Yuma® Developer Manual

YANG-Based Unified Modular Automation Tools

Table Of Contents

Yuma Developer Manual

1 Preface

1.1 Legal Statements

1.2 Restricted Rights Legend

1.3 Additional Resources

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

Yuma® Quickstart Guide

Depending on the version of Yuma you purchased, other documentation includes:

Yuma® User Manual

To obtain additional support you may email InterWorking Labs at this e-mail address

yuma-support@iwl.com

There are several sources of free information and tools for use with YANG and/or NETCONF.

The following section lists the resources available at this time.

1.3.1 WEB SITES

- **Yuma Tools Home Page**
  - http://yuma.iwl.com/
  - Official home page for Yuma Tools information
- **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

1.3.2 Mailing Lists

- **NETCONF Working Group**
  - http://www.ietf.org/html.charters/netconf-charter.html
  - Technical issues related to the NETCONF protocol are discussed on the NETCONF WG mailing list. Refer to the instructions on the WEB page for joining the mailing list.
- **NETMOD Working Group**
  - http://www.ietf.org/html.charters/netmod-charter.html
  - 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.

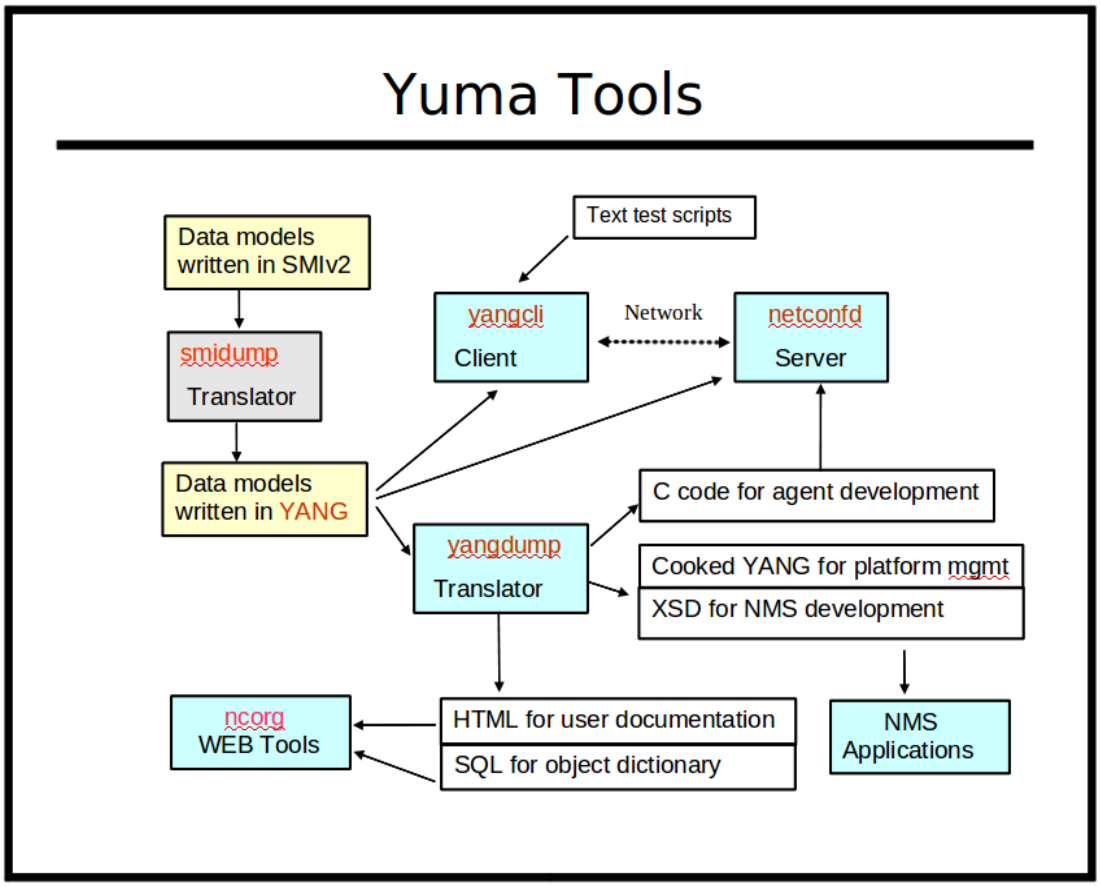
1.4 Conventions Used in this Document

The following formatting conventions are used throughout this document:

Documentation Conventions

Convention	Description
--foo	CLI parameter foo
<foo>	XML parameter foo
foo	**yangcli** command or parameter
$$FOO	Environment variable FOO
$$foo	**yangcli** global variable foo
`some text`	Example command or PDU
some text	Plain text

2 Software Overview



2.1 Introduction

Refer to section 3 of the Yuma User Manual for a complete introduction to Yuma Tools.

This section focuses on the software development aspects of NETCONF, YANG, and the **netconfd** server.

2.1.1 Intended Audience

This document is intended for developers of server instrumentation library software, which can be used with the programs in the Yuma suite. It covers the design and operation of the **netconfd** server, and the development of server instrumentation library code, intended for use with the **netconfd** server.

2.1.2 What does Yuma Do?

The Yuma Tools suite provides automated support for development and usage of network management information. Refer to the Yuma User Guide for an introduction to the YANG data modeling language and the NETCONF protocol.

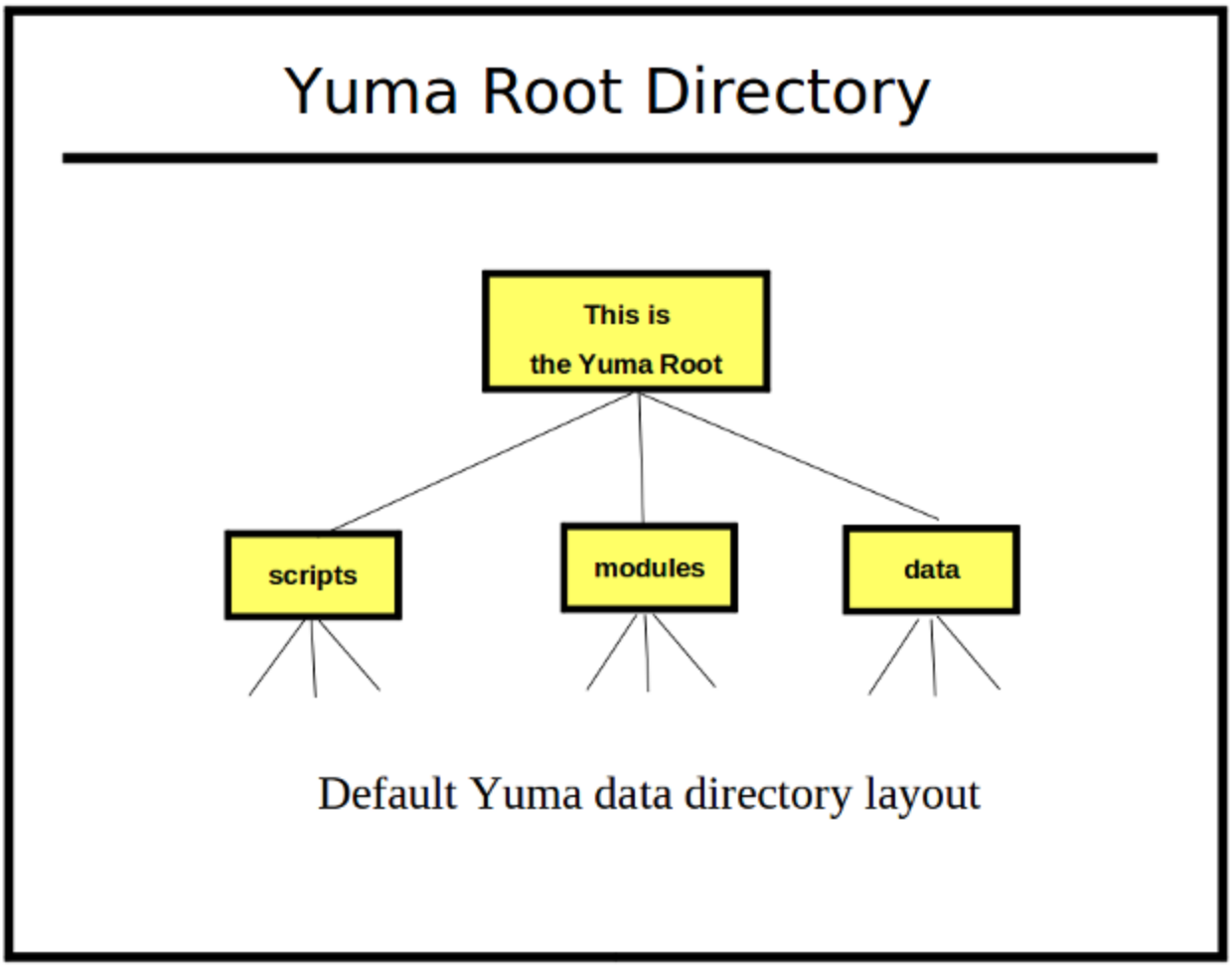
This section describes the Yuma development environment and the basic tasks that a software developer needs to perform, in order to integrate YANG module instrumentation into a device.

This manual contains the following information:

- Yuma Development Environment
- Yuma Runtime Environment
- Yuma Source Code Overview
- Yuma Server Instrumentation Library Development Guide

Yuma Tools programs are written in the C programming language, using the 'gnu99' C standard, and should be easily integrated into any operating system or embedded device that supports the Gnu C compiler.

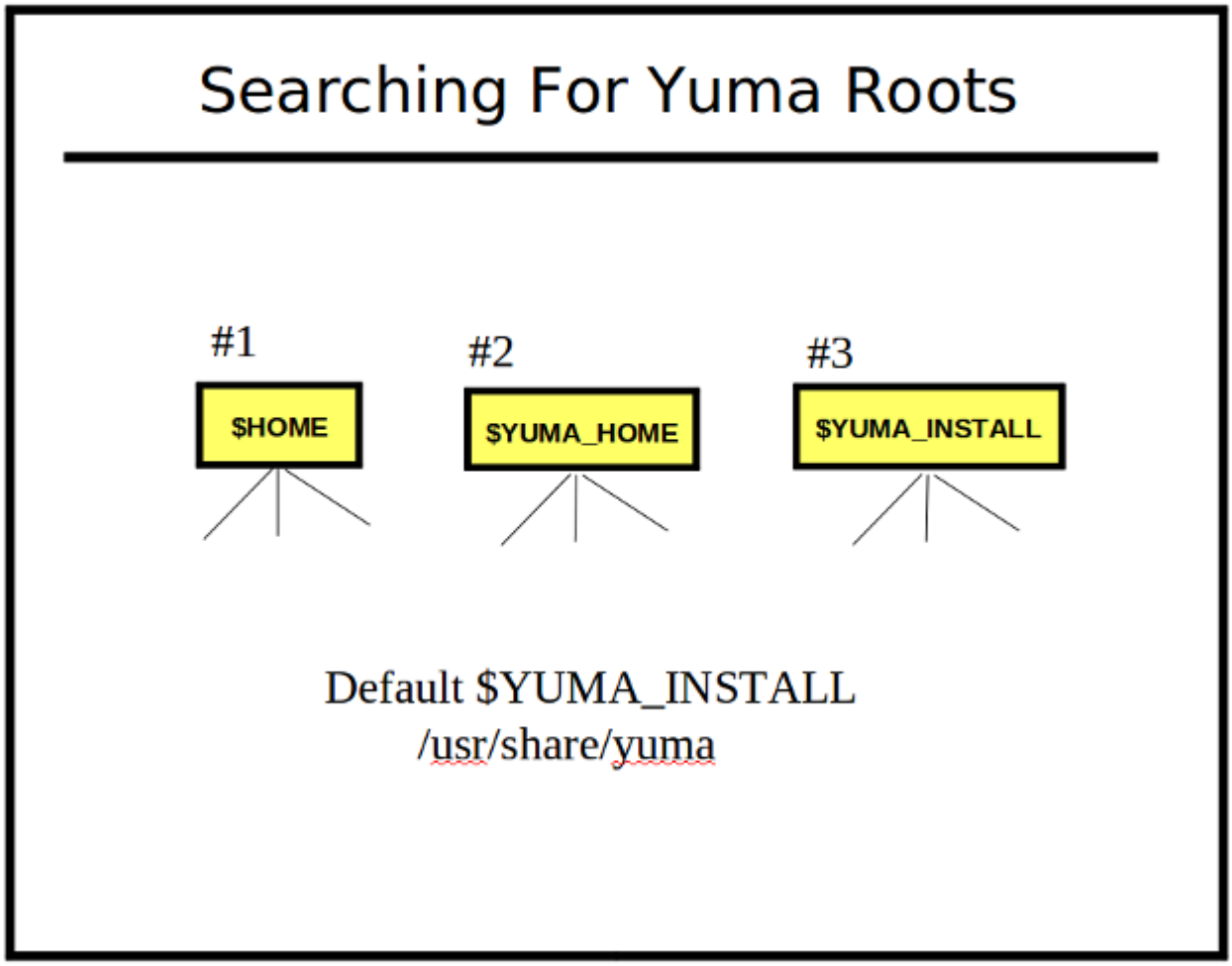
2.1.3 What is a Yuma Root?



The Yuma Tools programs will search for some types of files in default locations

- **YANG Modules**: The 'modules' sub-directory is used as the root of the YANG module library.
- **Client Scripts**: The yangcli program looks in the 'scripts' sub-directory for user scripts.
- **Program Data**: The yangcli and netconfd programs look for saved data structures in the 'data' sub-directory.

2.1.4 Searching Yuma Roots



1) $HOME Directory

The first Yuma root checked when searching for files is the directory identified by the $HOME environment variable. If a '**$HOME/modules**', '**$HOME/data**'. and/or '**$HOME/scripts**' directory exists, then it will be checked for the specified file(s).

2) The $YUMA_HOME Directory

The second Yuma root checked when searching for files is the directory identified by the $YUMA_HOME environment variable. This is usually set to private work directory, but a shared directory could be

used as well. If a '**$YUMA_HOME/modules**', '**$YUMA_HOME/data**'. and/or '**$YUMA_HOME/scripts**' directory exists, then it will be checked for the specified file(s).

3) The $YUMA_INSTALL Directory

The last Yuma root checked when searching for files is the directory identified by the $YUMA_INSTALL environment variable. If it is not set, then the default value of '**/usr/share/yuma**' is used instead. This is usually set to the public directory where all users should find the default modules. If a '**$YUMA_INSTALL/modules**', '**$YUMA_INSTALL/data**'. and/or '**$YUMA_INSTALL/scripts**' directory exists, then it will be checked for the specified file(s).

2.1.5 What is a SIL?

A SIL is a Server Instrumentation Library. It contains the 'glue code' that binds YANG content (managed by the **netconfd** server), to your networking device, which implements the specific behavior, as defined by the YANG module statements.

The **netconfd** server handles all aspects of the NETCONF protocol operation, except data model semantics that are contained in description statements. The server uses YANG files directly, loaded at boot-time or run-time, to manage all NETCONF content, operations, and notifications.

Callback functions are used to hook device and data model specific behavior to database objects and RPC operations. The **yangdump** program is used to generate the initialization, cleanup, and 'empty' callback functions for a particular YANG module. The callback functions are then completed (by you), as required by the YANG module semantics. This code is then compiled as a shared library and made available to the **netconfd** server. The 'load' command (via CLI, configuration file, protocol operation) is used (by the operator) to activate the YANG module and its SIL.

2.1.6 Basic Development Steps

The steps needed to create server instrumentation for use within Yuma are as follows:

- **Create the YANG module data model** definition, or use an existing YANG module.
  - Validate the YANG module with the **yangdump** program and make sure it does not contain any errors. All warnings should also be examined to determine if they indicate data modeling bugs or not.
  - `Example toaster.yang`
- Make sure the **$$YUMA_HOME** environment variable is defined, and pointing to your Yuma development tree.
- **Create a SIL development subtree**
  - Generate the directory structure and the Makefile with the **yuma-make-sil** script, installed in the **/usr/bin** directory.
  - `Example: mydir> yuma-make-sil toaster`
- **Create the C source code files** for the YANG module callback functions
  - Generate the H and C files with the 'make code' command, in the SIL 'src' directory
  - `Example: mydir/toaster/src> make code`
- **Use your text editor to fill in the device-specific instrumentation** for each object, RPC method, and notification. Almost all possible NETCONF-specific code is either handled in the

central stack, or generated automatically. so this code is responsible for implementing the semantics of the YANG data model.

