LIBNETCONF Library

Sunday, September 11, 2016 11:47 AM

Client workflow: https://rawgit.com/CESNET/libnetconf/master/doc/doxygen/html/d1/d25/client.html

There are two possible ways to integrate a specific device configuration into libnetconf.
- use transAPI modules.
- the custom datastore implementation

Here is a description of using libnetconf functions in a NETCONF client.

1. **Set verbosity (optional).**
   The verbosity of the libnetconf can be set by **nc_verbosity()**. By default, libnetconf is completely silent.
   There is a default message-printing function that writes messages on stderr. The application's specific message printing function can be set via **nc_callback_print()** function.
2. **Set SSH authentication methods priorities (optional).**
   libnetconf supports several SSH authentication methods for connecting to a NETCONF server over SSH. However, the used method is selected from a list of supported authentication methods provided by the server. Client is allowed to specify the priority of each supported authentication method via **nc_ssh_pref()** function. The authentication method can also be disabled using a negative priority value.
   Default priorities are following:
   - *Interactive* (value 3)
   - *Password* (value 2)
   - *Public keys* (value 1)
3. **Set your own callback(s) for the SSH authentication methods (optional).**
   User credentials are received via the callback functions specific for each authentication method. There are default callbacks, but application can set their own via:
   - *Interactive* - **nc_callback_sshauth_interactive()**
   - *Password* - **nc_callback_sshauth_password()**
   - *Public keys* - **nc_callback_sshauth_passphrase()**. Here can the paths to the key files be also specified by nc_set_publickey_path() and nc_set_privatekey_path(). If not set, libnetconf tries to find them in the default paths.
4. **Connect to the NETCONF server(s).**
   Simply call **nc_session_connect()** to connect to the specified host via SSH. Authentication method is selected according to the default values or the previous steps.
5. **Prepare NETCONF rpc message(s).**
   Creating NETCONF rpc messages is covered by the functions described in the section **NETCONF rpc**. The application prepares NETCONF rpc messages according to the specified attributes. These messages can be then repeatedly used for communication over any of the created NETCONF sessions.
6. **Send the message to the selected NETCONF server.**
   To send created NETCONF rpc message to the NETCONF server, use **nc_session_send_rpc()** function. **nc_session_send_recv()** function connects sending and receiving the reply (see the next step) into one blocking call.
7. **Get the server's rpc-reply message.** When the NETCONF rpc is sent, use **nc_session_recv_reply()** to receive the reply. To learn when the reply is coming, a file descriptor of the communication channel can be checked by poll(), select(), ... This descriptor can be obtained via **nc_session_get_eventfd()** function.
8. **Close the NETCONF session.**
   When the communication is done, the NETCONF session should be freed (session is also properly closed) via **nc_session_free()** function.
9. **Free all created objects.**
   Do not forget to free created rpc messages (**nc_rpc_free()**), **filters** (**nc_filter_free()**) or received NETCONF rpc-replies (**nc_reply_free()**).

Screen clipping taken: 9/11/2016 11:48 AM

https://rawgit.com/CESNET/libnetconf/master/doc/doxygen/html/da/db3/server.html

Server Workflow

Here is a description of using libnetconf functions in a NETCONF server. According to the used architecture, the workflow can be split between an agent and a server. For this purpose, functions nc_rpc_dump(), nc_rpc_build() and nc_session_dummy() can be very helpful.

