YumaPro ypclient-pro Manual

YANG-Based Unified Modular Automation Tools

YumaPro Client Library

Version 19.10-12

Table Of Contents

1 Preface

1.1 Legal Statements

1.2 Additional Resources

This document assumes you have successfully set up the software as described in the printed document:

YumaPro Installation Guide

Other documentation includes:

YumaPro API Quickstart Guide

YumaPro Quickstart Guide

YumaPro User Manual

YumaPro netconfd-pro Manual

YumaPro yangcli-pro Manual

YumaPro yangdiff-pro Manual

YumaPro yangdump-pro Manual

YumaPro Developer Manual

YumaPro yp-system API Guide

YumaPro yp-show API Guide

YumaPro Yocto Linux Quickstart Guide

YumaPro yp-snmp Manual

To obtain additional support you may contact YumaWorks technical support department:

support@yumaworks.com

1.2.1 WEB Sites

- **YumaWorks**
  - https://www.yumaworks.com
  - Offers support, training, and consulting for YumaPro.
- **Netconf Central**
  - http://www.netconfcentral.org/
  - Free information on NETCONF and YANG, tutorials, on-line YANG module validation and documentation database
- **Yang Central**

- http://www.yang-central.org
- Free information and tutorials on YANG, free YANG tools for download

- **NETCONF Working Group Wiki Page**
  - http://trac.tools.ietf.org/wg/netconf/trac/wiki
  - Free information on NETCONF standardization activities and NETCONF implementations
- **NETCONF WG Status Page**
  - http://tools.ietf.org/wg/netconf/
  - IETF Internet draft status for NETCONF documents
- **libsmi Home Page**
  - http://www.ibr.cs.tu-bs.de/projects/libsmi/
  - Free tools such as smidump, to convert SMIv2 to YANG

1.2.2 Mailing Lists

- **NETCONF Working Group**
  - https://mailarchive.ietf.org/arch/browse/netconf/
  - Technical issues related to the NETCONF protocol are discussed on the NETCONF WG mailing list. Refer to the instructions on https://www.ietf.org/mailman/listinfo/netconf for joining the mailing list.
- **NETMOD Working Group**
  - https://datatracker.ietf.org/wg/netmod/documents/
  - Technical issues related to the YANG language and YANG data types are discussed on the NETMOD WG mailing list. Refer to the instructions on the WEB page for joining the mailing list.

1.3 Conventions Used in this Document

The following formatting conventions are used throughout this document:

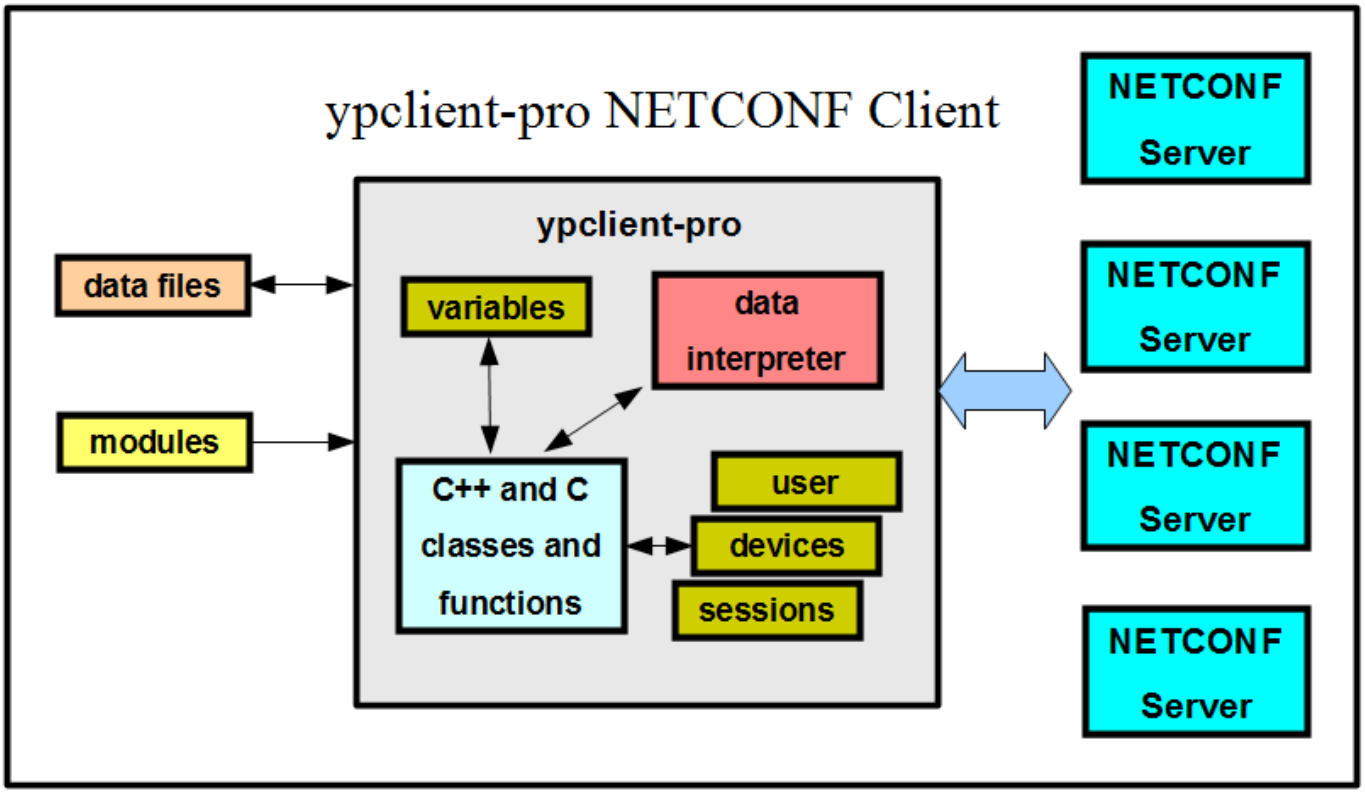
Documentation Conventions

Convention	Description
--foo	CLI parameter foo
<foo>	XML parameter foo
foo	**yangcli-pro** command or parameter
$FOO	Environment variable FOO
$$foo	**yangcli-pro** global variable foo
some text	Example command or PDU
some text	Plain text
Informational text	Useful or expanded information
Warning text	Warning information indicating possibly