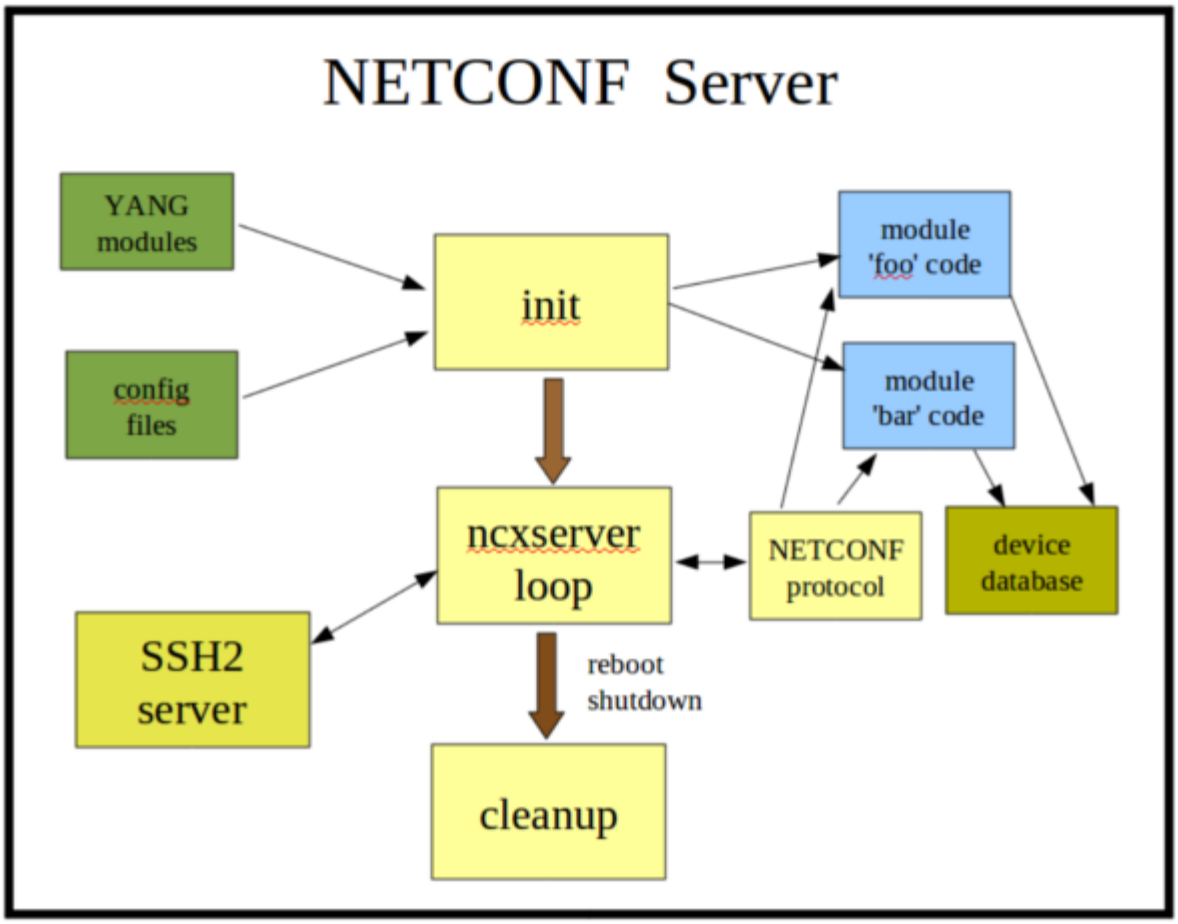
- **Compile your code**
  - Use the 'make' command in the SIL 'src' directory. This should generate a library file in the SIL 'lib' directory.
  - `Example:  mydir/toaster/src> make`

- **Install the SIL library so it is available to the netconfd server.**
  - Use the 'make install' command in the SIL 'src' directory.
  - `Example:  mydir/toaster/src> make install`

- **Run the netconfd server** (or build it again if linking with static libraries)
- **Load the new module**
  - Be sure to add a 'load' command to the configuration file if the module should be loaded upon each reboot.
  - `yangcli Example: load toaster`
- The **netconfd** server will load the specified YANG module and the SIL and make it available to all sessions.

2.2 Server Design

This section describes the basic design used in the **netconfd** server.



Initialization:

The **netconfd** server will process the YANG modules, CLI parameters, config file parameters, and startup device NETCONF database, then wait for NETCONF sessions.

ncxserver Loop:

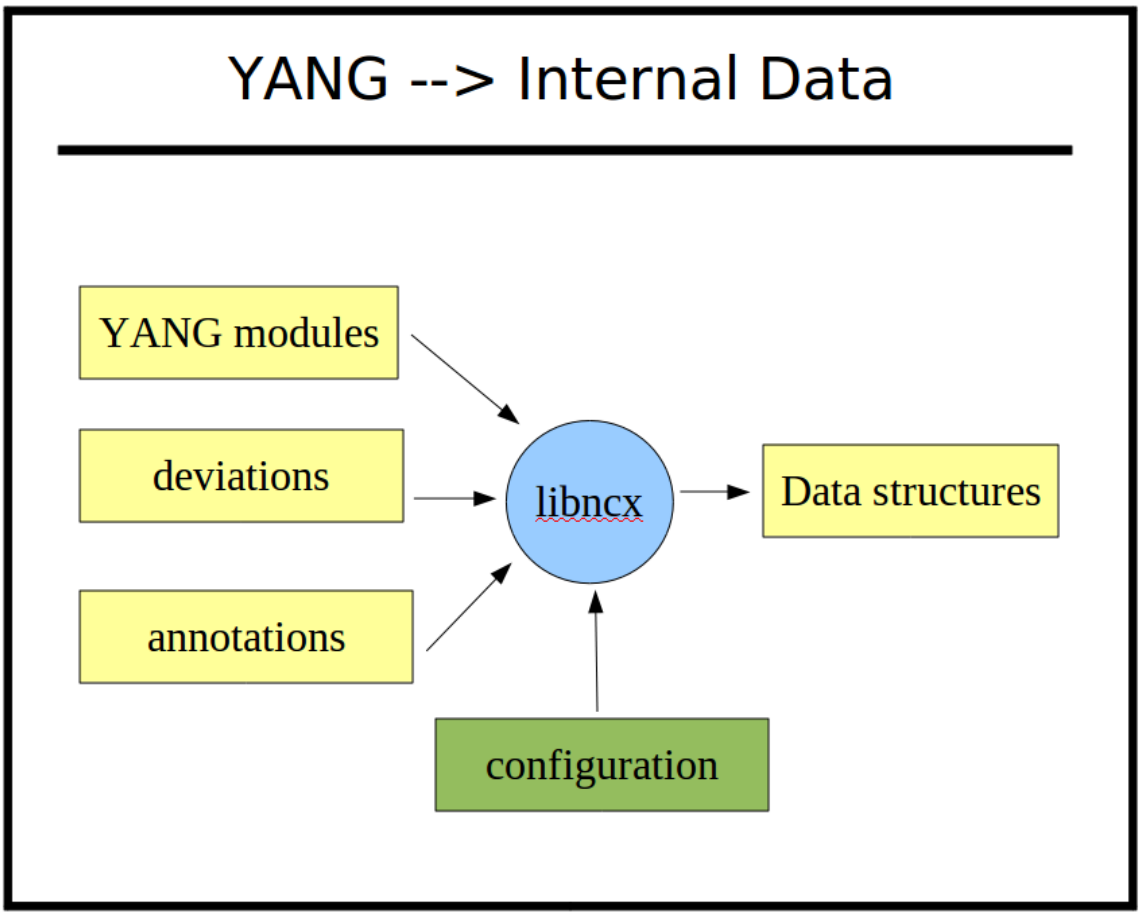
The SSH2 server will listen for incoming connections which request the 'netconf' subsystem.

When a new session request is received, the **netconf-subsystem** program is called, which opens a local connection to the **netconfd** server, via the ncxserver loop. NETCONF <rpc> requests are processed by the internal NETCONF stack. The module-specific callback functions (blue boxes) can be loaded into the system at build-time or run-time. This is the device instrumentation code, also called a server implementation library (SIL). For example, for **libtoaster**, this is the code that controls the toaster hardware.

Cleanup:

If the <shutdown> or <reboot> operations are invoked, then the server will cleanup. For a reboot, the init cycle is started again, instead of exiting the program.

2.2.1 YANG Native Operation



Yuma uses YANG source modules directly to implement NETCONF protocol operations automatically within the server. The same YANG parser is used by all Yuma programs. It is located in the 'ncx' source directory (libncx.so). There are several different parsing modes, which is set by the application.

In the 'server mode', the descriptive statements, such as 'description' and 'reference' are discarded upon input. Only the machine-readable statements are saved. All possible database validation, filtering, processing, initialization, NV-storage, and error processing is done, based on these machine readable statements.

For example, in order to set the platform-specific default value for some leaf, instead of hard-coded it into the server instrumentation, the default is stored in YANG data instead. The YANG file can be altered, either directly (by editing) or indirectly (via deviation statements), and the new or altered default value specified there.

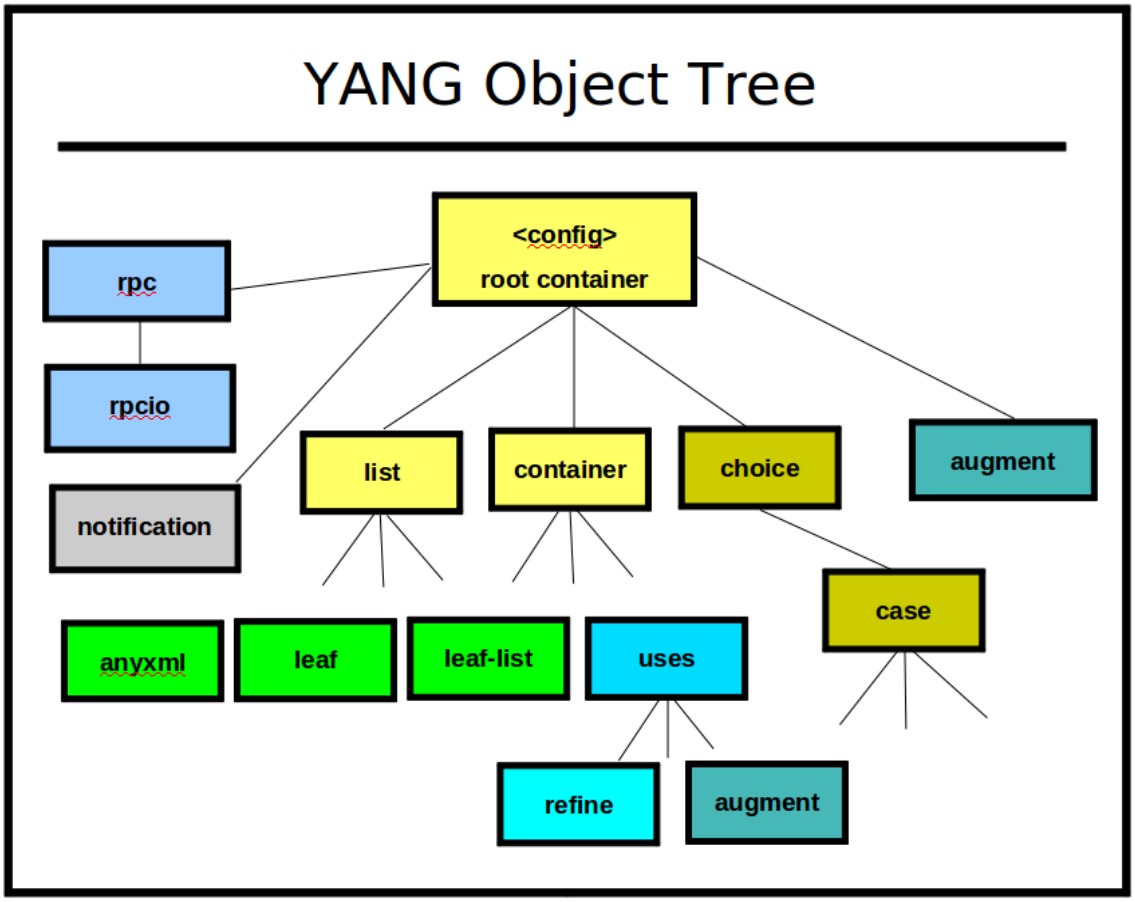
In addition, range statements, patterns, XPath expressions, and all other machine-readable statements are all processed automatically, so the YANG statements themselves are like server source code.

YANG also allows vendor and platform-specific deviations to be specified, which are like generic patches to the common YANG module for whatever purpose needed. YANG also allows annotations to be defined and added to YANG modules, which are specified with the 'extension' statement. Yuma uses some extensions to control some automation features, but any module can define extensions, and module instrumentation code can access these annotation during server operation, to control device behavior.

There are CLI parameters that can be used to control parser behavior such as warning suppression, and protocol behavior related to the YANG content, such as XML order enforcement and NETCONF

protocol operation support. These parameters are stored in the server profile, which can be customized for each platform.

2.2.2 YANG Object Tree



The YANG statements found in a module are converted to internal data structures.

For NETCONF and database operations, a single tree of **obj_template_t** data structures is maintained by the server. This tree represents all the NETCONF data that is supported by the server. It does not represent any actual data structure instances. It just defines the data instances that are allowed to exist on the server.

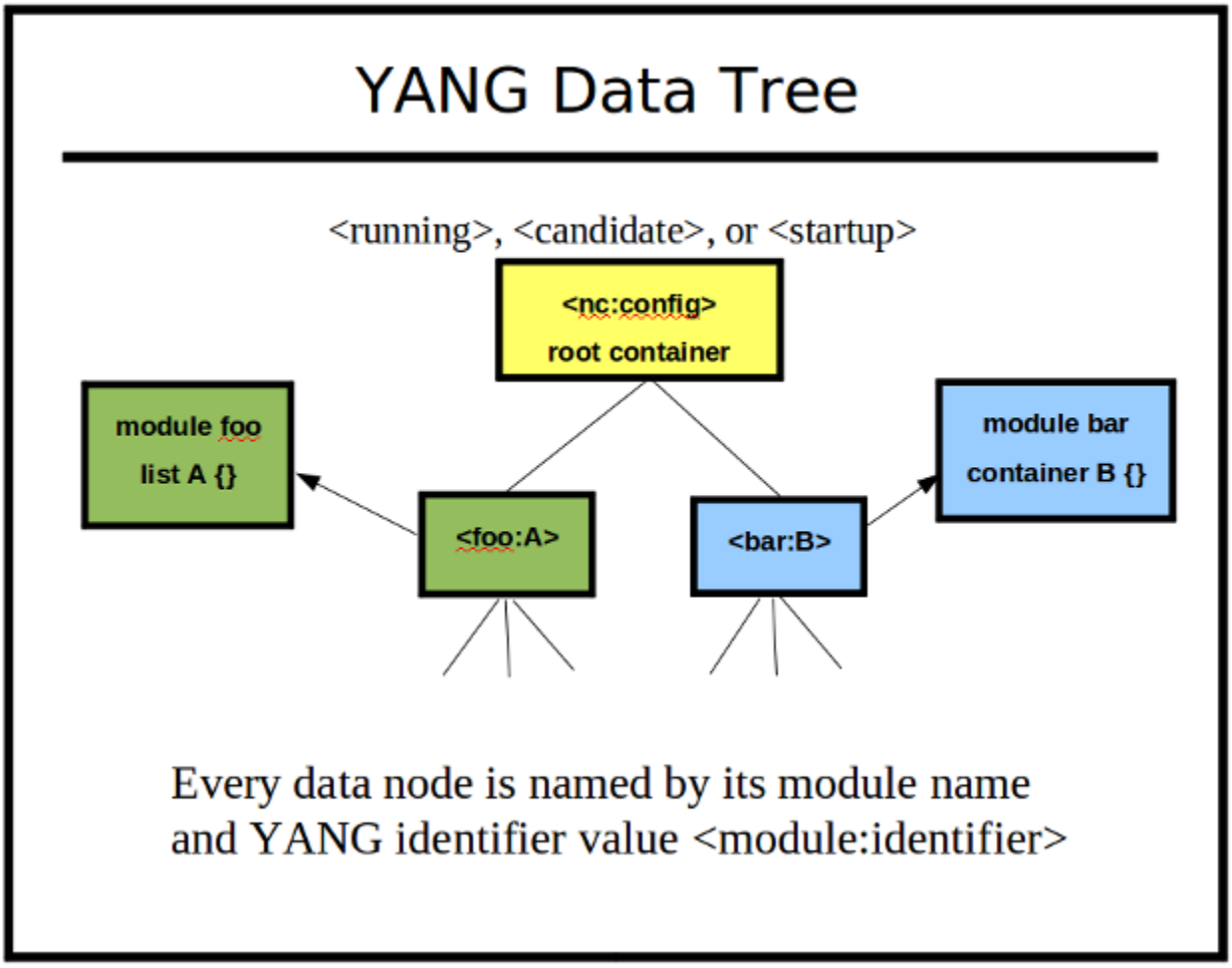
There are 14 different variants of this data structure, and the discriminated union of sub-data structures contains fields common to each sub-type. Object templates are defined in ncx/obj.h.

