JACOBS
UNIVERSITY

An Implementation of the RESTCONF Protocol for OpenWrt

by

Malte Aaron Granderath

Bachelor Thesis in Computer Science

Prof. J. Schoenwaelder
(Bachelor Thesis Supervisor)

Date of Submission: May 15, 2019

Jacobs University — Computer Science

With my signature, I certify that this thesis has been written by me using only the indicates resources and materials. Where I have presented data and results, the data and results are complete, genuine, and have been obtained by me unless otherwise acknowledged; where my results derive from computer programs, these computer programs have been written by me unless otherwise acknowledged. I further confirm that this thesis has not been submitted, either in part or as a whole, for any other academic degree at this or another institution.

Signature

Bremen, 15.05.2019

Place, Date

Abstract

In recent years the open source operating system, OpenWrt, has become a popular option for replacing proprietary firmware on networking devices, whether single home routers or enterprise networks. In order to configure a system, like setting up the firewall rules, the user has to either sign in to the web interface or manually change the configuration files on the device. While this is sufficient for many home networks, in more complex networks this only allows for limited automation of configuration management and becomes time-consuming. An interface that utilizes the Hypertext Transfer Protocol (HTTP) methods and follows a standardized protocol, allows for simpler and more automated configuration changes. This paper explores the design and implementation of the RESTCONF protocol for OpenWrt systems, that was developed to solve the issues stated above.

Contents

1 Introduction **1**

2 OpenWrt **2**
2.1 History . . . 2
2.2 Adoption & Relevancy . . . 2
2.3 Configuration . . . 2
2.3.1 libuci . . . 3
2.4 Web Server . . . 4

3 RESTCONF **5**
3.1 Configuration Datastore . . . 5
3.2 Architecture . . . 6
3.3 YANG . . . 6
3.3.1 Data Modeling . . . 7
3.3.2 Tree Representation of YANG Schema Trees . . . 7
3.4 Resources & Methods . . . 7
3.4.1 HTTP Methods . . . 8
3.4.2 Request URI . . . 9
3.5 Security . . . 9

4 Design **10**
4.1 Mapping YANG and UCI . . . 10
4.2 Annotating YANG for UCI . . . 11
4.3 YANG to JSON parsable format . . . 12
4.4 Configuration Datastores . . . 14

5 Implementation **15**
5.1 Adding YINson modules . . . 15
5.2 Reading UCI as JSON . . . 16
5.3 Writing JSON as UCI . . . 17
5.4 Verifying JSON according to YANG . . . 18
5.5 Limitations . . . 19

6 OpenWrt YANG modules **20**
6.1 System . . . 20
6.2 Network . . . 20
6.2.1 Globals . . . 21
6.2.2 Interfaces . . . 21
6.2.3 Rules . . . 21
6.2.4 Routes . . . 22
6.2.5 Switches . . . 23

7 Testing **24**
7.1 Performance . . . 24
7.2 Test cases . . . 25

8 Conclusion **26**
8.1 Future work . . . 26

A YANG UCI extension module **29**

B RESTCONF-Example YANG module **30**

C OpenWrt System data model **32**

D OpenWrt Network data model **36**

E UCI testing content **48**

F Existing RESTCONF implementations **49**
F.1 JetConf 49
F.2 MD-SAL 49
F.3 Clixon 49
F.4 Limitations 49

1 Introduction

Many home routers come with proprietary firmware preinstalled, but in recent years the availability of open-source firmware for routers has been expanding. OpenWrt is one of the popular open-source solutions and supports many of the commercially available home routers. Many of these firmware, including OpenWrt, have a web user interface where users can change configurations of the router. The need for being able to configure these routers without interacting with a graphical interface has been growing. Whether one wants to change the configuration of a single home router or a whole network of them, being restricted to a graphical interface introduces limitations and requires additional time. Having the possibility of interacting through Hypertext Transfer Protocol (HTTP) methods, would pose a significant improvement.

The Internet Engineering Task Force (IETF) has developed protocols for the specific task of configurations, such as the Network Configuration Protocol (NETCONF) [10], which utilizes RPC calls to retrieve and modify the configuration. NETCONF requires both the client and the server to support the protocol and with the current trend, the importance of light-weight configuration protocols has been rising. Therefore, the IETF has developed another protocol that has a subset of the functionality of NETCONF, RESTCONF [2], but allows for usage of functionality with tools that are very common on many devices, such as `curl`. RESTCONF provides an interface, for installing, changing and deleting configurations, that is accessible via HTTP methods. HTTP is a technology that is commonly used and accessible on many systems and is, therefore, a good choice for a lightweight protocol. RESTCONF supports several different encodings and has a Data-Model-Driven API. It utilizes YANG as the data modeling language for exposing the management resources and modeling the structure of configurations [4].

The OpenWrt open source project is an embedded operating system based on Linux which is mostly used on networking devices, such as routers. It has gained adoption for devices of various categories and hardware specifications. All of its components have been optimized to run efficiently on devices that have minimal resources so that it can be used on many embedded devices. It has a fully writable and accessible filesystem and package management and can install additional applications [14]. The OpenWrt operating system implements a new approach to configuration file management which is called the Unified Configuration Interface (UCI). All of the configurations should be stored in one directory and are following the uniform configuration file format [25]. This arose from the problem of having many different configurations of all installed packages spread out in different directories on the system and not having a central solution to expose the configurations. Application and network configurations should be changed easily and quickly, and multiple tools have been implemented to solve this problem including a command line interface (CLI) and a web interface [25].