Convention	Description
	unexpected side-effects

2 ypclient-pro User Guide

Program Components



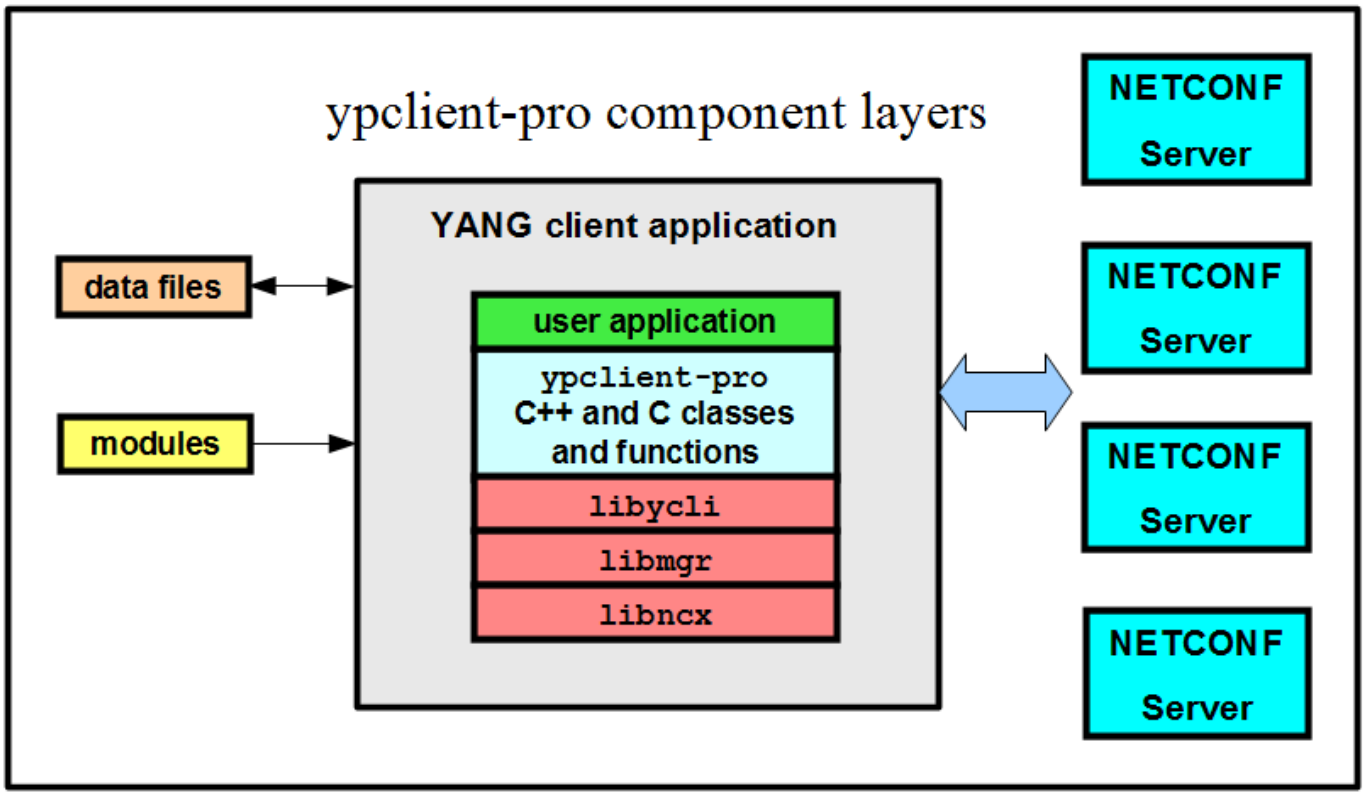
2.1 Introduction

The **ypclient-pro** library is designed to build NETCONF and RESTCONF client applications to connect and exchange YANG data with NETCONF and RESTCONF servers. It facilitates applications written in C++ and a C wrapper provides C and other programming languages such as Python, Java, etc. to use the library.

The code examples and the associated api-*.hpp & c-api-*.h files are liberally commented and contain detailed guidelines on how to uses the classes and functions contained in the library. It will be instructive to read the relevant parts of these files as you create your applications.

This document provides an outline of the functions in the example code and how to build and run the examples. Also the ypclient-pro factory classes are described.

Program Layers



The component layers of a client application created using ypclient-pro are shown in the diagram above and described in the table below.

ypclient-pro component layering

Function	Description
user application	the code you write
ypclient-pro	the classes and functions described in this manual and the associated header files
libycli	NETCONF & RESTCONF client support
libmgr	handles all the basic NETCONF and RESTCONF details so a simple internal API can be used by applications
libncx	handles many of the core NETCONF and RESTCONF/YANG data structure support, including all of the YANG/YIN, XML, and XPath processing

2.1.1 Features

The **ypclient-pro** client has the following features:

- C++ YANG client library plus a C wrapper to allow other languages to use it
- Provides session classes to create, save and load session information
- Support for multiple sessions to multiple servers
- Support for SSH & TLS
- Provides device classes to create, save, and load device information
- Provides a user class to create, save, and load user information
- Robust handling of errors
- Session notification monitoring
- Convenience functions that allow sending the server a request, capturing and displaying the response, and dealing with any errors

2.1.2 Files Required to Build Applications

You'll need the api-*.hpp files that your code is going to use from the list in the section “ypclient-pro C++ header files” and any other header files that you’ll need. If you are going to use C code then you’ll need to select the c-api-*.h files you require in the section “ypclient-pro C header files”.

The binary libraries used are listed in the Makefiles for each example app, see below.

NOTE: If you are building the code from source you will need to use the PTHREADS=1 flag for make as ypclient-pro requires two threads. If you are using a binary distribution use the appropriate yumapro-pthreads-* binary for your platform.

2.1.3 Example Code Preparation

In the /usr/share/yumapro/src/yp-client folder you will find several example programs. They are provided to show how the supplied library is used to connect and exchange information with servers and how notifications from the server are handled. The examples are based on the YumaWorks' netconfd-pro server but they should be easily modified to any standard NETCONF or RESTCONF server. Please refer to the your server's manual for compatibility with the examples.