YANG Object Types

object type	description
OBJ_TYP_ANYXML	This object represents a YANG **anyxml** data node.
OBJ_TYP_CONTAINER	This object represents a YANG presence or non-presence **container.**

OBJ_TYP_CONTAINER + ncx:root	If the ncx:root extension is present within a container definition, then the object represents a NETCONF database **root**. No child nodes
OBJ_TYP_LEAF	This object represents a YANG **leaf** data node.
OBJ_TYP_LEAF_LIST	This object represents a YANG **leaf-list** data node.
OBJ_TYP_LIST	This object represents a YANG **list** data node.
OBJ_TYP_CHOICE	This object represents a YANG **choice** schema node. The only children allowed are case objects. This object does not have instances in the data tree.
OBJ_TYP_CASE	This object represents a YANG **case** schema node. This object does not have instances in the data tree.
OBJ_TYP_USES	This object represents a YANG **uses** schema node. The contents of the grouping it represents will be expanded into object tree. It is saved in the object tree even during operation, in order for the expanded objects to share common data. This object does not have instances in the data tree.
OBJ_TYP_REFINE	This object represents a YANG **refine** statement. It is used to alter the grouping contents during the expansion of a uses statement. This object is only allowed to be a child of a uses statement. It does not have instances in the data tree.
OBJ_TYP_AUGMENT	This object represents a YANG **augment** statement. It is used to add additional objects to an existing data structure. This object is only allowed to be a child of a uses statement or a child of a 'root' container. It does not have instances in the data tree, however any children of the augment node will generate object nodes that have instances in the data tree.
OBJ_TYP_RPC	This object represents a YANG **rpc** statement. It is used to define new <rpc> operations. This object will only appear as a child of a 'root' container. It does not have instances in the data tree. Only 'rpcio' nodes are allowed to be children of an RPC node.
OBJ_TYP_RPCIO	This object represents a YANG **input** or **output** statement. It is used to define new <rpc> operations. This object will only appear as a child of an RPC node. It does not have instances in the data tree.
OBJ_TYP_NOTIF	This object represents a YANG **notification** statement. It is used to define new <notification> event types. This object will only appear as a child of a 'root' container. It does not have instances in the data tree.

2.2.3 YANG Data Tree



A YANG data tree represents the instances of 1 or more of the objects in the object tree.

Each NETCONF database is a separate data tree. A data tree is constructed for each incoming message as well. The server has automated functions to process the data tree, based on the desired NETCONF operation and the object tree node corresponding to each data node.

Every NETCONF node (including database nodes) are distinguished with XML Qualified Names (QName). The YANG module namespace is used as the XML namespace, and the YANG identifier is used as the XML local name.

Each data node contains a pointer back to its object tree schema node. The value tree is comprised of the **val_value_t** structure. Only real data is actually stored in the value tree. For example, there are no data tree nodes for choices and cases. These are conceptual layers, not real layers, within the data tree.

Object Types in the Data Tree

Object Type	Found In Data Tree?
OBJ_TYP_ANYXML	Yes

Object Type	Found In Data Tree?
OBJ_TYP_CONTAINER	Yes
OBJ_TYP_CONTAINER (ncx:root)	No
OBJ_TYP_LEAF	Yes
OBJ_TYP_LEAF_LIST	Yes
OBJ_TYP_LIST	Yes
OBJ_TYP_CHOICE	No
OBJ_TYP_CASE	No
OBJ_TYP_USES	No
OBJ_TYP_REFINE	No
OBJ_TYP_AUGMENT	No
OBJ_TYP_RPC	No
OBJ_TYP_RPCIO	No
OBJ_TYP_NOTIF	No

The NETCONF server engine accesses individual SIL callback functions through the data tree and object tree. Each data node contains a pointer to its corresponding object node.

Each data node may have several different callback functions stored in the object tree node. Usually, the actual configuration value is stored in the database, However, **virtual data nodes** are also supported. These are simply placeholder nodes within the data tree, and usually used for non-configuration nodes, such as counters. Instead of using a static value stored in the data node, a callback function is used to retrieve the instrumentation value each time it is accessed.

2.2.4 Service Layering

All of the major server functions are supported by service layers in the 'agt' or 'ncx' libraries:

- **memory management**: macros in platform/procdefs.h are used instead of using direct heap functions.
- **queue management**: APIs in ncx/dlq.h are used for all double-linked queue management.
- **XML namespaces**: XML namespaces (including YANG module namespaces) are managed with functions in ncx/xmlns.h. An internal 'namespace ID is used internally instead of the actual URI.
- **XML parsing**: XML input processing is found in ncx/xml_util.h data structures and functions.
- **XML message processing**: XML message support is found in ncx/xml_msg.h data structures and functions.
- **XML message writing with access control**: XML message generation is controlled through API functions located in ncx/xml_wr.h. High level (value tree output) and low-level (individual tag output) XML output functions are provided, which hide all namespace, indentation, and other details. Access control is integrated into XML message output to enforce the configured data access policies uniformly for all RPC operations and notifications. The access control model cannot be bypassed by any dynamically loaded module server instrumentation code.
- **XPath Services**: All NETCONF XPath filtering, and all YANG XPath-based constraint validation, is handled with common data structures and API functions. The XPath 1.0 implementation is native to the server, and uses the object and value trees directly to generate XPath results for NETCONF and YANG purposes. NETCONF uses XPath differently than XSLT, and libxml2 XPath

processing is memory intensive. These functions are located in ncx/xpath.h, ncx/xpath1.h, and ncx/xpath_yang.h. XPath filtered <get> responses are generated in agt/agt_xpath.c.

- **Logging service**: Encapsulates server output to a log file or to the standard output, filtered by a configurable log level. Located in ncx/log.h.
- **Session management**: All server activity is associated with a session. The session control block and API functions are located in ncx/ses.h. All input, output, access control, and protocol operation support is controlled through the session control block (ses_cb_t).
- **Timer service**: A periodic timer service is available to SIL modules for performing background maintenance within the main service loop. These functions are located in agt/agt_timer.h.
- **Connection management**: All TCP connections to the netconfd server are controlled through a main service loop, located in agt/agt_ncxserver.c. It is expected that the 'select' loop in this file will be replaced in embedded systems. The default netconfd server actually listens for local <ncx-connect> connections on an AF_LOCAL socket. The openSSH server listens for connections on port 830 (or other configured TCP ports), and the netconf-subsystem thin client acts as a conduit between the SSH server and the **netconfd** server.
- **Database management**: All configuration databases use a common configuration template, defined in ncx/cfg.h. Locking and other generic database functions are handled in this module. The actual manipulation of the value tree is handled by API functions in ncx/val.h, ncx/val_util.h, agt/agt_val_parse.h, and agt/agt_val.h.
- **NETCONF operations**: All standard NETCONF RPC callback functions are located in agt/agt_ncx.c. All operations are completely automated, so there is no server instrumentation APIs in this file.
- NETCONF request processing: All <rpc> requests and replies use common data structures and APIs, found in ncx/rpc.h and agt/agt_rpc.h. Automated reply generation, automatic filter processing, and message state data is contained in the RPC message control block.
- **NETCONF error reporting:** All <rpc-error> elements use common data structures defined in ncx/rpc_err,h and agt/agt_rpcerr.h. Most errors are handled automatically, but 'description statement' semantics need to be enforced by the SIL callback functions. These functions use the API functions in agt/agt_util.h (such as agt_record_error) to generate data structures that will be translated to the proper <rpc-error> contents when a reply is sent.
- **YANG module library management:** All YANG modules are loaded into a common data structure (ncx_module_t) located in ncx/ncxtypes.h. The API functions in ncx/ncxmod.h (such as ncxmod_load_module) are used to locate YANG modules, parse them, and store the internal data structures in a central library. Multiple versions of the same module can be loaded at once, as required by YANG.

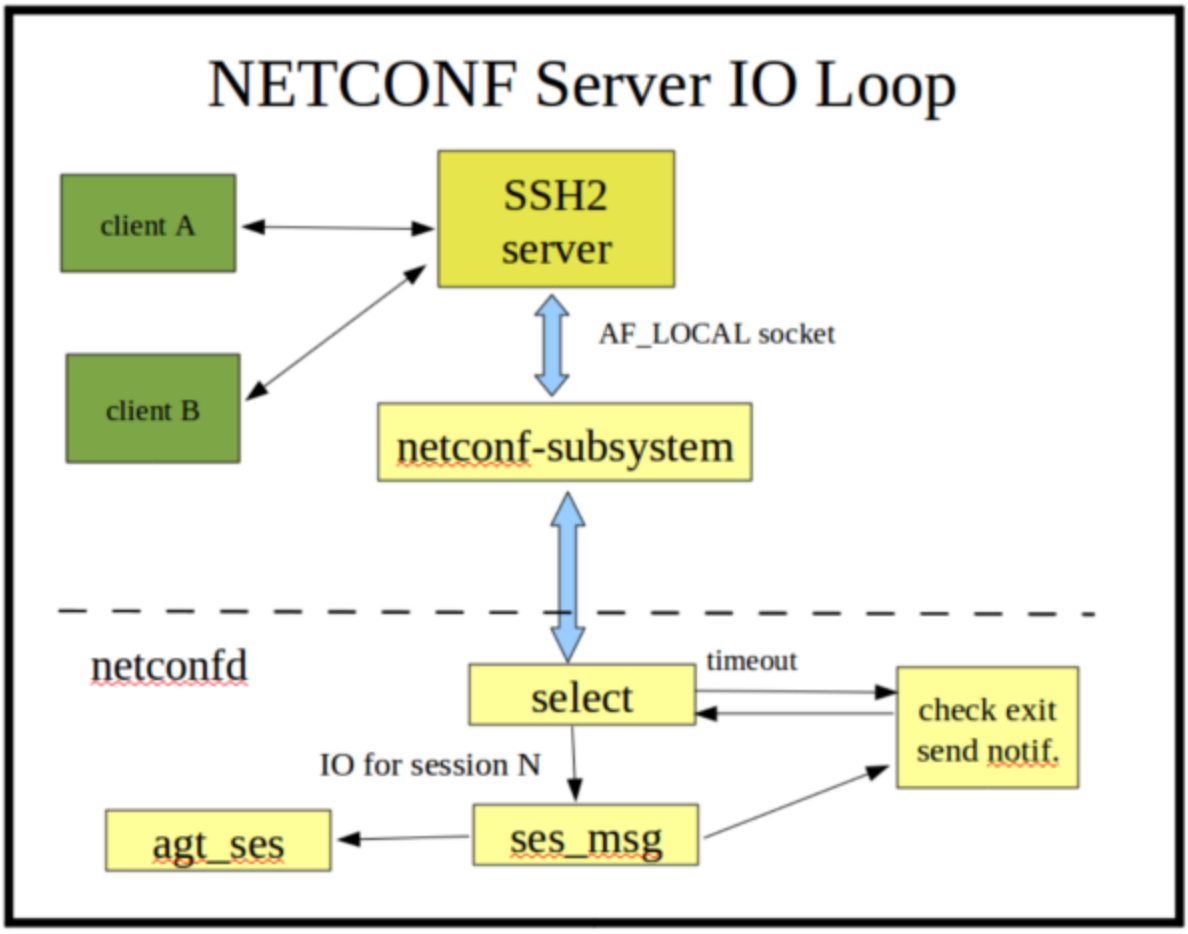
2.2.5 Session Control Block

Once a NETCONF session is started, it is assigned a session control block for the life of the session. All NETCONF and system activity in driven through this interface, so the **ncxserver** loop can be replaced in an embedded system.

Each session control block (ses_scb_t) controls the input and output for one session, which is associated with one SSH user name. Access control (see yuma-nacm.yang) is enforced within the context of a session control block. Unauthorized return data is automatically removed from the response. Unauthorized <rpc> or database write requests are automatically rejected with an 'access-denied' error-tag.

The user preferences for each session are also stored in this data structure. They are initially derived from the server default values, but can be altered with the <set-my-session> operation and retrieved with the <get-my-session> operation.

2.2.6 Main ncxserver Loop



The **ncxserver** loop does very little, and it is designed to be replaced in an embedded server that has its own SSH server:

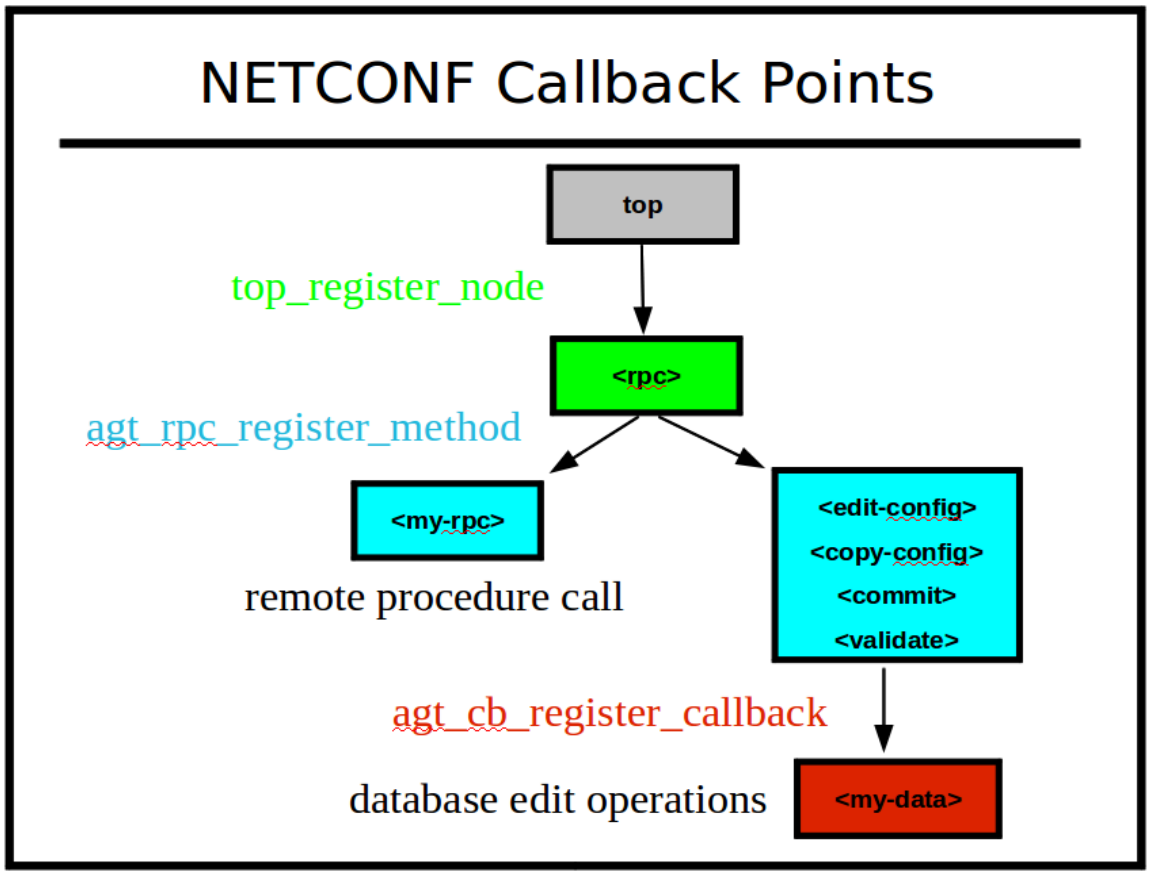
- A client request to start an SSH session results in an SSH channel being established to an instance of the **netconf-subsystem** program.
- The **netconf-subsystem** program will open a local socket (/tmp/ncxserver.sock) and send a proprietary <ncxconnect> message to the netconfd server, which is listening on this local socket with a select loop (in agt_ncxserver.c).
- When a valid <ncxconnect> message is received by **netconfd**, a new NETCONF session is created.
- After sending the <ncxconnect> message, the **netconf-subsystem** program goes into 'transfer mode', and simply passes input from the SSH channel to the **netconfd** server, and passes output from the **netconfd** server to the SSH server.
- The **ncxserver** loop simply waits for input on the open connections, with a quick timeout. Each timeout, the server checks if a reboot, shutdown, signal, or other event occurred that needs attention.