1. **Set the verbosity** (optional)
   The verbosity of the libnetconf can be set by nc_verbosity(). By default, libnetconf is completely silent.
   There is a default message printing function writing messages on stderr. On the server side, this is not very useful, since server usually runs as a daemon without stderr. In this case, something like syslog should be used. The application's specific message printing functio
2. **Initiate libnetconf**
   As the first step, libnetconf MUST be initiated using nc_init(). At this moment, the libnetconf subsystems, such as NETCONF Notifications or NETCONF Access Control, are initiated according to the specified parameter of the nc_init() function.
3. **Set With-defaults basic mode** (optional)
   By default, libnetconf uses *explicit* basic mode of the with-defaults capability. The basic mode can be changed via ncdflt_set_basic_mode() function. libnetconf supports *explicit*, *trim*, *report-all* and *report-all-tagged* basic modes of the with-defaults capability.
4. **Initiate datastore**
   Now, a NETCONF datastore(s) can be created. Each libnetconf's datastore is connected with a single configuration data model. This connection is defined by calling the ncds_new() function, which returns a datastore handler for further manipulation with an uninitialized type will be used. Optionally, some implementation-type-specific parameters can be set (e.g. ncds_file_set_path()). Finally, datastore must be initiated by ncds_init() that returns datastore's ID which is used in the subsequent calls. There is a set of special implicit datastc Optionally, each datastore can be extended by an augment data model that can be specified by ncds_add_model(). The same function can be used to specify models to resolve YANG's `import` statements. Alternatively, using ncds_add_models_path(), caller can spec for the needed models based on the modules names. Filename of the model is expected in a form `module_name[@revision].yin`.
   Caller can also switch on or off the YANG `feauters` in the specific module using ncds_feature_enable(), ncds_feature_disable(), ncds_features_enableall() and ncds_features_disableall() functions.
   Finally, ncds_consolidate() must be called to check all the internal structures and to solve all `import`, `uses` and `augment` statements.
5. **Initiate the controlled device**
   This step is actually out of the libnetconf scope. From the NETCONF point of view, startup configuration data should be applied to the running datastore at this point. ncds_device_init() can be used to perform this task, but applying running configuration data to the con
6. **Accept incoming NETCONF connection.**
   This is done by a single call of nc_session_accept() or nc_session_Accept_username() alternatively. Optionally, any specific capabilities supported by the server can be set as the function's parameter.
7. Server loop
   Repeat these three steps:
   a. **Process incoming requests.**
      Use nc_session_recv_rpc() to get the next request from the client from the specified NETCONF session. In case of an error return code, the state of the session should be checked by nc_session_get_status() to learn if the session can be further used.
      According to the type of the request (nc_rpc_get_type()), perform an appropriate action:
      - *NC_RPC_DATASTORE_READ* or *NC_RPC_DATASTORE_WRITE*: use ncds_apply_rpc2all() to perform the requested operation on the datastore. If the request affects the running datastore (nc_rpc_get_target() returns NC_DATASTORE_RUNNING), a
      - *NC_RPC_SESSION*: See the Netopeer example server source codes. There will be a common function added in the future to handle these requests.

   b. **Reply to the client's request.**
      The reply message is automatically generated by the ncds_apply_rpc2all() function. However, server can generate its own replies using nc_reply_ok(), nc_reply_data() (nc_reply_data_ns()) or nc_reply_error() functions. The reply is sent to the client using nc_s

   c. **Free all unused objects.**
      Do not forget to free received rpc messages (nc_rpc_free()) and any created replies (nc_reply_free()).
8. **Close the NETCONF session.**
   Use functions nc_session_free() to close and free all the used sources and structures connected with the session. Server should close the session when a nc_session_* function fails and libnetconf set the status of the session as non-working (nc_session_get_status !=
9. **Close the libnetconf instance**
   Close internal libnetconf structures and subsystems by the nc_close() call.

Screen clipping taken: 9/11/2016 2:00 PM

Transport Support on the Server Side

There is no specific support for neither SSH or TLS on the server side. libnetconf doesn't implement SSH nor TLS server - it is expected, that NETCONF server application uses external application (sshd, stunnel,...) serving as S server stdin, where libnetconf can read the data, and getting data from the NETCONF server stdout to encapsulate the data and send to a client.

For both cases, SSH as well as TLS, there are two functions: **nc_session_accept()** and **nc_session_accept_username()**, that serve to accept incoming connection despite the transport protocol. As mentioned, they read data functions is in recognizing NETCONF username. **nc_session_accept()** guesses username from the process's UID. For example, in case of using SSH Subsystem mechanism in OpenSSH implementation, SSH daemon automati (NETCONF server/agent) to the UID of the logged user. But in case of other SSH/TLS server, this doesn't have to be done. In such a case, NETCONF server itself is supposed to correctly recognize the NETCONF username an **nc_session_accept_username()**.

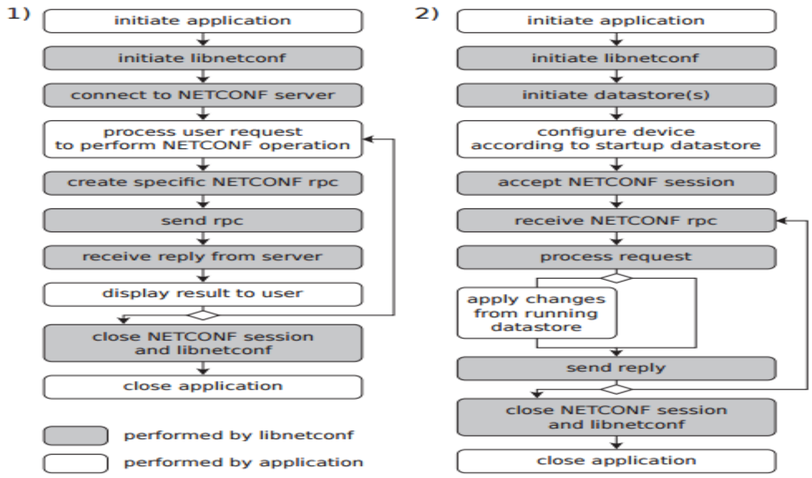

Fig. 3. Simplified workflow of the: 1) *libnetconf* client; 2) *libnetconf* server.

This diagram is from the technical paper on Libnetconf by the original Author.