The C++ examples

Example code	Function
sget-system.cpp	Minimum code to connect to a server and exchange information using SSH
sget-system-tls.cpp	Minimum code to connect to a server and exchange information using TLS
toaster.cpp	An app following the toaster example in the YumaPro Quickstart Guide
libtest.cpp	tests loading libraries

The C wrapper example

Example code	Function
c-toaster.c	A c-wrapper app following the toaster example in the YumaPro Quickstart Guide

In the toaster examples above, toaster.cpp and c-toaster.c, C++ and C code respectively, follow the example of libtoaster in the section "Getting Started with toaster.yang" in the YumaPro Quickstart Guide. The example demonstrates the basic principles of how to use a NETCONF session.

The examples have three parameters that are needed to connect to a server

Parameter	Usage	Default value
cServer	Server name	"localhost"
cUsername	User name	"user"
cPassword	User password	None – the user is prompted for the password

Before running the code you should edit the .cpp & .c files to apply the cServer and cUsername values that are appropriate for your system, cPassword is prompted for in the example code by a convenience function that asks for your password. If needed this could be edited to use a static string or to use some other authentication mechanism if required.

If you are running a netconfd-pro server you should launch it with the parameters below to allow you to see the debug messages as the examples are running and avoid any issues with previously loaded YANG modules or data:

```
netconfd-pro log-level=debug3 access-control=off factory
```

2.1.4 Example Code: Minimal Example - sget-system.cpp

The sget-system.cpp example code demonstrates the following:

- setup code to deal with any exceptions
- start a NETCONF session to a server for a user – ask them for their password
- some YangAPI object settings are made to make things more readable
- send the request "sget /netconf-state/statistics" to the server to request a node of the server's YANG datastore
- receive the response from the server into a buffer that can be processed
- close the session to the server

This is an outline of the minimal code required for an application. It's function is to send a command, in this instance "sget /netconf-state/statistics", to a NETCONF server, display the response it receives and deal with any errors that may occur. The response is stored in a buffer which can then be processed with string functions, an XML parser, etc.

The outline below provides a simple structure for ease of readability and annotation for the file sget-system.cpp with non-essential comments and code removed.

On some of the internal functions the removed code has been replaced by "..." to indicate there is missing code. Please refer to the file sget-system.cpp for the full code and comments.

```
// All of the YumaWorks C++ code is in namespace yuma, to avoid naming clashes
// and cluttering the default namespace.
using namespace yuma;

// Used to tell this code to prompt for a password. Don't change this.
const xstring cPromptForPassword { "*" };

// The server to use. Note that the yuma::xstring class requires that its
// initializers be in UTF-8 format. There's no need to explicitly mark the
// strings here as UTF-8 since they're entirely ASCII, we've done so solely for
// explanatory purposes.
const xstring cServer { u8"localhost" };

// The username to use.
const xstring cUsername { u8"user" };

// The password to use. The cPrompForPassword here is a signal to the code below
// that it should prompt the user for his password. You can replace this with an
// actual password if desired, or nullptr to use no password. (You can also
// achieve the no-password effect by omitting the password() call to the
// SessionFactory object entirely.)
const xstring cPassword { cPromptForPassword };

...

int main() {
    // The first step is to initialize the API by creating a yuma::YangAPI
    // object. No parameters are needed for this, but the object must persist
    // as long as you want to use the library.
    YangAPI api;

    // You can change many settings using the YangAPI object. As an example,
    // we'll set the indent value to four spaces instead of the default two.
    api.indent(4);

    // for XML structured response output add
    api.displayMode(displaymode::xml);

    // We enclose most of the API code in a try/catch structure, to catch any
    // exceptions detected in it. The code will generally use the
    // yuma::StatusFailure exception, but a few other std::exception-based
    // classes might also appear on occasion.
    int r = EXIT_FAILURE;
    try {
        // We'll start by creating a notification stack to receive notifications
        // for the session. We'll tie it into the session through a
        // SessionFactory call. We could do it with the Session object's
        // addNotificationFunction member function instead, but we don't need
        // the Token to remove it later, which is the only advantage of doing
        // it that way.
        NotificationStack notifications;

        // If the user has requested that we prompt for a password, we'll do so
        // here.
        xstring password = (cPassword == cPromptForPassword ?
            fetchPassword("Enter your password for server " + cServer + ":",
                true) :
            cPassword);

        // Now we'll create a User. The User class holds information about the
        // user account on a server -- its username and credentials. There's no
        // requirement for this, we could specify all of these options when
        // creating the Session instead, but a User can be saved to a file for
```

```
        // later reloading and reuse.
        User user = UserFactory("me", cUsername).password(password).create();

        // We'll create a Device too. The Device class holds information about
        // the server we're trying to contact: its type, address information,
        // and protocols. Note that we've omitted the port parameter (it would
        // normally be specified after the server's address in the DeviceFactory
        // constructor), which tells the API to use the default port for the
        // session type.
        //
        // As with the User object, there's no requirement to create this. We
        // could specify all this information when creating a Session. The only
        // advantage to using a Device is that it can be saved to a file for
        // later reloading and reuse.
        Device server = NetconfDeviceFactory("myDevice", cServer).create();

        // Now we can create a session. We use one of the SessionFactory classes
        // for this, ending with a call to SessionFactory::create().
        cout << "Attempting to establish connection to server " << cServer <<
            "..." << endl;
        session = DeviceSessionFactory(user, server)
            .notificationFunction(notifications)
            .create();
        cout << "Session established!" << endl << endl;

        do {

             // Collect the response from the server to the "sget  /netconf-
state/statistics" request
             Response response = session.command("sget /netconf-
state/statistics");
        if (response.error) {
            cout << "Received error:" << endl << response.result << endl << endl;
            break;
             }
             cout << "error:" << response.error << endl << "response:" << endl <<
response.result << endl;

            // If we get this far, we've succeeded.
            r = EXIT_SUCCESS;
        } while (0);

        // Finally, close the session.
        if (!sendCommand("close-session")) {
            cout << "Unexpected failure, aborting." << endl;
            return EXIT_FAILURE;
        }

        // When the Session object goes out of scope, the session is terminated
        // and automatically cleaned up.
    } catch (StatusFailure &e) {
        cout << "StatusFailure exception: " << e.what() << endl;
    } catch (std::exception &e) {
        cout << "other exception: " << e.what() << endl;
    }

    // When the YangAPI object goes out of scope, the library is deinitialized.
    return r;
}
```

