHP object, using templates to display the web pages.

2010: The forth major version development started again Jean-Charles Siegel together with the French Skolelinux Team in 2010 for CipUX 3.4.0.x. This is still under development.

The goal of cipuxPHP is to *manage easily* the pupils in Skolelinux. Create, modify, delete them. But also manage classes (considered as groups) and move pupils from one class to an other.

Home page: http://wiki.skolelinux.fr/CipuxXmlRpc (French)

(Jean-Charles Siegel)

11.3 CipUX-Passwd

CipUX::Passwd is a Perl module and a command line script that can be used as an CipUX XML-RPC client to set once own password. It was developed by Christian Külker in 2009 and released as CipUX-Passwd-3.4.0.2.

11.4 CATweasel

The project was started in early 2009 for the RLP-Skolelinux project. Main contributor is the ed-media GmbH from Kaiserslautern Germany. Debian maintainer for CATweasel is Jonas Smedegard. Many thanks for that Jonas! CATweasel was started because there was a steady demand for more interactivity in the web UI for CipUX. Klaus Knopper then decided to contract the ed-media GmbH to start a new project that should establish a basis on a modern web framework to accomplish that. That was the birth of CATweasel.

At the moment there is no fancy website with lots of screenshots. Just a boring Wiki with docu and the repository on bitbucket:
http://bitbucket.org/edmedia/catweasel/wiki/Home

(Jochen Breuer)