- Notifications may also be sent during the timeout check, if any events are queued for processing. The **--max-burst** configuration parameter controls the number of notifications sent to each notification subscription, during this timeout check.
- Input <rpc> messages are buffered, and when a complete message is received (based on the NETCONF End-of-Message marker), it is processed by the server and any instrumentation module callback functions that are affected by the request.

When the agt_ncxserver_run function in agt/agt_ncxserver.c is replaced within an embedded system, the replacement code must handle the following tasks:

- Call **agt_ses_new_session** in **agt/agt_ses.c** when a new NETCONF session starts.
- Call **ses_accept_input** in **ncx/ses.c** with the correct session control block when NETCONF data is received.
- Call **agt_ses_process_first_ready** in **agt/agt_ses.c** after input is received. This should be called repeatedly until all serialized NETCONF messages have been processed.
- Call **agt_ses_kill_session** in **agt/agt_ses.c** when the NETCONF session is terminated.
- The following functions are used for sending NETCONF responses, if responses are buffered instead of sent directly (streamed).
  - **ses_msg_send_buffs** in **ncx/ses_msg.c** is used to output any queued send buffers.
- The following functions need to be called periodically:
  - **agt_shutdown_requested** in **agt/agt_util.c** to check if the server should terminate or reboot
  - **agt_ses_check_timeouts** in **agt/agt_ses.c** to check for idle sessions or sessions stuck waiting for a NETCONF <hello> message.
  - **agt_timer_handler** in **agt/agt_timer.c** to process server and SIL periodic callback functions.
  - **send_some_notifications** in **agt/agt_ncxserver.c** to process some outgoing notifications.

2.2.7 SIL Callback Functions



- Top Level: The top-level incoming messages are registered, not hard-wired, in the server message processing design. The **agt_ncxserve**r module accepts the <ncxconnect> message from **netconf-subsystem**. The **agt_rpc** module accepts the NETCONF <rpc> message. Additional messages can be supported by the server using the **top_register_node** function.
- All RPC operations are implemented in a data-driven fashion by the server. Each NETCONF operation is handled by a separate function in **agt_ncx.c**. Any proprietary operation can be automatically supported, using the **agt_rpc_register_method** function.
  - Note: Once the YANG module is loaded into the server, all RPC operations defined in the module are available. If no SIL code is found, these will be dummy 'no-op' functions. This mode can be used to provide some server simulation capability for client applications under development.
- All database operations are performed in a structured manner, using special database access callback functions. Not all database nodes need callback functions. One callback function can be used for each 'phase', or the same function can be used for multiple phases. The **agt_cb_register_callback** function in **agt/agt_cb.c** is used by SIL code to hook into NETCONF database operations.

2.3 Server Operation

This section briefly describes the server internal behavior for some basic NETCONF operations.

2.3.1 Initialization

The file **netconfd/netconfd.c** contains the initial 'main' function that is used to start the server.

- The common services support for most core data structures is located in 'libncx.so'. The '**ncx_init**' function is called to setup these data structures. This function also calls the bootstrap_cli function in ncx/ncx.c, which processes some key configuration parameters that need to be set right away, such as the logging parameters and the module search path.
- Most of the actual server code is located in the 'agt' directory. The '**agt_init1**' function is called to initialize core server functions. The configuration parameters are processed, and the server profile is completed.
  - The **agt_profile_t** data structure in agt/agt.h is used to contain all the vendor-related boot-time options, such as the database target (candidate or running). The **init_server_profile** function can be edited if the Yuma default values are not desired. This will insure the proper factory defaults for server behavior are used, even if no configuration parameters are provided.
- The function **init_server_profile** in **agt/agt.c** is used to set the factory defaults for the server behavior.

The **agt_init1** function also loads the core NETCONF protocol, **netconfd** CLI, and YANG data type modules.

- Note: **netconfd** uses **yuma-netconf.yang**, not **ietf-netconf.yang** to support a data-driven implementation. The only difference is that the yuma version adds some data structures and extensions (such as ncx:root), to automate processing of all NETCONF messages.

After the core definition modules are loaded successfully, the **agt_cli_process_input** function in **agt/agt_cli.c** is called to process any command line and/or configuration file parameters that have been entered.

- Note: Any defaults set in the G module definitions will be added to the CLI parameter set. The **val_set_by_default** function in **ncx/val.c** can be used to check if the node is set by the server to the YANG default value. If not set, and the node has the YANG default value, then the client set this value explicitly. This is different than the **val_is_default** function in **ncx/val.c**, which just checks if the node contains the YANG default value.

All the configuration parameters are saved, and those that can be processed right away are handled. The **agt_cli_get_valset** function in **agt/agt_cli.c** can be used to retrieve the entire set of load-time configuration parameters.

2.3.2 Loading Modules and SIL Code

YANG modules and their associated device instrumentation can be loaded dynamically with the **--module** configuration parameter. Some examples are shown below:

```
module=foo
module=bar
module=baz@2009-01-05
module=~/mymodules/myfoo.yang
```

- The **ncxmod_find_sil_file** function in **ncx/ncxmod.c** is used to find the library code associated with the each module name. The following search sequence is followed:
  - Check the **$YUMA_HOME/target/lib** directory
  - Check each directory in the **$YUMA_RUNPATH** environment variable or **--runpath** configuration variable.

    - Check the **/usr/lib/yuma** directory
- If the module parameter contains any sub-directories or a file extension, then it is treated as a file, and the module search path will not be used. Instead the absolute or relative file specification will be used.
- If the first term starts with an environment variable or the tilde (~) character, and will be expanded first
- If the 'at sign' (@) followed by a revision date is present, then that exact revision will be loaded.
- If no file extension or directories are specified, then the module search path is checked for YANG and YIN files that match. The first match will be used, which may not be the newest, depending on the actual search path sequence.
- The **$YUMA_MODPATH** environment variable or **--modpath** configuration parameter can be used to configure one or more directory sub-trees to be searched.
- The **$YUMA_HOME** environment variable or **--yuma-home** configuration parameter can be used to specify the Yuma project tree to use if nothing is found in the currect directory or the module search path.
- The **$YUMA_INSTALL** environment variable or default Yuma install location (/usr/share/yuma/modules) will be used as a last resort to find a YANG or YIN file.

The server processes **--module** parameters by first checking if a dynamic library can be found which has an 'soname' that matches the module name. If so, then the SIL phase 1 initialization function is called, and that function is expected to call the ncxmod_load_module function.

If no SIL file can be found for the module, then the server will load the YANG module anyway, and support database operations for the module, for provisioning purposes. Any RPC operations defined in the module will also be accepted (depending on access control settings), but the action will not actually be performed. Only the input parameters will be checked, and <or> or some <rpc-error> returned.

2.3.3 Core Module Initialization

The **agt_init2** function in **agt/agt.c** is called after the configuration parameters have been collected.

- Initialize the core server code modules
- Static device-specific modules can be added to the agt_init2 function after the core modules have been initialized
- Any 'module' parameters found in the CLI or server configuration file are processed.
- The **agt_cap_set_modules** function in **agt/agt_cap.c** is called to set the initial module capabilities for the **ietf-netconf-monitoring** module

2.3.4 Startup Configuration Processing

After the static and dynamic server modules are loaded, the **--startup** (or **--no-startup**) parameter is processed by **agt_init2** in **agt/agt.c**:

- If the --startup parameter is used and includes any sub-directories, it is treated as a file and must be found, as specified.
- Otherwise, the **$YUMA_DATAPATH** environment variable or **--datapath** configuration parameter can be used to determine where to find the startup configuration file.
- If neither the **--startup** or **--no-startup** configuration parameter is present, then the data search path will be used to find the default **startup-cfg.xml**

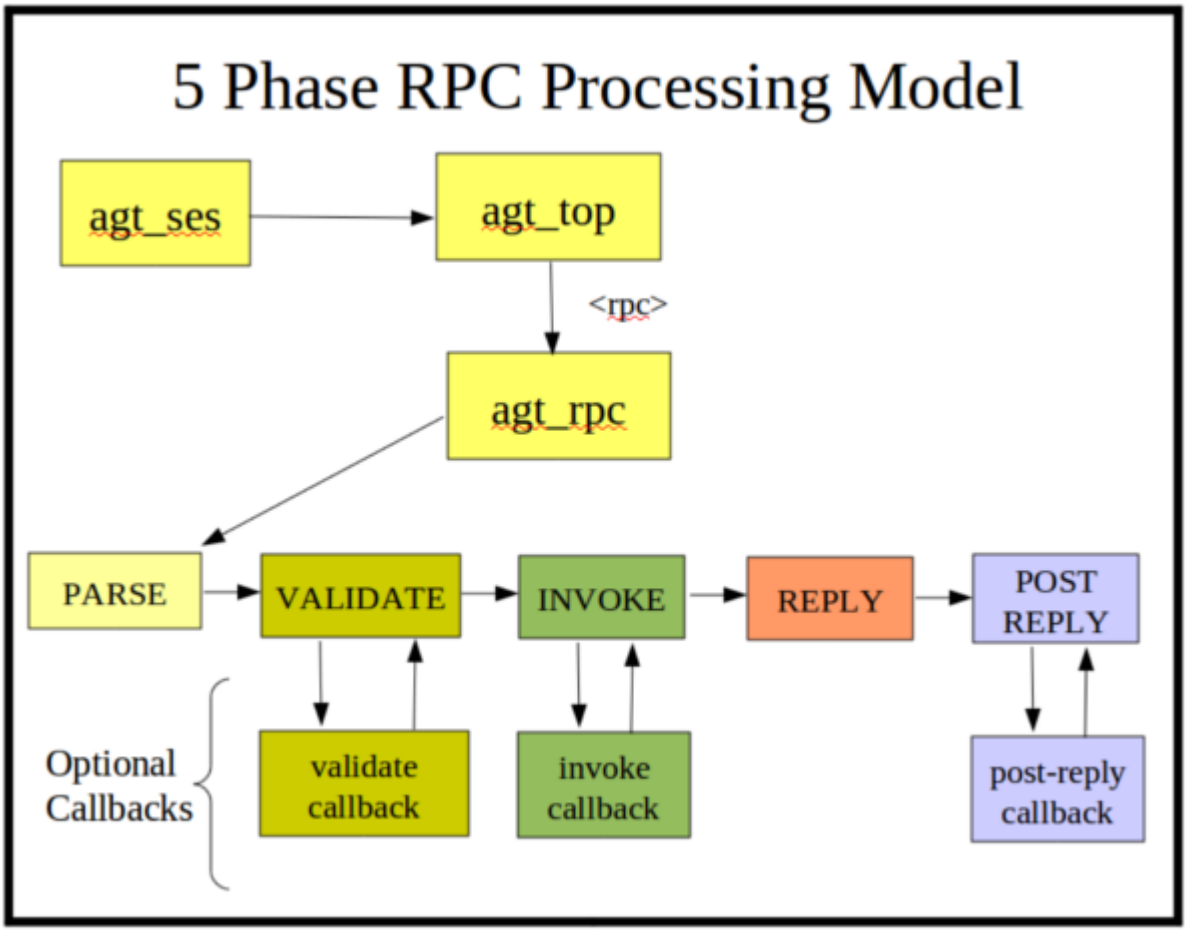
- The **$YUMA_HOME** environment variable or --yuma-home configuration parameter is checked if no file is found in the data search path. The **$YUMA_HOME/data** directory is checked if this parameter is set.
- The **$YUMA_INSTALL** environment variable or default location (/usr/share/yuma/data) is checked next, if the startup configuration is still not found.

It is a fatal error if a startup config is specified and it cannot be found.

As the startup configuration is loaded, any SIL callbacks that have been registered will be invoked for the association data present in the startup configuration file.. The edit operation will be OP_EDITOP_LOAD during this callback.

After the startup configuration is loaded into the running configuration database, all the stage 2 initialization routines are called. These are needed for modules which add read-only data nodes to the tree containing the running configuration. SIL modules may also use their 'init2' function to create factory default configuration nodes (which can be saved for the next reboot).

2.3.5 Process an Incoming <rpc> Request

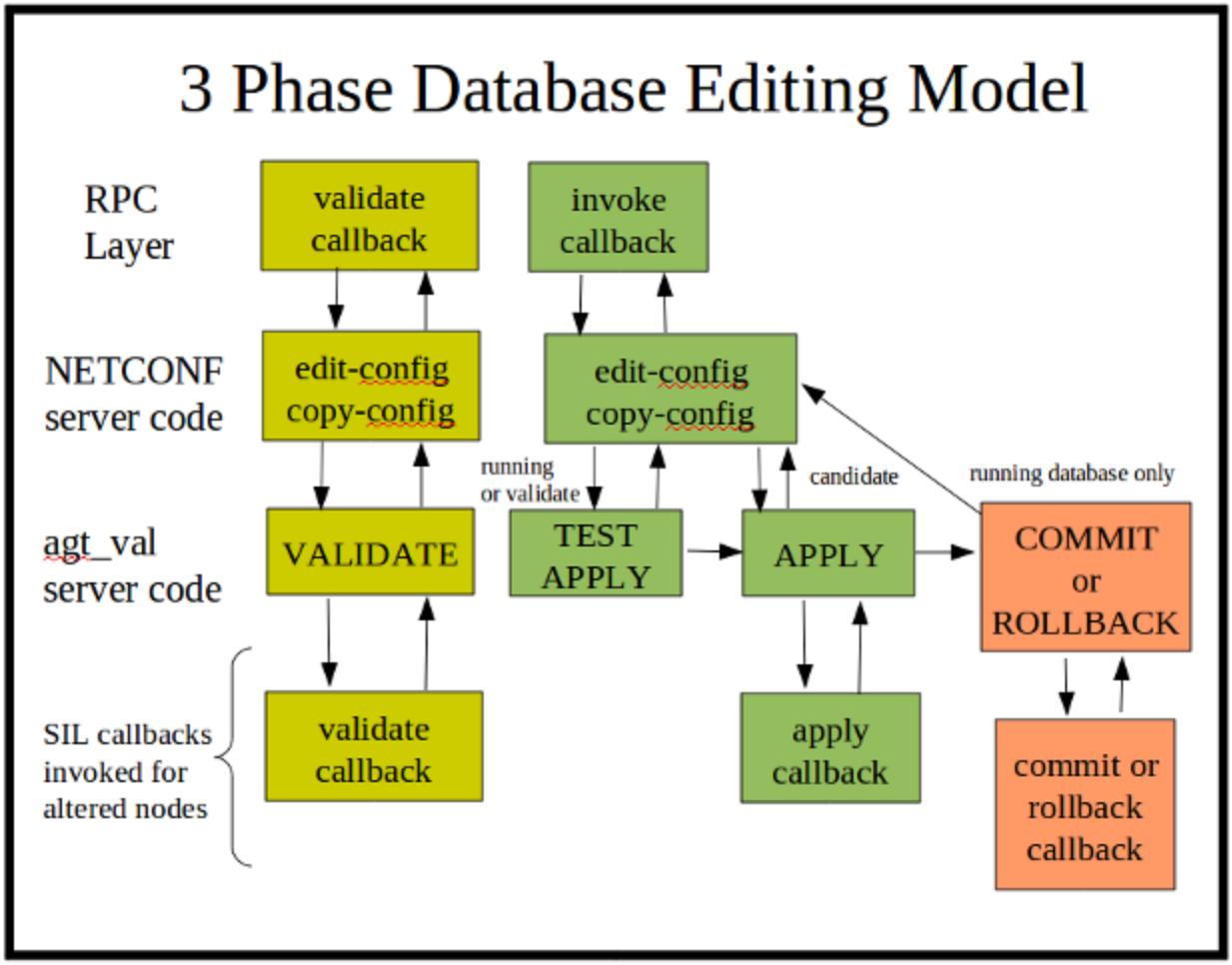


- **PARSE Phase:** The incoming buffer is converted to a stream of XML nodes, using the **xmlTextReader** functions from **libxml2**. The agt_val_parse function is used to convert the stream of XML nodes to a **val_value_t** structure, representing the incoming request according

to the YANG definition for the RPC operation. An **rpc_msg_t** structure is also built for the request.