When sget-system.cpp is built and executed the following is displayed:

```
user@YW-U18W sget-system$ make
g++ -g -std=c++11 -I/usr/include/yumapro/ycli -I/usr/include/yumapro/mgr
-I/usr/include/yumapro/ncx -I/usr/include/yumapro/platform
-I/usr/include/libxml2/libxml -I/usr/include/libxml2 -pthread -o sget-system sget-
system.cpp -lyumapro_ycli -lyumapro_mgr -lyumapro_ncx -lyumapro_subsys-pro -lssl
-lcrypto -lssh2 -lcrypt
user@YW-U18W sget-system$
user@YW-U18W sget-system$ ./sget-system
Enter your password for server localhost: ***********
Attempting to establish connection to server localhost...
Session established!

error:0
response:
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <data>
        <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-
monitoring">
            <statistics>
                <netconf-start-time>2019-12-13T23:26:08Z</netconf-start-time>
                <in-bad-hellos>0</in-bad-hellos>
                <in-sessions>4</in-sessions>
                <dropped-sessions>0</dropped-sessions>
                <in-rpcs>9</in-rpcs>
                <in-bad-rpcs>0</in-bad-rpcs>
                <out-rpc-errors>0</out-rpc-errors>
                <out-notifications>0</out-notifications>
            </statistics>
        </netconf-state>
    </data>
</rpc-reply>
 Sending: close-session
Received: OK

user@YW-U18W sget-system$
```

2.1.5 Example Code: Minimal Example using TLS - sget-system-tls.cpp

In the previous example, sget-system.cpp, sets up a session to communicate with the servers using SSH with the session defined by:

```
session = DeviceSessionFactory(user, server)
            ...
            .create();
```

the user defined by:

```
User user = UserFactory("me", cUsername).password(password).create();
```

and the device defined by:

```
Device server = NetconfDeviceFactory("myDevice", cServer).create();
```

To use TLS rather than SSH use the sget-system-tls.cpp example with the session defined the same way:

```
session = DeviceSessionFactory(user, server)
            ...
            .create();
```

the user defined by:

```
User user = UserFactory("me", cUsername)
          .sslClientCertificate("$HOME/certs/client.crt")
          .sslClientCertificateKeyFile("$HOME/certs/client.key")
          .sslTrustStore("/etc/ssl/certs/")
          .fallbackOkay(false).create();
```

(To configure TLS certificates for YumaPro SDK see the “Configure TLS” section of the YumaPro SDK Installation Guide)

Specify the port and protocol the communication with the device will use with:

```
Device server = NetconfDeviceFactory("myDevice",
                                     cServer,
                                     6513,
                                     netconf-tls").create();
```

When invoking sget-system-tls the output will look something like:

```
user@YW-U18W sget-system-tls$ make
g++ -g -std=c++11 -I/usr/include/yumapro/ycli -I/usr/include/yumapro/mgr
-I/usr/include/yumapro/ncx -I/usr/include/yumapro/platform
-I/usr/include/libxml2/libxml -I/usr/include/libxml2 -pthread -o sget-system-tls
sget-system-tls.cpp -lyumapro_ycli -lyumapro_mgr -lyumapro_ncx -lyumapro_subsys-
pro -lssl -lcrypto -lssh2 -lcrypt
user@YW-U18W
user@YW-U18W sget-system-tls$ ./sget-system-tls
Attempting to establish connection to server localhost...
Session established!

 Sending: $$display-mode = cli
Received: OK

error:0
response:
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <data>
        <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-
monitoring">
            <statistics>
                <netconf-start-time>2019-12-14T00:17:57Z</netconf-start-time>
                <in-bad-hellos>0</in-bad-hellos>
                <in-sessions>1</in-sessions>
                <dropped-sessions>0</dropped-sessions>
                <in-rpcs>1</in-rpcs>
                <in-bad-rpcs>0</in-bad-rpcs>
                <out-rpc-errors>0</out-rpc-errors>
                <out-notifications>0</out-notifications>
            </statistics>
        </netconf-state>
    </data>
</rpc-reply>
 Sending: close-session
Received: OK

user@YW-U18W sget-system-tls$
```

2.1.6 Example Code: Toaster Application - toaster.cpp

The toaster.cpp example code demonstrates the following, see also the example of libtoaster in the section “Getting Started with toaster.yang” in the YumaPro Quickstart Guide.

- setup code to deal with any exceptions
- start a NETCONF session to a server for a user – ask them for their password
- display the session number (if you are running the server in debug mode you’ll see the session numbers match)
- setup to receive notifications for this session
- setup a subscription with the server for this session
- messages received from the server will display the message number that will match the message numbers displayed in the debug output from the server)
- set some session globals and local variables to show how that is done
- load the toaster.yang module to the server
- mgrload the toaster.yang module
- lock the datastore
- create the /toaster node (if it already exists, due to running the program several times, ignore the error)
- unlock the datastore
- display the /toaster node by sending the request “sget /toaster”
- display the /toaster node by sending the request “xget /toaster”
- send the command to make-toast with specific parameters
- receive the toastStatus=done notification (after about 12seconds)
- resend the command to make-toast with specific parameters
- don’t wait for the toastStatus=done notification, send the cancel-toast command
- receive the toastStatus=cancelled notification
- close the session to the server

The outline below provides a simple structure for ease of readability and annotation for the file toaster.cpp with non-essential comments and code removed.