- **VALIDATE Phase:** If a message is parsed correctly, then the incoming message is validated according to the YANG machine-readable constraints. Any description statement constraints need to be checked with a callback function. The **agt_rpc_register_method** function in **agt/agt_rpc.c** is used to register callback functions.
- **INVOKE Phase:** If the message is validated correctly, then the invoke callback is executed. This is usually the only required callback function. Without it, the RPC operation has no affect. This callback will set fields in the **rpc_msg_t** header that will allow the server to construct or stream the <rpc-reply> message back to the client.
- **REPLY Phase:** Unless some catastrophic error occurs, the server will generate an <rpc-reply> response. If any <rpc-error> elements are needed, they are generated first. If there is any response data to send, that is generated or streamed (via callback function provided earlier) at this time. Any unauthorized data (according to to the **yuma-nacm.yang** module configuration) will be silently dropped from the message payload. If there were no errors and no data to send, then an <ok> resonse is generated.
- **POST_REPLY Phase:** After the response has been sent, a rarely-used callback function can be invoked to cleanup any memory allocation or other data-model related tasks. For example, if the **rpc_user1** or **rpc_user2** pointers in the message header contain allocated memory then they need to be freed at this time.

2.3.6 Edit the Database



- **Validate Phase:** The server will determine the edit operation and the actual nodes in the target database (candidate or running) that will be affected by the operation. All of the machine-readable YANG statements which apply to the affected node(s) are tested against the incoming PDU and the target database. If there are no errors, the server will search for a SIL validate callback function for the affected node(s). If the SIL code has registered a database callback function for the node or its local ancestors, it will be invoked. This SIL callback function usually checks additional constraints that are contained in the YANG description statements for the database objects.
- **Test-Apply and Apply Phase:** If the validate phase completes without errors, then the requested changes are applied to the target database. If the target database is the running configuration, or if the edit-config 'test-option' parameter is set to 'test-then-set' (the default if **--with-validate**=true), then the test-apply phase is executed first. This is essentially the same as the real apply phase, except that changes are made to a copy of the target database. Once all objects have been altered as requested, the entire test database is validated, including all cross-referential integrity tests. If this test completes without any errors, then the procedure is repeated on the real target database.
  - Note: This phase is used for the internal data tree manipulation and validation only. It is not used to alter device behavior. Resources may need to be reserved during the SIL apply callback, but the database changes are not activated at this time.
- **Commit or Rollback Phase:** If the validate and apply phases complete without errors, then then the server will search for SIL commit callback functions for the affected node(s) in the

target database. This SIL callback phase is used to apply the changes to the device and/or network. It is only called when a commit procedure is attempted. This can be due to a <commit> operation, or an <edit-config> or <copy-config> operation on the running database.

- Note: If there are errors during the commit phase, then the backup configuration will be applied, and the server will search for a SIL callback to invoke with a 'rollback operation'. The same procedure is used for confirmed commit operations which timeout or canceled by the client.

3 SIL External Interface

Each SIL has 2 initialization functions and 1 cleanup function that must be present.

- The first initialization callback function is used to set up the configuration related objects.
- The second initialization callback is used to setup up non-configuration objects, after the running configuration has been loaded from the startup file.
- The cleanup callback is used to remove all SIL data structures and unregister all callback functions.

These are the only SIL functions that the server will invoke directly. They are generated by **yangdump** with the **--format**=c parameter, and usually do not require any editing by the developer.

Most of the work done by SIL code is through callback functions for specific RPC operations and database objects. These callback functions are registered during the initialization functions.

3.1 Stage 1 Initialization

The stage 1 initialization function is the first function called in the library by the server.

If the **netconfd** configuration parameters include a 'load' command for the module, then this function will be called during server initialization. It can also be called if the <load> operation is invoked during server operation.

This function MUST NOT attempt to access any database. There will not be any configuration databases if this function is called during server initialization. Use the 'init2' function to adjust the running configuration.

This callback function is expected to perform the following functions:

- initialize any module static data
- make sure the requested module name and optional revision date parameters are correct
- load the requested module name and revision with **ncxmod_load_module**
- setup top-level object cache pointers (if needed)
- register any RPC method callbacks with **agt_rpc_register_method**
- register any database object callbacks with **agt_cb_register_callback**
- perform any device-specific and/or module-specific initialization

Name Format:

y_<modname>_init

Input:

- modname == string containing module name to load

- revision == string containing revision date to use
  == NULL if the operator did not specify a revision.

Returns:

- operation status (0 if success)

Example function generated by **yangdump**:

```
/********************************************************************
* FUNCTION y_toaster_init
*
* initialize the toaster server instrumentation library
*
* INPUTS:
*    modname == requested module name
*    revision == requested version (NULL for any)
*
* RETURNS:
*     error status
********************************************************************/
status_t
    y_toaster_init (
        const xmlChar *modname,
        const xmlChar *revision)
{
    agt_profile_t *agt_profile;
    status_t res;

    y_toaster_init_static_vars();

    /* change if custom handling done */
    if (xml_strcmp(modname, y_toaster_M_toaster)) {
        return ERR_NCX_UNKNOWN_MODULE;
    }

    if (revision && xml_strcmp(revision, y_toaster_R_toaster)) {
        return ERR_NCX_WRONG_VERSION;
    }

    agt_profile = agt_get_profile();

    res = ncxmod_load_module(
        y_toaster_M_toaster,
        y_toaster_R_toaster,
```

```
    &agt_profile->agt_savedevQ,
    &toaster_mod);
if (res != NO_ERR) {
    return res;
}

toaster_obj = ncx_find_object(
    toaster_mod,
    y_toaster_N_toaster);
if (toaster_mod == NULL) {
    return SET_ERROR(ERR_NCX_DEF_NOT_FOUND);
}

make_toast_obj = ncx_find_object(
    toaster_mod,
    y_toaster_N_make_toast);
if (toaster_mod == NULL) {
    return SET_ERROR(ERR_NCX_DEF_NOT_FOUND);
}

cancel_toast_obj = ncx_find_object(
    toaster_mod,
    y_toaster_N_cancel_toast);
if (toaster_mod == NULL) {
    return SET_ERROR(ERR_NCX_DEF_NOT_FOUND);
}

toastDone_obj = ncx_find_object(
    toaster_mod,
    y_toaster_N_toastDone);
if (toaster_mod == NULL) {
    return SET_ERROR(ERR_NCX_DEF_NOT_FOUND);
}

res = agt_rpc_register_method(
    y_toaster_M_toaster,
    y_toaster_N_make_toast,
    AGT_RPC_PH_VALIDATE,
    y_toaster_make_toast_validate);
if (res != NO_ERR) {
    return res;
}
```

```
    res = agt_rpc_register_method(
        y_toaster_M_toaster,
        y_toaster_N_make_toast,
        AGT_RPC_PH_INVOKE,
        y_toaster_make_toast_invoke);
    if (res != NO_ERR) {
        return res;
    }

    res = agt_rpc_register_method(
        y_toaster_M_toaster,
        y_toaster_N_cancel_toast,
        AGT_RPC_PH_VALIDATE,
        y_toaster_cancel_toast_validate);
    if (res != NO_ERR) {
        return res;
    }

    res = agt_rpc_register_method(
        y_toaster_M_toaster,
        y_toaster_N_cancel_toast,
        AGT_RPC_PH_INVOKE,
        y_toaster_cancel_toast_invoke);
    if (res != NO_ERR) {
        return res;
    }

    res = agt_cb_register_callback(
        y_toaster_M_toaster,
        (const xmlChar *)"/toaster",
        (const xmlChar *)"2009-11-20",
        y_toaster_toaster_edit);
    if (res != NO_ERR) {
        return res;
    }

    /* put your module initialization code here */

    return res;
} /* y_toaster_init */
```

3.2 Stage 2 Initialization

The stage 2 initialization function is the second function called in the library by the server:

- It will only be called if the stage 1 initialization is called first, and it returns 0 (NO_ERR status).
- This function is used to initialize any needed data structures in the running configuration, such as factory default configuration, read-only counters and status objects.
- It is called after the startup configuration has been loaded into the server.
- If the <load> operation is used during server operation, then this function will be called immediately after the state 1 initialization function.

Note that configuration data structures that are loaded during server initialization **(load_running_config**) will be handled by the database callback functions registered during phase 1 initialization.

Any server-created configuration nodes should be created during phase 2 initialization (this function), after examining the explicitly-provided configuration data. For example, the top-level /nacm container will be created (by agt_acm.c) if it is not provided in the startup configuration.

This callback function is expected to perform the following functions:

- load non-configuration data structures into the server (if needed)
- initialize top-level data node cache pointers (if needed)
- load factory-default configuration data structures into the server (if needed)
- optionally save a cached pointer to a data tree node (such as the root node for the module). The **agt_create_cache** function in agt/agt_util.h is used to initialize such a module-static variable.

Name Format:

y_<modname>_init2

Returns:

- operation status (0 if success)

Example function generated by **yangdump**:

```
/*********************************************************************
* FUNCTION y_toaster_init2
*
* SIL init phase 2: non-config data structures
* Called after running config is loaded
*
* RETURNS:
*     error status
*********************************************************************/
status_t
    y_toaster_init2 (void)
{
    status_t res;
```

```
    res = NO_ERR;

    toaster_val = agt_init_cache(
        y_toaster_M_toaster,
        y_toaster_N_toaster,
        &res);
    if (res != NO_ERR) {
        return res;
    }

    /* put your init2 code here */

    return res;
} /* y_toaster_init2 */
```

3.3 Cleanup

The cleanup function is called during server shutdown. It is only called if the stage 1 initialization function is called. It will be called right away if either the stage 1 or stage 2 initialization functions return a non-zero error status.

This callback function is expected to perform the following functions:

- cleanup any module static data
- free any top-level object cache pointers (if needed)
- unregister any RPC method callbacks with **agt_rpc_unregister_method**
- unregister any database object callbacks with **agt_cb_unregister_callbacks**
- perform any device-specific and/or module-specific cleanup

Name Format:

y_<modname>_cleanup

Example function generated by **yangdump**:

```
/*********************************************************************
* FUNCTION y_toaster_cleanup
*    cleanup the server instrumentation library
*
*********************************************************************/
void
    y_toaster_cleanup (void)
{
    agt_rpc_unregister_method(
        y_toaster_M_toaster,
```

```
        y_toaster_N_make_toast);

    agt_rpc_unregister_method(
        y_toaster_M_toaster,
        y_toaster_N_cancel_toast);

    agt_cb_unregister_callbacks(
        y_toaster_M_toaster,
        (const xmlChar *)"/toaster");

    /* put your cleanup code here */

} /* y_toaster_cleanup */
```

4 SIL Callback Interface

This section briefly describes the SIL code that a developer will need to create to handle the data-model specific details. SIL functions access internal server data structures, either directly or through utility functions. Database mechanics and XML processing are done by the server engine, not the SIL code. A more complete reference can be found in section 5.

When a <rpc> request is received, the NETCONF server engine will perform the following tasks before calling any SIL:

- parse the RPC operation element, and find its associated YANG rpc template
- if found, check if the session is allowed to invoke this RPC operation
- if the RPC is allowed, parse the rest of the XML message, using the **rpc_template_t** for the RPC operation to determine if the basic structure is valid.
- if the basic structure is valid, construct an **rpc_msg_t** data structure for the incoming message.
- check all YANG machine-readable constraints, such as must, when, if-feature, min-elements, etc.
- if the incoming message is completely 'YANG valid', then the server will check for an RPC validate function, and call it if found. This SIL code is only needed if there are additional system constraints to check. For example:
  - need to check if a configuration name such as <candidate/> is supported
  - need to check if a configuration database is locked by another session
  - need to check description statement constraints not covered by machine-readable constraints
  - need to check if a specific capability or feature is enabled
- If the validate function returns a NO_ERR status value, then the server will call the SIL invoke callback, if it is present. This SIL code should always be present, otherwise the RPC operation will have no real affect on the system.
- At this point, an <rpc-reply> is generated, based on the data in the rpc_msg_t.
  - Errors are recorded in a queue when they are detected.

    - The server will handle the error reply generation for all errors it detects.
    - For SIL detected errors, the **agt_record_error** function in agt/agt_util.h is usually used to save the error details.
    - Reply data can be generated by the SIL invoke callback function and stored in the rpc_msg_t structure.
    - Replay data can be streamed by the SIL code via reply callback functions. For example, the <get> and <get-config> operations use callback functions to deal with filters, and stream the reply by walking the target data tree.
- After the <rpc-reply> is sent, the server will check for an RPC post reply callback function. This is only needed if the SIL code allocated some per-message data structures. For example, the **rpc_msg_t** contains 2 SIL controlled pointers (**rpc_user1** and **rpc_user2**). The post reply callback is used by the SIL code to free these pointers, if needed.

The database edit SIL callbacks are only used for database operations that alter the database. The validate and invoke callback functions for these operations will in turn invoke the data-model specific SIL callback functions, depending on the success or failure of the edit request.

4.1 RPC Operation Interface

All RPC operations are data-driven within the server, using the YANG rpc statement for the operation and SIL callback functions.

Any new protocol operation can be added by defining a new YANG rpc statement in a module, and providing the proper SIL code.

4.1.1 RPC Callback Initialization

The **agt_rpc_register_method** function in agt/agt_rpc.h is used to provide a callback function for a specific callback phase. The same function can be used for multiple phases if desired.

```
/* Template for RPC server callbacks
 * The same template is used for all RPC callback phases
 */
typedef status_t
    (*agt_rpc_method_t) (ses_cb_t *scb,
                         rpc_msg_t *msg,
                         xml_node_t *methnode);

extern status_t
    agt_rpc_register_method (const xmlChar *module,
                             const xmlChar *method_name,
                             agt_rpc_phase_t  phase,
                             agt_rpc_method_t method);
```

agt_rpc_register_method

Parameter	Description
module	The name of the module that contains the rpc statement
method_name	The identifier for the rpc statement
phase	AGT_PH_VALIDATE(0): validate phase AGT_PH_INVOKE(1): invoke phase AGT_PH_POST_REPLY(2): post-reply phase
method	The address of the callback function to register

4.1.2 RPC Message Header

The NETCONF server will parse the incoming XML message and construct an RPC message header, which is used to maintain state and any other message-specific data during the processing of an incoming <rpc> request.

The **rpc_msg_t** data structure in ncx/rpc.h is used for this purpose. The following table summarizes the fields:

rpc_msg_t

Field	Type	User Mode	Description
qhdr	dlq_hdr_t	none	Queue header to store RPC messages in a queue (within the session header)
mhdr	xml_msg_hdr_t	none	XML message prefix map and other data used to parse the request and generate the reply.
rpc_in_attrs	xml_attrs_t *	read write	Queue of xml_attr_t representing any XML attributes that were present in the <rpc> element. A callback function may add xml_attr_t structs to this queue to send in the reply.
rpc_method	obj_template_t *	read	Back-pointer to the object template for this RPC operation.
rpc_agt_state	int	read	Enum value (0, 1, 2) for the current RPC callback phase.
rpc_err_option	op_errop_t	read	Enum value for the <error-option> parameter. This is only set if the <edit-config> operation is in progress.
rpc_top_editop	op_editop_t	read	Enum value for the <default-operation> parameter. This is only set if the <edit-config> operation is in progress.
rpc_input	val_value_t *	read	Value tree representing the container of 'input' parameters for this RPC operation.
rpc_user1	void *	read write	Void pointer that can be used by the SIL functions to store their own message-specific data.
rpc_user2	void *	read write	Void pointer that can be used by the SIL functions to store their own message-

Field	Type	User Mode	Description
			specific data.
rpc_returncode	uint32	none	Internal return code used to control nested callbacks.
rpc_data_type	rpc_data_t	write	For RPC operations that return data, this enumeration is set to indicate which type of data is desired. **RPC_DATA_STD**: A <data> container will be used to encapsulate any returned data, within the <rpc-reply> element. **RPC_DATA_YANG**: The <rpc-reply> element will be the only container encapsulated any returned data.
rpc_datacb	void *	write	For operations that return streamed data, this pointer is set to the desired callback function to use for generated the data portion of the <rpc-reply> XML response. The template for this callback is **agt_rpc_data_cb_t,** found in agt_rpc.h
rpc_dataQ	dlq_hdr_t	write	For operations that return stored data, this queue of val_value_t structures can be used to provide the response data. Each val_value_t structure will be encoded as one of the corresponding RPC output parameters.
rpc_filter	op_filter_t	none	Internal structure for optimizing subtree and XPath retrieval operations.
rpc_need_undo	boolean	none	Internal flag to indicate if rollback-on-error is in effect dusing an <edit-config> operation.
rpc_undoQ	dlq_hdr_t	none	Queue of **rpc_undo_rec_t** structures, used to undo edits if rollback-on-error is in affect during an <edit-config> operation.
rpc_auditQ	dlq_hdr_t	none	Queue of rpc_audit_rec_t structures used internally to generate database alteration notifications and audit log entries.

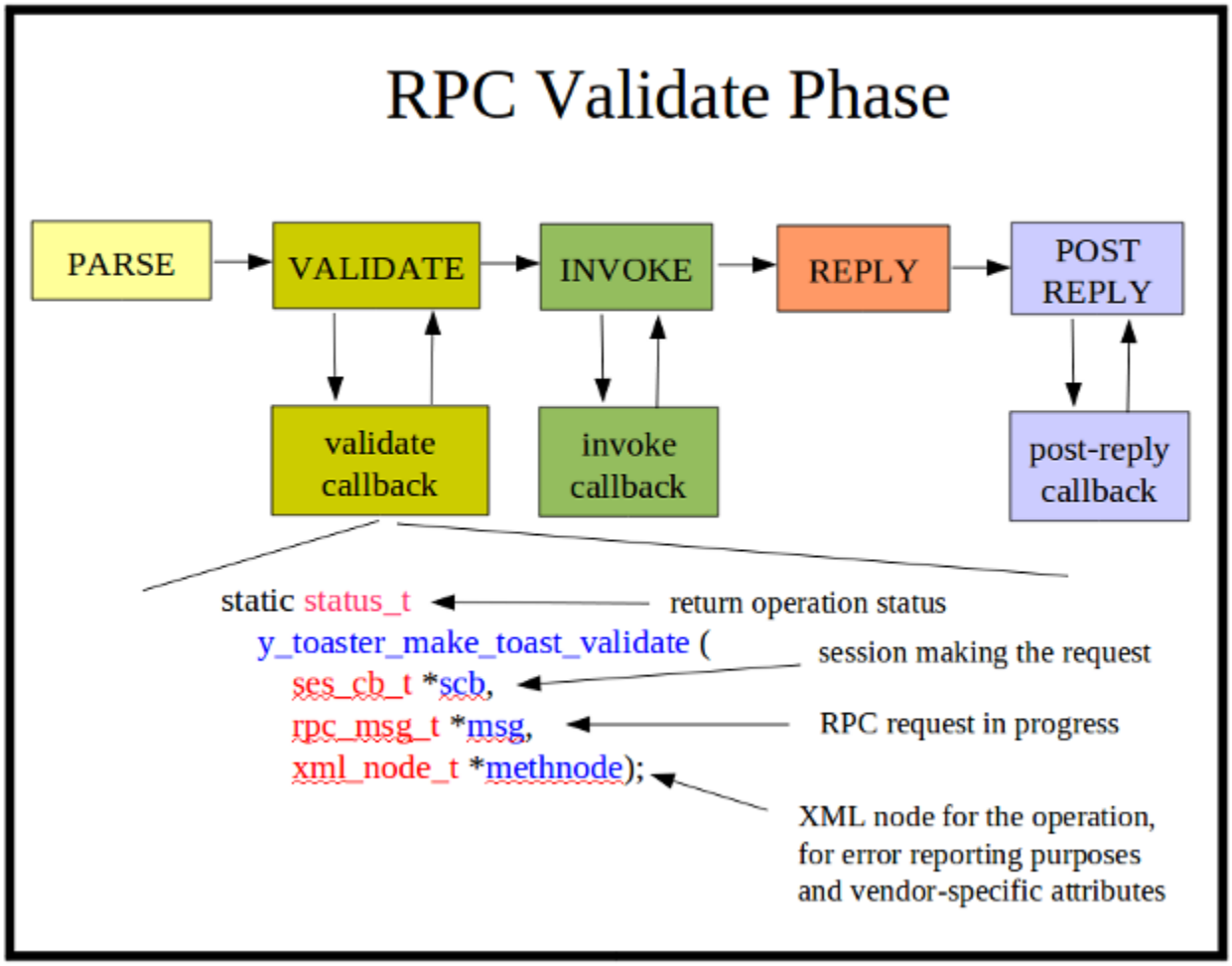
The following C code represents the **rpc_msg_t** data structure:

```
/* NETCONF Server and Client RPC Request/Reply Message Header */
typedef struct rpc_msg_t_ {
    dlq_hdr_t        qhdr;

    /* generic XML message header */
    xml_msg_hdr_t    mhdr;

    /* incoming: top-level rpc element data */
```

```
xml_attrs_t     *rpc_in_attrs;     /* borrowed from <rpc> elem */

/* incoming:
 * 2nd-level method name element data, used in agt_output_filter
 * to check get or get-config
 */
struct obj_template_t_ *rpc_method;

/* incoming: SERVER RPC processing state */
int              rpc_agt_state;        /* agt_rpc_phase_t */
op_errop_t       rpc_err_option;
op_editop_t      rpc_top_editop;
val_value_t     *rpc_input;

/* incoming:
 * hooks for method routines to save context or whatever
 */
void           *rpc_user1;
void           *rpc_user2;
uint32          rpc_returncode;   /* for nested callbacks */

/* incoming: get method reply handling builtin
 * If the rpc_datacb is non-NULL then it will be used as a
 * callback to generate the rpc-reply inline, instead of
 * buffering the output.
 * The rpc_data and rpc_filter parameters are optionally used
 * by the rpc_datacb function to generate a reply.
 */
rpc_data_t      rpc_data_type;          /* type of data reply */
void           *rpc_datacb;              /* agt_rpc_data_cb_t */
dlq_hdr_t       rpc_dataQ;       /* data reply: Q of val_value_t */
op_filter_t     rpc_filter;        /* backptrs for get* methods */

/* incoming: rollback or commit phase support builtin
 * As an edit-config (or other RPC) is processed, a
 * queue of 'undo records' is collected.
 * If the apply phase succeeds then it is discarded,
 * otherwise if rollback-on-error is requested,
 * it is used to undo each change that did succeed (if any)
 * before an error occured in the apply phase.
 */
boolean          rpc_need_undo;   /* set by edit_config_validate */
dlq_hdr_t        rpc_undoQ;       /* Q of rpc_undo_rec_t */
```

```
    dlq_hdr_t        rpc_auditQ;       /* Q of rpc_audit_rec_t */

} rpc_msg_t;
```

4.1.3 RPC Validate Callback Function



The RPC validate callback function is optional to use. Its purpose is to validate any aspects of an RPC operation, beyond the constraints checked by the server engine. Only 1 function can register for each YANG rpc statement. The standard NETCONF operations are reserved by the server engine. There is usually zero or one of these callback functions for every 'rpc' statement in the YANG module associated with the SIL code.

It is enabled with the **agt_rpc_register_method** function, within the phase 1 initialization callback function.

The **yangdump** code generator will create this SIL callback function by default. There will C comments in the code to indicate where your additional C code should be added.

The **val_find_child** function is commonly used to find particular parameters within the RPC input section, which is encoded as a **val_value_t** tree.

The **agt_record_error** function is commonly used to record any parameter or other errors. In the **libtoaster** example, there are internal state variables (**toaster_enabled** and **toaster_toasting**), maintained by the SIL code, which are checked in addition to any provided parameters.

Example SIL Function Registration

```
    res = agt_rpc_register_method(
        y_toaster_M_toaster,
        y_toaster_N_make_toast,
        AGT_RPC_PH_VALIDATE,
        y_toaster_make_toast_validate);
    if (res != NO_ERR) {
        return res;
    }
```

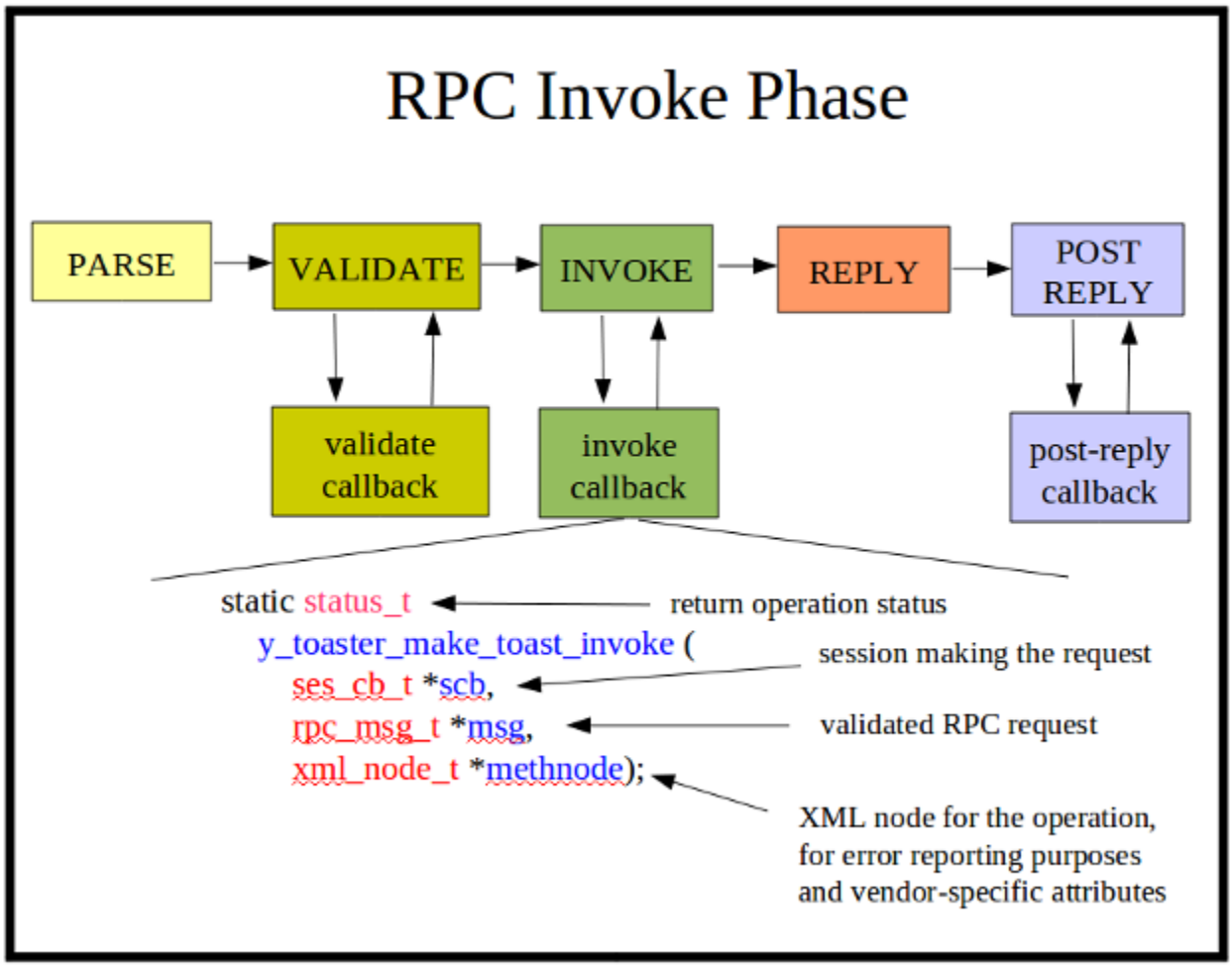
Example SIL Function:

```
/********************************************************************
* FUNCTION y_toaster_make_toast_validate
*
* RPC validation phase
* All YANG constraints have passed at this point.
* Add description-stmt checks in this function.
*
* INPUTS:
*     see agt/agt_rpc.h for details
*
* RETURNS:
*     error status
********************************************************************/
static status_t
    y_toaster_make_toast_validate (
        ses_cb_t *scb,
        rpc_msg_t *msg,
        xml_node_t *methnode)
{
    status_t res;
    val_value_t *errorval;
    const xmlChar *errorstr;
    val_value_t *toasterDoneness_val;
    val_value_t *toasterToastType_val;
```

```
    uint32 toasterDoneness;
    val_idref_t *toasterToastType;

    res = NO_ERR;
    errorval = NULL;
    errorstr = NULL;

    toasterDoneness_val = val_find_child(
        msg->rpc_input,
        y_toaster_M_toaster,
        y_toaster_N_toasterDoneness);
    if (toasterDoneness_val != NULL && toasterDoneness_val->res == NO_ERR) {
        toasterDoneness = VAL_UINT(toasterDoneness_val);
    }

    toasterToastType_val = val_find_child(
        msg->rpc_input,
        y_toaster_M_toaster,
        y_toaster_N_toasterToastType);
    if (toasterToastType_val != NULL && toasterToastType_val->res == NO_ERR) {
        toasterToastType = VAL_IDREF(toasterToastType_val);
    }

    /* added code starts here */
    if (toaster_enabled) {
        /* toaster service enabled, check if in use */
        if (toaster_toasting) {
            res = ERR_NCX_IN_USE;
        } else {
            /* this is where a check on bread inventory would go */

            /* this is where a check on toaster HW ready would go */
        }
    } else {
        /* toaster service disabled */
        res = ERR_NCX_RESOURCE_DENIED;

    }
    /* added code ends here */

    /* if error: set the res, errorstr, and errorval parms */
    if (res != NO_ERR) {
        agt_record_error(
```

```
            scb,
            &msg->mhdr,
            NCX_LAYER_OPERATION,
            res,
            methnode,
            NCX_NT_STRING,
            errorstr,
            NCX_NT_VAL,
            errorval);
    }

    return res;

} /* y_toaster_make_toast_validate */
```

4.1.4 RPC Invoke Callback Function



The RPC invoke callback function is used to perform the operation requested by the client session. Only 1 function can register for each YANG rpc statement. The standard NETCONF operations are reserved by the server engine. There is usually one of these callback functions for every 'rpc' statement in the YANG module associated with the SIL code.

The RPC invoke callback function is optional to use, although if no invoke callback is provided, then the operation will have no affect. Normally, this is only the case if the module is be tested by an application developer, using **netconfd** as a server simulator.