On some of the internal functions the removed code has been replaced by “...” to indicate there is missing code. Please refer to the file toaster.cpp for the full code and comments.

```
// All of the YumaWorks C++ code is in namespace yuma, to avoid naming clashes
// and cluttering the default namespace.
using namespace yuma;

// Used to tell this code to prompt for a password. Don't change this.
```

```
const xstring cPromptForPassword { "*" };

// The server to use. Note that the yuma::xstring class requires that its
// initializers be in UTF-8 format. There's no need to explicitly mark the
// strings here as UTF-8 since they're entirely ASCII, we've done so solely for
// explanatory purposes.
const xstring cServer { u8"localhost" };

// The username to use.
const xstring cUsername { u8"john" };

// The password to use. The cPrompForPassword here is a signal to the code below
// that it should prompt the user for his password. You can replace this with an
// actual password if desired, or nullptr to use no password. (You can also
// achieve the no-password effect by omitting the password() call to the
// SessionFactory object entirely.)
const xstring cPassword { cPromptForPassword };

...

int main() {
    // The first step is to initialize the API by creating a yuma::YangAPI
    // object. No parameters are needed for this, but the object must persist
    // as long as you want to use the library.
    YangAPI api;

    // You can change many settings using the YangAPI object. As an example,
    // we'll set the indent value to four spaces instead of the default two.
    api.indent(4);

    // We enclose most of the API code in a try/catch structure, to catch any
    // exceptions detected in it. The code will generally use the
    // yuma::StatusFailure exception, but a few other std::exception-based
    // classes might also appear on occasion.
    int r = EXIT_FAILURE;
    try {
        // We'll start by creating a notification stack to receive notifications
        // for the session. We'll tie it into the session through a
        // SessionFactory call. We could do it with the Session object's
        // addNotificationFunction member function instead, but we don't need
        // the Token to remove it later, which is the only advantage of doing
        // it that way.
        NotificationStack notifications;

        // If the user has requested that we prompt for a password, we'll do so
        // here.
        xstring password = (cPassword == cPromptForPassword ?
            fetchPassword("Enter your password for server " + cServer + ":",
                true) :
            cPassword);

        // Now we'll create a User. The User class holds information about the
        // user account on a server -- its username and credentials. There's no
        // requirement for this, we could specify all of these options when
        // creating the Session instead, but a User can be saved to a file for
        // later reloading and reuse.
        User user = UserFactory("me", cUsername).password(password).create();

        // We'll create a Device too. The Device class holds information about
        // the server we're trying to contact: its type, address information,
        // and protocols. Note that we've omitted the port parameter (it would
        // normally be specified after the server's address in the DeviceFactory
        // constructor), which tells the API to use the default port for the
```

```
        // session type.
        //
        // As with the User object, there's no requirement to create this. We
        // could specify all this information when creating a Session. The only
        // advantage to using a Device is that it can be saved to a file for
        // later reloading and reuse.
        Device server = NetconfDeviceFactory("myDevice", cServer).create();

        // Now we can create a session. We use one of the SessionFactory classes
        // for this, ending with a call to SessionFactory::create().
        //
        // The notificationFunction option tells it where to send notifications
        // (messages that are not direct replies to commands). The timeout
        // option tells it how long to wait before deciding that something has
        // gone wrong with the connection; it defaults to sixty seconds, but
        // telling it to time out after fifteen will let it fail more quickly,
        // if we know that the connection to the device we're trying to contact
        // doesn't have an inordinate amount of latency.
        cout << "Attempting to establish connection to server " << cServer <<
            "..." << endl;
        session = DeviceSessionFactory(user, server)
            .notificationFunction(notifications)
            .timeout(std::chrono::seconds(15))
            .create();
        cout << "Session " << session.serverSessionId() << " established!"
            << endl << endl;

        // Setup a subscription with the server
        if (!sendCommand("create-subscription")) {
            cout << "Unexpected failure, aborting." << endl;
            return EXIT_FAILURE;
        }

        do {
            // You can create or read global and local variables through the
            // Session class as well. These variables can contain anything
            // except the null character.
            session.global("param1", "parameter value including \nnewlines,
'single-' and \"double-quotes\", and \tt\ta\tb\ts");
            cout << "param1 is currently set to '" << session.global("param1")
                << "'" << endl << endl;

            // If this one fails (maybe the 'toaster' module isn't installed?),
            // we'll skip the rest of the tests and just close the session. If
            // it succeeds, it should receive (and print) the toaster module's
            // revision, though we're not checking for that.
            if (!sendCommand("ysys:load toaster")) {
                cout << "Unexpected failure, skipping toaster tests." << endl;
                break;
            }


            // This one should succeed.
            if (!sendCommand("get-locks")) {
                cout << "Unexpected failure, aborting." << endl;
                break;
            }

            // This one will fail if the /toaster has already been created.
            // For our purposes, that's not an error. If it succeeds, save the
            // changes.
            if (sendCommand("create /toaster")) {
                if (!sendCommand("save")) {
```

```
                    cout << "Unexpected failure, aborting." << endl;
                    break;
                }
            }

            // This one should succeed.
            if (!sendCommand("release-locks")) {
                cout << "Unexpected failure, aborting." << endl;
                break;
            }

            // These two commands should return (identical) data. We don't check
            // the data, just that there's no error.
            if (!sendCommand("sget /toaster") || !sendCommand("xget /toaster")) {
                cout << "Unexpected failure, aborting." << endl;
                break;
            }

            // Let's make some toast! We'll check for any recent notifications
            // first, to ensure that there aren't any that might throw us off.
            while (!notifications.empty()) {
                notifications.pop();
            }

           if (sendCommand("make-toast toasterDoneness=1
toasterToastType=toast:wonder-bread")) {
                // This will take about twelve seconds, if all goes well, so
                // we'll use a twenty-second timeout.
                cout << "Toast should be done in about 12 seconds..." << endl;
                if (!notifications.waitForNotification(std::chrono::seconds(20)))
{
                    cout << "Did not receive toastDone notification within
provided time limit, aborting."
                        << endl;
                    break;
                } else {
                    // Remove the toastDone notification from the stack, so it
                    // doesn't interfere with the later call.
                    notifications.pop();
                }
            } else {
                cout << "Unexpected failure, aborting." << endl;
                break;
            }

            // We'll repeat the make-toast command, but this time we'll cancel
            // it immediately.
            if (!notifications.empty()) {
                cout << "Unexpected notification in stack, aborting." << endl;
                break;
            } else if (sendCommand("make-toast toasterDoneness=1
toasterToastType=toast:frozen-waffle")) {
                if (sendCommand("cancel-toast")) {
                    // This notification should be sent almost immediately.
                    // We'll only use a five-second timeout, and no message.
                    if (!
notifications.waitForNotification(std::chrono::seconds(5))) {
                        cout << "Did not receive toastDone notification within
provided time limit, aborting."
                            << endl;
                        break;
                    } else {
                        // Remove the toastDone notification from the stack.
```

```
                    notifications.pop();
                }
            } else {
                cout << "Unexpected failure, aborting." << endl;
                break;
            }
        } else {
            cout << "Unexpected failure, aborting." << endl;
            break;
        }

        // If we get this far, we've succeeded.
        r = EXIT_SUCCESS;
    } while (0);

    // Finally, close the session.
    if (!sendCommand("close-session")) {
        cout << "Unexpected failure, aborting." << endl;
        return EXIT_FAILURE;
    }

    // When the Session object goes out of scope, the session is terminated
    // and automatically cleaned up.
} catch (StatusFailure &e) {
    cout << "StatusFailure exception: " << e.what() << endl;
} catch (std::exception &e) {
    cout << "other exception: " << e.what() << endl;
}

// When the YangAPI object goes out of scope, the library is deinitialized.
return r;
}
```