It is enabled with the **agt_rpc_register_method** function, within the phase 1 initialization callback function.

The **yangdump** code generator will create this SIL callback function by default. There will C comments in the code to indicate where your additional C code should be added.

The **val_find_child** function is commonly used to retrieve particular parameters within the RPC input section, which is encoded as a **val_value_t** tree. The **rpc_user1** and **rpc_user2** cache pointers in the **rpc_msg_t** structure can also be used to store data in the validation phase, so it can be immediately available in the invoke phase.

The **agt_record_error** function is commonly used to record any internal or platform-specific errors. In the **libtoaster** example, if the request to create a timer callback control block fails, then an error is recorded.

For RPC operations that return either an <ok> or <rpc-error> response, there is nothing more required of the RPC invoke callback function.

For operations which return some data or <rpc-error>, the SIL code must do 1 of 2 additional tasks:

- add a **val_value_t** structure to the **rpc_dataQ** queue in the **rpc_msg_t** for each parameter listed in the YANG rpc 'output' section.
- set the rpc_datacb pointer in the rpc_msg_t structure to the address of your data reply callback function. See the **agt_rpc_data_cb_t** definition in agt/agt_rpc.h for more details.

Example SIL Function Registration

```
    res = agt_rpc_register_method(
        y_toaster_M_toaster,
        y_toaster_N_make_toast,
        AGT_RPC_PH_INVOKE,
        y_toaster_make_toast_invoke);
    if (res != NO_ERR) {
        return res;
    }
```

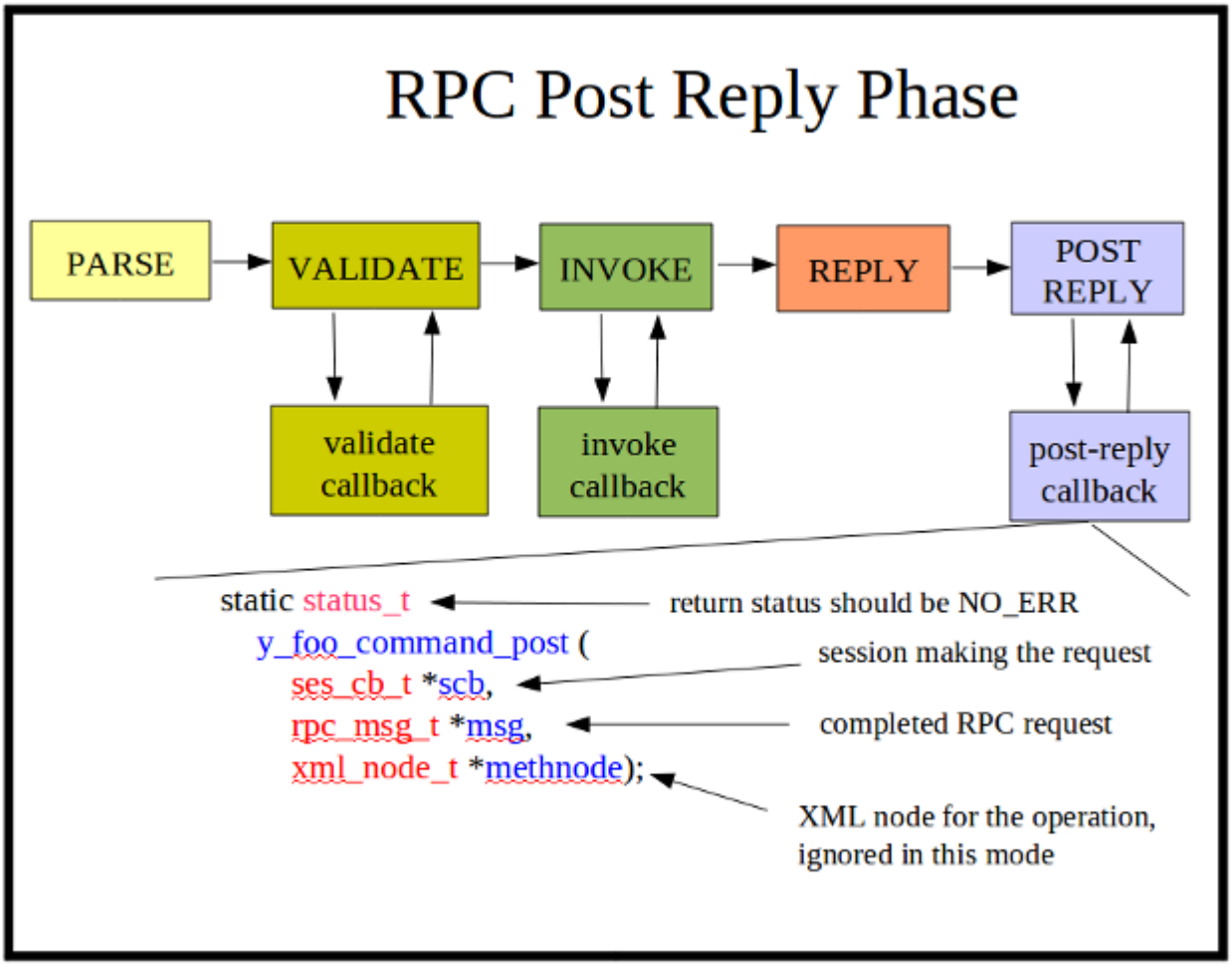
Example SIL Function:

```
/********************************************************************
* FUNCTION y_toaster_make_toast_invoke
*
* RPC invocation phase
* All constraints have passed at this point.
* Call device instrumentation code in this function.
*
* INPUTS:
*     see agt/agt_rpc.h for details
*
* RETURNS:
*     error status
********************************************************************/
static status_t
    y_toaster_make_toast_invoke (
        ses_cb_t *scb,
        rpc_msg_t *msg,
        xml_node_t *methnode)
{
    status_t res;
    val_value_t *toasterDoneness_val;
```

```
val_value_t *toasterToastType_val;
uint32 toasterDoneness;
val_idref_t *toasterToastType;

res = NO_ERR;
toasterDoneness = 0;

toasterDoneness_val = val_find_child(
    msg->rpc_input,
    y_toaster_M_toaster,
    y_toaster_N_toasterDoneness);
if (toasterDoneness_val != NULL && toasterDoneness_val->res == NO_ERR) {
    toasterDoneness = VAL_UINT(toasterDoneness_val);
}

toasterToastType_val = val_find_child(
    msg->rpc_input,
    y_toaster_M_toaster,
    y_toaster_N_toasterToastType);
if (toasterToastType_val != NULL && toasterToastType_val->res == NO_ERR) {
    toasterToastType = VAL_IDREF(toasterToastType_val);
}

/* invoke your device instrumentation code here */

/* make sure the toasterDoneness value is set */
if (toasterDoneness_val == NULL) {
    toasterDoneness = 5;   /* set the default */
}


/* arbitrary formula to convert toaster doneness to the
 * number of seconds the toaster should be on
 */
toaster_duration = toasterDoneness * 12;

/* this is where the code would go to adjust the duration
 * based on the bread type
 */

if (LOGDEBUG) {
    log_debug("\ntoaster: starting toaster for %u seconds",
              toaster_duration);
```

```
    }

    /* this is where the code would go to start the toaster
     * heater element
     */

    /* start a timer to toast for the specified time interval */
    res = agt_timer_create(toaster_duration,
                           FALSE,
                           toaster_timer_fn,
                           NULL,
                           &toaster_timer_id);
    if (res == NO_ERR) {
        toaster_toasting = TRUE;
    } else {
        agt_record_error(
            scb,
            &msg->mhdr,
            NCX_LAYER_OPERATION,
            res,
            methnode,
            NCX_NT_NONE,
            NULL,
            NCX_NT_NONE,
            NULL);
    }
    /* added code ends here */

    return res;

} /* y_toaster_make_toast_invoke */
```

4.1.5 RPC Post Reply Callback Function



The RPC post-reply callback function is used to clean up after a message has been processed. Only 1 function can register for each YANG rpc statement. The standard NETCONF operations are reserved by the server engine. This callback is not needed unless the SIL validate or invoke callback allocated some memory that needs to deleted after the <rpc-reply> is sent.

The RPC post reply callback function is optional to use. It is enabled with the **agt_rpc_register_method** function, within the phase 1 initialization callback function.

The **yangdump** code generator will not create this SIL callback function by default.

Example SIL Function Registration

```
res = agt_rpc_register_method(
    y_foo_M_foo,
    y_foo_N_command,
    AGT_RPC_PH_POST_REPLY,
    y_foo_command_post);
if (res != NO_ERR) {
    return res;
}
```

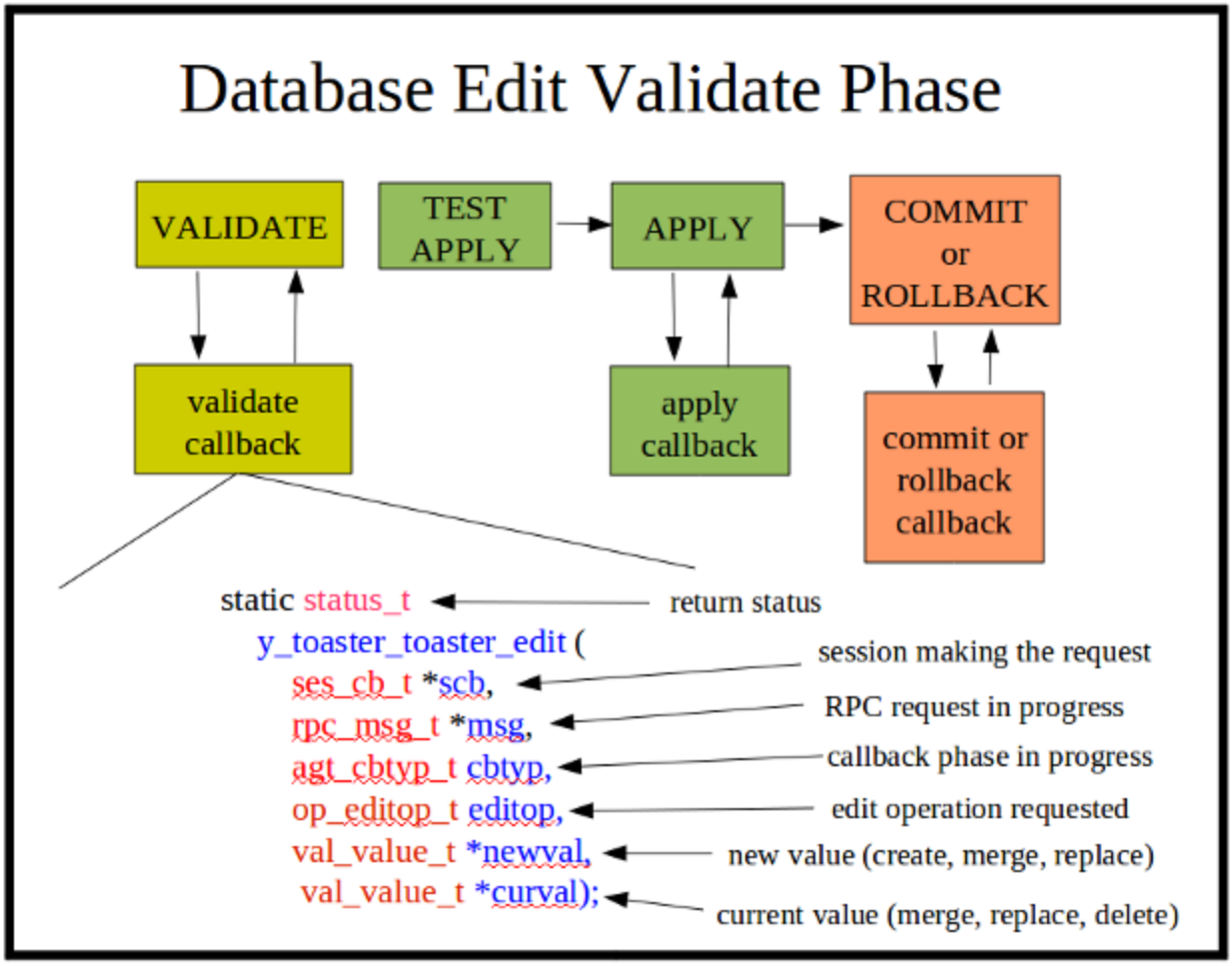
Example SIL Function:

```
/********************************************************************
* FUNCTION y_foo_command_post
*
* RPC post reply phase
*
* INPUTS:
*     see agt/agt_rpc.h for details
*
* RETURNS:
*     error status
********************************************************************/
static status_t
    y_foo_command_post (
        ses_cb_t *scb,
        rpc_msg_t *msg,
        xml_node_t *methnode)
{
    (void)scb;
    (void)methnode;
    if (msg->rpc_user1 != NULL) {
        m__free(msg->rpc_user1);
        msg->rpc_user1 = NULL;
    }
    return NO_ERR;
}  /* y_foo_command_post */
```

4.2 Database Operations

4.2.1 Database Callback Initialization

4.2.2 Database Edit Validate Callback Function



Example SIL Function:

```
/********************************************************************
* FUNCTION y_toaster_toaster_edit
*
* Edit database object callback
* Path: /toaster
* Add object instrumentation in COMMIT phase.
*
* INPUTS:
*     see agt/agt_cb.h for details
*
* RETURNS:
*     error status
********************************************************************/
static status_t
    y_toaster_toaster_edit (
```

```
        ses_cb_t *scb,
        rpc_msg_t *msg,
        agt_cbtyp_t cbtyp,
        op_editop_t editop,
        val_value_t *newval,
        val_value_t *curval)
{
    status_t res;
    val_value_t *errorval;
    const xmlChar *errorstr;

    res = NO_ERR;
    errorval = NULL;
    errorstr = NULL;

    switch (cbtyp) {
    case AGT_CB_VALIDATE:
        /* description-stmt validation here */
        break;
    case AGT_CB_APPLY:
        /* database manipulation done here */
        break;
    case AGT_CB_COMMIT:
        /* device instrumentation done here */
        switch (editop) {
        case OP_EDITOP_LOAD:
            toaster_enabled = TRUE;
            toaster_toasting = FALSE;
            break;
        case OP_EDITOP_MERGE:
            break;
        case OP_EDITOP_REPLACE:
            break;
        case OP_EDITOP_CREATE:
            toaster_enabled = TRUE;
            toaster_toasting = FALSE;
            break;
        case OP_EDITOP_DELETE:
            toaster_enabled = FALSE;
            if (toaster_toasting) {
                agt_timer_delete(toaster_timer_id);
                toaster_timer_id = 0;
                toaster_toasting = FALSE;
```

```
            y_toaster_toastDone_send((const xmlChar *)"error");
        }
        break;
    default:
        res = SET_ERROR(ERR_INTERNAL_VAL);
    }

    if (res == NO_ERR) {
        res = agt_check_cache(
            &toaster_val,
            newval,
            curval,
            editop);
    }

    if (res == NO_ERR &&
        (editop == OP_EDITOP_LOAD || editop == OP_EDITOP_CREATE)) {
        res = y_toaster_toaster_mro(newval);
    }
    break;
case AGT_CB_ROLLBACK:
    /* undo device instrumentation here */
    break;
default:
    res = SET_ERROR(ERR_INTERNAL_VAL);
}

/* if error: set the res, errorstr, and errorval parms */
if (res != NO_ERR) {
    agt_record_error(
        scb,
        &msg->mhdr,
        NCX_LAYER_CONTENT,
        res,
        NULL,
        NCX_NT_STRING,
        errorstr,
        NCX_NT_VAL,
        errorval);
}

return res;
```

```
} /* y_toaster_toaster_edit */
```

4.2.3 Database Edit Commit Callback Function

4.2.4 Database Edit Rollback Callback Function

4.2.5 Database Virtual Node Get Callback Function

4.3 Notifications

4.3.1 Notification Send Function

5 Creating a SIL

5.1 Create or Obtain the YANG File(s)

5.2 Setup the Build Environment

5.3 Generate the Initial C Source Code

5.4 Fill in the Data Model Specific Code

5.5 Build the SIL library

5.6 Install the SIL Library

5.7 Add the Module to the Server Configuration

5.8 Module Partitioning With YANG Features

5.8.1 Static Features

5.8.2 Dynamic Features

5.8.3 Configuration Parameters for Controlling Features

6 Development Environment

6.1 Programs and Libraries Needed

There are several components used in the Yuma software development environment:

- gcc compiler and linker
- ldconfig and install programs
- GNU make program
- shell program, such as bash
- Yuma development tree: the source tree containing Yuma Tools code, specified with the **$$YUMA_HOME** environment variable.
- SIL development tree: the source tree containing server instrumentation code

The following external program is used by Yuma, and needs to be pre-installed:

- **opensshd (needed by netconfd)**
  - The SSH2 server code does not link with **netconfd**. Instead, the **netconf-subsystem** program is invoked, and local connections are made to the **netconfd** server from this SSH2 subsystem.