When toaster is built and executed the following is displayed:

```
user@YW-U18W

user@YW-U18W toaster$ make
g++ -g -std=c++11 -I/usr/include/yumapro/ycli -I/usr/include/yumapro/mgr
-I/usr/include/yumapro/ncx -I/usr/include/yumapro/platform
-I/usr/include/libxml2/libxml -I/usr/include/libxml2 -pthread -o toaster
toaster.cpp -lyumapro_ycli -lyumapro_mgr -lyumapro_ncx -lyumapro_subsys-pro -lssl
-lcrypto -lssh2 -lcrypt
user@YW-U18W toaster$
user@YW-U18W toaster$ ./toaster
Enter your password for server localhost: **********
Attempting to establish connection to server localhost...
Session 4 established!

Sending command: create-subscription
Received OK (2)

param1 is currently set to 'parameter value including
newlines, 'single-' and "double-quotes", and   t      a      b      s'

Sending command: ysys:load toaster
```

```
Received notification:
notification {
    eventTime 2019-12-14T00:28:53Z
    yang-library-change {
        module-set-id 3755
    }
}

Received notification:
notification {
    eventTime 2019-12-14T00:28:53Z
    netconf-capability-change {
        changed-by {
            username john
            session-id 4
            source-host 127.0.0.1
        }
        added-capability http://netconfcentral.org/ns/toaster?
module=toaster&revision=2009-11-20
    }
}

Received response (3):
rpc-reply {
    mod-revision 2009-11-20
}

Sending command: get-locks
Received OK (4)

Sending command: create /toaster
Received OK (6)

Sending command: save
Received notification:
notification {
    eventTime 2019-12-14T00:28:54Z
    netconf-config-change {
        changed-by {
            username john
            session-id 4
            source-host 127.0.0.1
        }
        datastore running
        edit {
            target /toast:toaster
            operation create
        }
    }
}

Received OK (7)

Sending command: release-locks
Received OK (8)

Sending command: sget /toaster
Received response (10):
rpc-reply {
    data {
        toaster {
            toasterManufacturer 'Acme, Inc.'
            toasterModelNumber 'Super Toastamatic 2000'
```

```
            toasterStatus up
        }
    }
}

Sending command: xget /toaster
Received response (11):
rpc-reply {
    data {
        toaster {
            toasterManufacturer 'Acme, Inc.'
            toasterModelNumber 'Super Toastamatic 2000'
            toasterStatus up
        }
    }
}

Sending command: make-toast toasterDoneness=1 toasterToastType=toast:wonder-bread
Received OK (12)

Toast should be done in about 12 seconds...
Received notification:
notification {
    eventTime 2019-12-14T00:29:07Z
    toastDone {
        toastStatus done
    }
}

Sending command: make-toast toasterDoneness=1 toasterToastType=toast:frozen-waffle
Received OK (13)

Sending command: cancel-toast
Received notification:
notification {
    eventTime 2019-12-14T00:29:08Z
    toastDone {
        toastStatus cancelled
    }
}

Received OK (14)

Sending command: close-session
Received OK (15)

user@YW-U18W toaster$
```

2.1.7 NotificationStack class to receive session notifications

```
// This class will receive notifications for the session, through its operator()
// function. It also holds them and manages access to them from multiple
// threads, necessary because they're usually added from a background thread.
//
// To the outside world, this class will act like std::stack<xstring>.
class NotificationStack {
    public:
    NotificationStack(): mData(std::make_shared<data_t>()) { }

    // Functions to emulate std::stack<xstring>.
    bool empty() const { lock_t lock(mData->mutex); return mData->notes.empty(); }
    size_t size() const { lock_t lock(mData->mutex); return mData->notes.size(); }
    const xstring& top() const { lock_t lock(mData->mutex); return mData-
>notes.front(); }
    void push(const xstring &n) { lock_t lock(mData->mutex); mData-
>notes.push_front(n); }
    void pop() { lock_t lock(mData->mutex); mData->notes.pop_front(); }

    // This function blocks until a notification is received or the provided
    // wait time runs out.
    bool waitForNotification(const std::chrono::duration<clock_t> &maxWaitTime) {
        // Start waiting. The condition variable eliminates the need for
        // wasteful polling.
        lock_t lock(mData->mutex);
        return mData->waiter.wait_until(lock, clock_t::now() + maxWaitTime,
            [=]() { return !mData->notes.empty(); });
    }

    // This function turns objects created from this class into "functors" --
    // objects that act like a function, but can hold a non-global state as
    // well. We're exploiting that capability to create an object that we can
    // use as a notification function.
    bool operator()(const xstring &newNotification) {
        cout << "Received notification:" << endl << newNotification << endl
            << endl;

        // Must lock the mutex before all accesses to mData->notes, because this
        // function will usually be called from a background thread.
        {
            lock_t lock(mData->mutex);
            mData->notes.push_back(newNotification);
        }
        mData->waiter.notify_one();

        // We could return false here, if we were only looking for a particular
        // notification and had found it. That would remove this notification
        // function from the Session's list and prevent it from receiving any
        // later notifications. Since we want to continue receiving them, we'll
        // return true instead.
        return true;
    }

    private:
    typedef std::mutex mutex_t;
    typedef std::unique_lock<mutex_t> lock_t;
    typedef std::chrono::steady_clock clock_t;
```

```
    struct data_t {
        mutable mutex_t mutex;
        std::deque<xstring> notes;
        std::condition_variable waiter;
    };

    // Since we're using this class as a functor, it has to be copyable, and
    // neither std::mutex nor std::condition_variable is. This works around that
    // problem.
    std::shared_ptr<data_t> mData;
};
```

2.1.8 Convenience function: sendCommand

```
// A convenience function for sending a command and reporting its results.
static bool sendCommand(const xstring &command) {
    try {
        cout << " Sending: " << command << endl;

        Response response = session.command(command);
        if (response.error) {
            cout << "Received error:" << endl << response.result << endl << endl;
            return false;
        }

        if (response.result.empty())
            cout << "Received: OK" << endl << endl;
        else
            cout << "Received:" << endl << response.result << endl << endl;
        return true;
    } catch (StatusFailure &e) {
        cout << "StatusFailure exception: " << e.what() << endl << endl;
        return false;
    } catch (std::exception &e) {
        cout << "other exception: " << e.what() << endl << endl;
        return false;
    }
}
```

2.1.9 Convenience function: fetchPassword

```
// A function to request a password from the user, showing only asterisks on the
// screen.
xstring fetchPassword(const xstring &prompt, bool showAsterisk = true) {
    const char cReturn = 10, cBackspace = 127;

    std::cout << prompt << ' ';

    xstring r;
    while (true) {
        struct termios t_old;
        tcgetattr(STDIN_FILENO, &t_old);
        struct termios t_new = t_old;
        t_new.c_lflag &= ~(ICANON | ECHO);
        tcsetattr(STDIN_FILENO, TCSANOW, &t_new);
        xmlChar ch = getchar();
        tcsetattr(STDIN_FILENO, TCSANOW, &t_old);

        if (ch == cReturn) {
            break;
        } else if (ch == cBackspace) {
            if (r.length() != 0) {
                if (showAsterisk)
                    std::cout << "\b \b";
                r = r.substr(0, r.length() - 1);
            }
        } else {
            r += ch;
            if (showAsterisk)
                cout << '*';
        }
    }

    std::cout << std::endl;
    return r;
}
```

2.1.10 Useful Server Information and Configuration

Useful information from the server

Data	Function	Example
Session number	session.serverSessionId()	toaster.cpp
Message number	Response response = session.command(command); xstring id = (response.internalRequestId && !response.internalRequestId.empty() ? " (" + response.internalRequestId + ")" : "");	toaster.cpp
Set display mode to XML	api.displayMode(displaymode::xml) – see below	sget-system.cpp

With api.displayMode(displaymode::xml); then sending the command: "sget /netconf-state/statistics" generates a response similar to:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <data>
        <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-
monitoring">
            <statistics>
                <netconf-start-time>2019-12-14T00:38:22Z</netconf-start-time>
                <in-bad-hellos>0</in-bad-hellos>
                <in-sessions>3</in-sessions>
                <dropped-sessions>0</dropped-sessions>
                <in-rpcs>16</in-rpcs>
                <in-bad-rpcs>0</in-bad-rpcs>
                <out-rpc-errors>0</out-rpc-errors>
                <out-notifications>5</out-notifications>
            </statistics>
        </netconf-state>
    </data>
</rpc-reply>
```

and with api.displayMode(displaymode::json); the response is:

```
{
    "yuma-netconf:rpc-reply": {
        "yuma-netconf:data": {
            "ietf-netconf-monitoring:netconf-state": {
                "statistics": {
                    "netconf-start-time":"2019-12-14T00:38:22Z",
                    "in-bad-hellos":0,
                    "in-sessions":4,
                    "dropped-sessions":0,
                    "in-rpcs":19,
                    "in-bad-rpcs":0,
                    "out-rpc-errors":0,
                    "out-notifications":5
                }
            }
        }
    }
}
```

3 Factory classes: Session, Device, and User

3.1.1 Session Factories

Listed below are the four classes for creating Session objects: NetconfSessionFactory, RestconfSessionFactory, YangapiSessionFactory and DeviceSessionFactory. They are defined in api-session.hpp and use the base class SessionFactory.

```
class NetconfSessionFactory:
    virtual public SessionFactory,
    public _CommonAddin<NetconfSessionFactory>,
    public _KeyfilesAddin<NetconfSessionFactory>,
    public _SslFilesAddin<NetconfSessionFactory>
{
    public:
    // 'username' is the username for the account. 'target' is the ASCII IP
    // address or DNS hostname of the target machine. 'port' is the port number
    // to use, if not default. Will throw std::invalid_argument if 'username' or
    // 'target' are NULL.
    NetconfSessionFactory(const xstring &username, const xstring &target,
        uint16_t port = 0);

    // This variant uses a User object, which may specify several options.
    NetconfSessionFactory(const User &user, const xstring &target, uint16_t port
        = 0);

    NetconfSessionFactory& tcpTransport();
    NetconfSessionFactory& tcpNcxTransport();
    NetconfSessionFactory& sshTransport();
    NetconfSessionFactory& tlsTransport();

    private:
    NetconfSessionFactory(pvt_t&, const xstring &target, uint16_t port);
};
```

```
class RestconfSessionFactory:
    virtual public SessionFactory,
    public _CommonAddin<RestconfSessionFactory>,
    public _EncodingAddin<RestconfSessionFactory>,
    public _KeyfilesAddin<RestconfSessionFactory>,
    public _SslFilesAddin<RestconfSessionFactory>
{
    public:
    // 'username' is the username for the account. 'target' is the ASCII IP
    // address or DNS hostname of the target machine. 'port' is the port number
    // to use, if not default. Will throw std::invalid_argument if 'username' or
    // 'target' are NULL.
    RestconfSessionFactory(const xstring &username, const xstring &target,
        uint16_t port = 0);

    RestconfSessionFactory(const User &user, const xstring &target, uint16_t
```

```
        port = 0);

    RestconfSessionFactory& tcpTransport();
    RestconfSessionFactory& tlsTransport();

    private:
    RestconfSessionFactory(pvt_t&, const xstring &target, uint16_t port);
};
```

```
class YangapiSessionFactory:
    virtual public SessionFactory,
    public _CommonAddin<YangapiSessionFactory>,
    public _EncodingAddin<YangapiSessionFactory>,
    public _KeyfilesAddin<YangapiSessionFactory>,
    public _SslFilesAddin<YangapiSessionFactory>
{
    public:
    // 'username' is the username for the account. 'target' is the ASCII IP
    // address or DNS hostname of the target machine. 'port' is the port number
    // to use, if not default. Will throw std::invalid_argument if 'username' or
    // 'target' are NULL.
    YangapiSessionFactory(const xstring &username, const xstring &target,
        uint16_t port = 0);

    YangapiSessionFactory(const User &user, const xstring &target, uint16_t port
        = 0);

    private:
    YangapiSessionFactory(pvt_t&, const xstring &target, uint16_t port);
};
```

```
// Use this when you have a Device that you want to use for configuration. It
// will automatically determine the device type and the default transport to
// use. You can alter the transport, but attempting to select one that doesn't
// match the Device type will fail with an exception.
class DeviceSessionFactory:
    virtual public SessionFactory,
    public _CommonAddin<DeviceSessionFactory>,
    public _EncodingAddin<DeviceSessionFactory>,
    public _KeyfilesAddin<DeviceSessionFactory>,
    public _SslFilesAddin<DeviceSessionFactory>
{
    public: //
    DeviceSessionFactory(const xstring &username, const Device &device);
    DeviceSessionFactory(const User &user, const Device &device,
                         bool use_defserver = false);

    // Only usable with a Netconf or Restconf device.
    DeviceSessionFactory& tcpTransport();

    // Only usable with a Netconf device.
    DeviceSessionFactory& tcpNcxTransport();

    // Only usable with a Netconf device.
    DeviceSessionFactory& sshTransport();
```

```
    // Only usable with a Restconf device.
    DeviceSessionFactory& tlsTransport();

    private:
    DeviceSessionFactory(pvt_t&, const Device &device);

    Device::Type::type_t mType;
};
```

3.1.2 Device Factories

Listed below are the three classes for creating Device objects: NetconfDeviceFactory, RestconfDeviceFactory, and YangapiDeviceFactory. They are defined in api-device.hpp and use the base class DeviceFactory.