The following program is part of Yuma Tools, and needs to be installed:

- **netconf-subsystem (needed by netconfd)**
  - The thin client sub-program that is called by **sshd** when a new SSH2 connection to the 'netconf' sub-system is attempted.
    - This program will use an AF_LOCAL socket, using a proprietary **<ncxconnect>** message, to communicate with the **netconfd** server..
    - After establishing a connection with the **netconfd** server, this program simply transfers SSH2 NETCONF channel data between **sshd** and **netconfd**.

The following program is part of Yuma Tools, and usually found within the Yuma development tree:

- **netconfd**
  - The NETCONF server that processes all protocol operations.
    - The **agt_ncxserver** component will listen for **<ncxconnect>** messages on the designated socket (/tmp/ncxserver.sock). If an invalid message is received, the connection will be dropped. Otherwise, the **netconf-subsystem** will begin passing NETCONF channel data to the netconfd server. The first message is expected to be a valid NETCONF <hello> PDU. If

The following external libraries are used by Yuma, and need to be pre-installed:

- **ncurses (needed by yangcli)**
  - character processing library needed by **libtecla**; used within **yangcli**
- **libc (or glibc, needed by all applications)**
  - unix system library
- **libssh2 (needed by yangcli)**
  - SSH2 client library, used by yangcli
- **libxml2 (needed by all applications)**
  - xmlTextReader XML parser
  - pattern support

6.2 Yuma Source Tree Layout

6.3 Platform Profile

6.4 SIL Makefile

6.4.1 Build Targets

6.4.2 Command Line Build Options

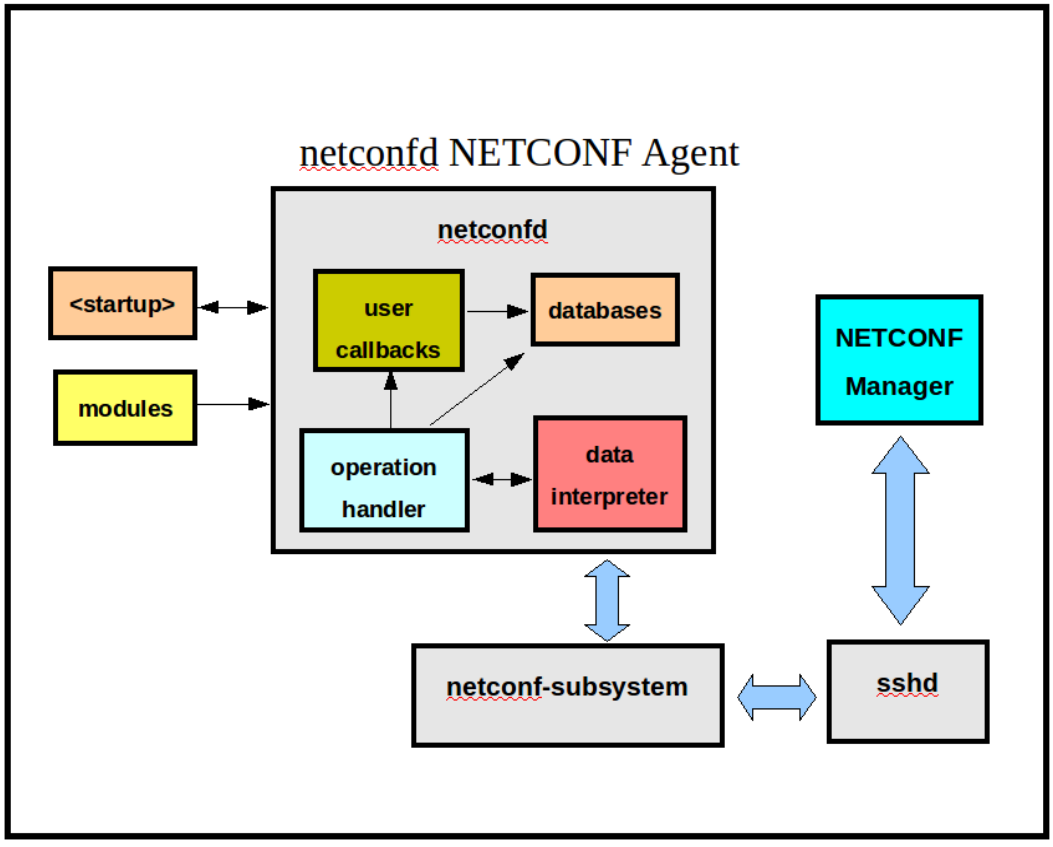
DEBUG

STATIC

MEMTRACE

MAC

7 Runtime Environment



7.1 Memory Management

7.2 Capability Management

7.3 Session Management

7.4 Database Management

7.5 Network Input and Output

7.6 Logging and Debugging

7.7 XML

7.8 XPath

7.9 YANG Data Structures

7.10 NETCONF Operations

7.11 RPC Reply Generation

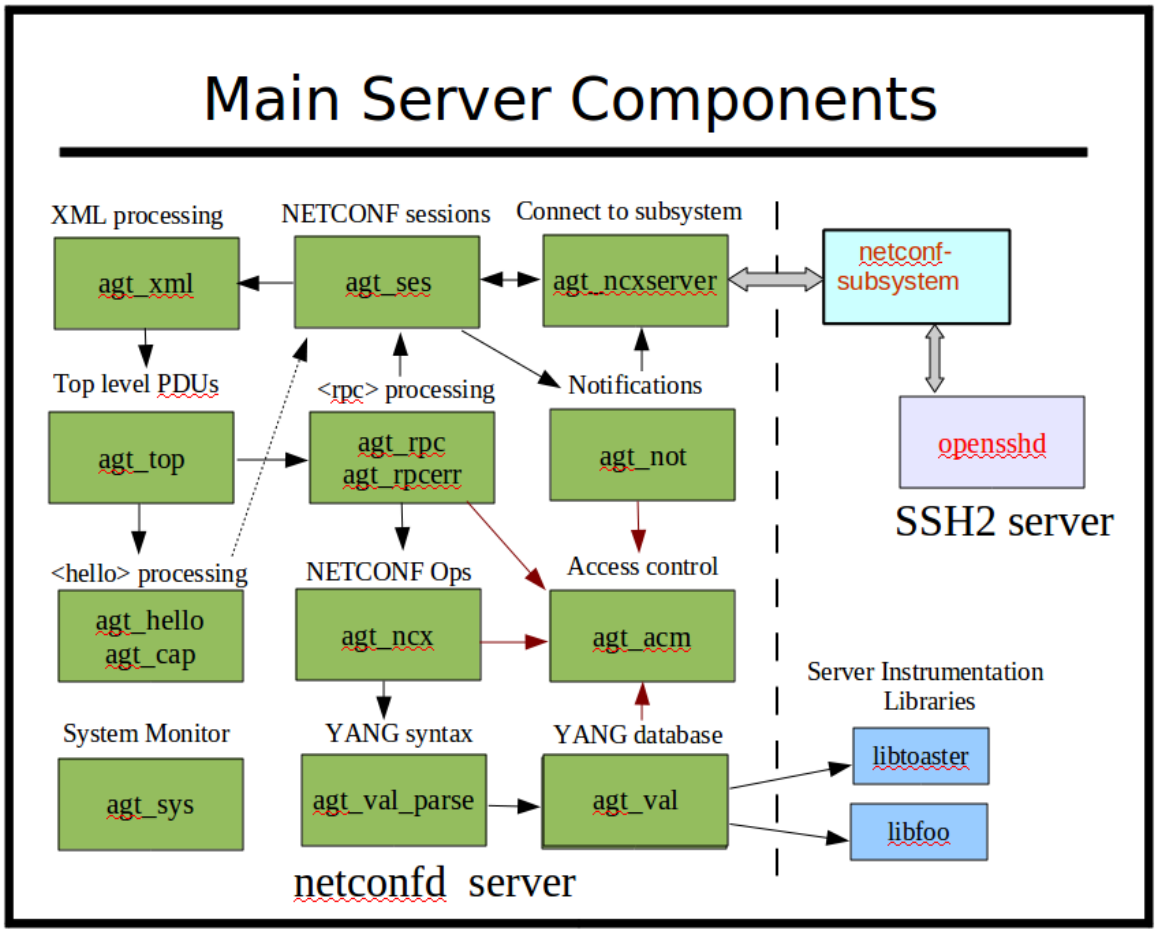
7.12 RPC Error Reporting

7.13 Notifications

7.14 Retrieval Filtering

7.15 Access Control

7.16 Message Flows



The **netconfd** server provides the following type of components:

- NETCONF session management
- NETCONF/YANG database management
- NETCONF/YANG protocol operations
- Access control configuration and enforcement
- RPC error reporting
- Notification subscription management
- Default data retrieval processing
- Database editing
- Database validation
- Subtree and XPath retrieval filtering
- Dynamic and static capability management
- Conditional bbject management (if-feature, when)
- Memory management
- Logging management
- Timer services

All NETCONF and YANG protocol operation details are handled automatically within the **netconfd** server. All database locking and editing is also handled by the server. There are callback functions available at different points of the processing model for your module specific instrumentation code to process each server request, and/or generate notifications. Everything except the 'description statement' semantics are usually handled

The server instrumentation stub files associated with the data model semantics are generated automatically with the **yangdump** program. The developer fills in server callback functions to activate the networking device behavior represented by each YANG data model.

7.17 Automation Control

The YANG language includes many ways to specify conditions for database validity, which traditionally are only documented in DESCRIPTION clauses:

7.17.1 YANG Language Extensions

There are several YANG extensions that are supported by Yuma. They are all defined in the YANG file named **ncx.yang**. They are used to 'tag' YANG definitions for some sort of automatic processing by Yuma programs. Extensions are position-sensitive, and if not used in the proper context, they will be ignored. A YANG extension statement must be defined (somewhere) for every extension used in a YANG file, or an error will be occur.

Most of these extensions apply to **netconfd** server behavior, but not all of them. For example, the **ncx:hidden** extension will prevent **yangcli** from displaying help for an object containing this extension. Also, **yangdump** will skip this object in HTML output mode.

The following table describes the supported YANG language extensions. All other YANG extension statements will be ignored by Yuma, if encountered in a YANG file:

YANG Language Extensions

extension	description
ncx:hidden;	Declares that the object definition should be hidden from all automatic documentation generation. Help will not be available for the object in **yangcli**.
ncx:metadata "*attr-type attr-name*";	Defines a qualified XML attribute in the module namespace. Allowed within an RPC input parameter. **attr-type** is a valid type name with optional YANG prefix. **attr-name** is the name of the XML attribute.
ncx:no-duplicates;	Declares that the **ncx:xsdlist** data type is not allowed to contain duplicate values. The default is to allow duplicate token strings within an **ncx:xsdlist** value.
ncx:password;	Declares that a string data type is really a password, and will not be displayed or matched by any filter.
ncx:qname;	Declares that a string data type is really an XML qualified name. XML prefixes will be properly generated by **yangcli** and **netconfd**.
ncx:root;	Declares that the container parameter is really a NETCONF database root, like <config> in the <edit-config> operations. The child nodes of this container are not specified in the YANG file. Instead, they are allowed to contain any top-level object from any YANG file supported by the server.
ncx:schema-instance;	Declares that a string data type is really an special schema instance identifier string. It is the same as an instance-identifier built-in type except the key leaf predicates are optional. For example, missing key values indicate wild cards that will match all values in **nacm** <dataRule> expressions.
ncx:secure;	Declares that the database object is a secure object. If the object is an **rpc** statement, then only the **netconfd** 'superuser' will be allowed to invoke this operation by default. Otherwise, only read access will be allowed to this object by default, Write access will only be allowed by the 'superuser', by default.
ncx:very-secure;	Declares that the database object is a very secure object.

	Only the 'superuser' will be allowed to access the object, by default.
ncx:xsdlist "*list-type*";	Declares that a string data type is really an XSD style list. **list-type** is a valid type name with optional YANG prefix. List processing within <edit-config> will be automatically handled by **netconfd**.
ncx:xpath;	Declares that a string data type is really an XPath expression. XML prefixes and all XPath processing will be done automatically by **yangcli** and **netconfd**.

8 Function Reference

8.1 ncx

8.1.1 b64.c

8.1.2 blob.c

8.1.3 bobhash.c

8.1.4 cap.c

8.1.5 cfg.c

8.1.6 cli.c

8.1.7 conf.c

8.1.8 DEF_REG.C

8.1.9 DLQ.C

8.1.10 EXT.C

8.1.11 GRP.C

8.1.12 HELP.C

8.1.13 LOG.C

8.1.14 NCX.C

8.1.15 NCXMOD.C

8.1.16 OBJ.C

8.1.17 OBJ_HELP.C

8.1.18 OP.C

8.1.19 RPC.C

8.1.20 RPC_ERR.C

8.1.21 RUNSTACK.C

8.1.22 SEND_BUFF.C

8.1.23 SES.C

8.1.24 SES_MSG.C

8.1.25 STATUS.C

8.1.26 TK.C

8.1.27 TOP.C

8.1.28 TSTAMP.C

8.1.29 TYP.C

8.1.30 VAL.C

8.1.31 VAL_UTIL.C

8.1.32 var.c

8.1.33 xml_msg.c

8.1.34 xmlns.c

8.1.35 xml_util.c

8.1.36 xml_val.c

8.1.37 xml_wr.c

8.1.38 xpath1.c

8.1.39 xpath.c

8.1.40 xpath_wr.c

8.1.41 xpath_yang.c

8.1.42 yang.c

8.1.43 yang_ext.c

8.1.44 YANG_GRP.C

8.1.45 YANG_OBJ.C

8.1.46 YANG_PARSE.C

8.1.47 YANG_TYP.C

8.2 agt

8.2.1 AGT_ACM.C

8.2.2 AGT.C

8.2.3 AGT_CAP.C

8.2.4 AGT_CB.C

8.2.5 AGT_CLI.C

8.2.6 AGT_CONNECT.C

8.2.7 AGT_EXPR.C

8.2.8 AGT_HELLO.C

8.2.9 AGT_IF.C

8.2.10 AGT_NCX.C

8.2.11 AGT_NCXSERVER.C

8.2.12 AGT_NOT.C

8.2.13 AGT_PROC.C

8.2.14 AGT_RPC.C

8.2.15 AGT_RPCERR.C

8.2.16 AGT_SES.C

8.2.17 AGT_SIGNAL.C

8.2.18 AGT_STATE.C

8.2.19 AGT_SYS.C

8.2.20 AGT_TIMER.C

8.2.21 AGT_TOP.C

8.2.22 AGT_TREE.C

8.2.23 AGT_UTIL.C

8.2.24 AGT_VAL.C

8.2.25 AGT_VAL_PARSE.C

8.2.26 AGT_XML.C

8.2.27 AGT_XPATH.C

8.3 platform

8.3.1 CURVERSION.H

8.3.2 PLATFORM.PROFILE

8.3.3 PLATFORM.PROFILE.DEPEND

8.3.4 PLATFORM.PROFILE.SIL

8.3.5 PROCDEFS.H

8.3.6 SETVERSION.SH

8.4 subsys

8.4.1 NETCONF-SUBSYSTEM.C

8.5 netconfd

8.5.1 NETCONFD.C