```
class NetconfDeviceFactory:
    public DeviceFactory
{
    public: //
    NetconfDeviceFactory(const xstring &name,
                         const xstring &address,
                         uint16_t port = 0,
                         const xstring &transport = "ssh");

};

class RestconfDeviceFactory:
    public DeviceFactory
{
    public: //
    RestconfDeviceFactory(const xstring &name,
                          const xstring &address,
                          uint16_t port = 0);
};

class YangapiDeviceFactory:
    public DeviceFactory
{
    public: //
    YangapiDeviceFactory(const xstring &name,
                         const xstring &address,
                         uint16_t port = 0);
};
```

3.1.3 User Factory

Listed below is the User Factory which is defined in api-users.hpp,

```
// Creating a user could require a number of parameters, many of which have
// default values. The UserFactory class provides an easy-to-use way to specify
// only the parameters that you need to.
//
// All parameters, except the 'id' and 'username' parameters to the constructor,
// are optional.
class UserFactory {
    public: //
    // Creates a UserFactory object. 'id' is the name this User will be known
    // by; 'username' is the username for logging into a server.
    UserFactory(const xstring &id, const xstring &username);

    // The password. Omit this (or use nullptr for it) if no password will be
    // used, or if you'll specify the password separately.
    UserFactory& password(const xstring &password);

    // The filespecs for the public key/private key files.
    UserFactory& publicPrivateKeyFiles(const filesystem::path &publicFilespec,
        const filesystem::path &privateFilespec);

    // The filespecs for the SSL client certificate and its keyfile.
    UserFactory& sslClientCertificate(const filesystem::path &filespec);
    UserFactory& sslClientCertificateKeyFile(const filesystem::path &filespec);

    // The filespec for the SSL trust store. Required (?) if using SSL.
    UserFactory& sslTrustStore(const filesystem::path &filespec);

    // Allow fallback to tcp if ssl failed. Defaults to true.
    UserFactory& fallbackOkay(bool setTo);

    // After filling in the required parameters, you can either call this
    // function or pass the class to the User constructor.
    User create() const;

    // Writes a text-format version of the object's settings, for debugging
    // purposes.
    std::string str() const;

    ////////////////////////////////////////////////////////////////////////
    private: //
    xstring mName;
    xstring mUsername;

    xstring mPassword;

    filesystem::path mPublicKey, mPrivateKey;

    bool mSslFallbackOk;
    filesystem::path mSslCert, mSslKey, mSslTrustStore;

    friend class User;
};
```

4 ypclient-pro C++ Header Files

A description of the ypclient-pro .hpp header files follows. These files can be found in /usr/include/yumapro/mgr and /usr/include/yumapro/ycli.

4.1 /usr/include/yumapro/mgr

4.1.1 api-devices.hpp

A class representing a Device that this library can connect to, and classes and functions to create, save, load, and modify Devices.

4.1.2 api-exceptions.hpp

The domain-specific exceptions used in the C++ code.

4.1.3 api-filesystem.hpp

A limited version of the std::filesystem library proposed for C++17. Code written against this should work with std::filesystem with minimal changes.

4.1.4 api-session.hpp

A class representing a connected Session, and classes and functions for creating and dealing with Sessions.

4.1.5 api-token.hpp

A class representing a value that is guaranteed to be unique within a run of the program. It is used for several purposes in the API.

4.1.6 api-users.hpp

A class representing a user account on a Device, and classes and functions to create, save, load, and modify Users.

4.1.7 api-xstring.hpp

A string class using unsigned characters, designed to handle XML data, which can also represent a nullptr.

4.2 /usr/include/yumapro/ycli

4.2.1 api-grouping.hpp

A class representing a group of objects, typedefs, and subgroups.

4.2.2 api-library.hpp

A class representing a library of module files and allowing you to enumerate, find, and load them.

4.2.3 api-module.hpp

A class representing a Module and describing its interface.

4.2.4 api-object.hpp

A class representing an Object, describing its parameters, type, and any child Objects.

4.2.5 api-typedef.hpp

A class representing a type definition and its parameters.

4.2.6 api-yangapi.hpp

A class representing the API library itself. It handles initialization, cleanup, and general settings of the library.

5 ypclient-pro C Header Files

The C++ API classes allow use of the ypclient-pro API, but only from C++ programs. Languages like C and Python can't use them directly. The functions described in the c-api-* files give programming languages that can access libraries using C functions access to the ypclient-pro API.

Do not attempt to mix the C and C++ API functions.

A description of the ypclient-pro .h header files follows. These files can be found in /usr/include/yumapro/mgr and /usr/include/yumapro/ycli.

5.1 /usr/include/yumapro/mgr

5.1.1 c-api-devices.h

Functions to create, save, load, and modify a device/server.

5.1.2 c-api-session.h

Functions dealing with sessions.

5.1.3 c-api-users.h

Functions to create, save, load, and modify a user account on a device.

5.2 /usr/include/yumapro/ycli

5.2.1 c-api-error.h

Functions for retrieving and displaying C-style error codes.

5.2.2 c-api-grouping.h

Functions for dealing with groups of objects, typedefs, and subgroups.

5.2.3 c-api-library.h

Functions for dealing with libraries.

5.2.4 c-api-module.h

Functions for dealing with modules.

5.2.5 c-api-object.h

Functions for dealing with objects.

5.2.6 c-api-typedef.h

Functions for dealing with a type definition and its parameters.

5.2.7 c-api-yangapi.h

Functions for the initialization, cleanup, and general settings of the library.