The goal of this thesis is to develop a prototype RESTCONF implementation for OpenWrt devices, that can be run on all OpenWrt supported devices, and utilizes the UCI as a configuration store. It will closely follow the specifications of the RESTCONF protocol. The implementation will be evaluated according to the performance and limitations, and further points for improvement will be stated. Additionally, specific YANG modules for the OpenWrt configuration options should be created to allow for utilizing the RESTCONF implementation on devices.

2 OpenWrt

2.1 History

The OpenWrt project was originally developed for Linksys WRT54G devices, with development starting in January 2004. Until the beginning of 2005, it was built on top of the Linksys GPL sources, but the development was then switched over to a GNU/Linux base and only modifying it with patches for the system and network interfaces. In 2007 the first stable version of OpenWrt was released under the code name ”White Russian” and the current stable release is OpenWrt 18.06 [22].

2.2 Adoption & Relevancy

OpenWrt is one of the first successful open source embedded operating systems specifically targeting networking. It enables many features on routers that would usually not be found on firmware of consumer-level routers. Unlike many of the preinstalled firmware that comes on routers, it is an accessible Linux environment that has a fully writable file system and thus allows a user to install additional packages without building and reflashing a complete firmware image. Many research teams like `Make-Wifi-Fast1`[1] have adopted OpenWrt and the improvements made from this are often merged back into the OpenWrt project [17].

2.3 Configuration

While Linux configuration files are usually stored in the `/etc/` directory and use various file formats, OpenWrt tries to unify the format and location of the configuration files. The configuration interface is called Unified Configuration Interface (UCI). Every UCI configuration file is stored in the `/etc/config` directory. All of the configurations have the same format under UCI and, due to this, can be exposed via an Application Programming Interface (API). UCI is the successor of the NV-RAM based configuration system of older OpenWrt versions [25].

UCI configurations files are divided into different sections that start with a `config` statement. The values and names in the configuration files do not need to be quoted unless they contain spaces or tabs and both single and double quotes can be used. Inside the sections, the different options are defined using the `option` keyword and a pair of type and value. The `list` keyword declares a list of values, by utilizing the same list name for several items. The sections can be so-called ”named” and ”unnamed” sections which are shown in the example below [25].

```
config example
    list example_list "1"
    list example_list "2"

config example "test"
    ...
```

[1] https://www.bufferbloat.net/projects/make-wifi-fast/wiki/

```
config-file =  *(section / CRLF)

section = ("config" 1*SP value [1*SP name]) CRLF option

option = *WSP (("option" / "list") 1*SP name 1*SP value) [(CRLF
    option) / CRLF]

name = %x22 1*allowed-name-chars %x22 / "'" 1*allowed-name-chars
    "'" / 1*allowed-name-chars

value = %x22 *(VCHAR / WSP) %x22 / ("'" *(VCHAR / WSP) "'") / *
    VCHAR

allowed-name-chars = (ALPHA / "_" / DIGIT)
```

Figure 1: The ABNF for the UCI configuration file format.

There is no current formal grammar to describe the UCI configuration language, but the above Augmented Backus-Naur Form (ABNF) (Figure 1) is derived from the description and the source code of the parser used[2]. The ABNF is based on the specification in RFC 5234 [23] and provides an exact specification of the rules for naming sections and types.

If a ”loopback” interface were to be configured in the UCI format, with the above rules, the `/etc/config/network` file would look like Figure 2.

```
config interface 'loopback_0'
    option ifname 'lo'
    option proto "static"
    option ipaddr '127.0.0.1'
    option netmask '255.0.0.0'
```

Figure 2: Example UCI loopback interface configuration.

There are different ways of how these configuration files can be modified. The files can be edited directly by changing them in the `/etc/config` directory and then reloading the service or through any of the APIs provided by `libuci`. Many packages only utilize the UCI configuration files as a read-only store and when reloading the service, using initialization scripts in `/etc/init.d`, convert the UCI configuration file to the file format that the application requires. OpenWrt even provides default shell helpers for reading and converting the files in `/lib/functions.sh`. The current standard, therefore, is a unidirectional flow from UCI configuration to the standard configuration file format of the package [15].

2.3.1 libuci

libuci is a small library for UCI written in C. It exposes multiple functions that can be used to read, write and modify the configuration files. It is installed together with the OpenWrt distribution and is the main point of access for applications to access configuration files. It

[2] https://git.openwrt.org/project/uci.git

handles the locking of the configuration files during changes and prevents write collisions. It also exposes some further additional functions that help validate values and names of sections.

Uniform Resource Identifier (URI)

Libuci utilizes a URI to identify resources in the UCI system.

```
selector = package-selector "." section-selector ["." name]
section-selector = ( name / "@" name "[" index "]" )
package-selector = name
option-selector = name
index = *DIGIT
```

Figure 3: The UCI uniform resource identifier ABNF (Extension of Figure 1).

The uniform resource identifier is used to access every possible option, list, and section in the configuration files. It follows the same specification for naming as the UCI file format and can be represented as an extension to the ABNF of the UCI format (Figure 1). Additionally, it allows for access to unnamed sections by using an array like access with an index, which is enumerated by counting the list of sections with the same type name.

The example in Figure 2 converted to the URI format of Figure 3 would give the following output:

```
network.loopback_0=interface
network.loopback_0.ifname='lo'
network.loopback_0.proto='static'
network.loopback_0.ipaddr='127.0.0.1'
network.loopback_0.netmask='255.0.0.0'
```

2.4 Web Server

OpenWrt recommends using the `uHTTPd` web server. `uHTTPd` was specifically developed for OpenWrt and, therefore, was developed towards being efficient, stable, and most suitable for lightweight tasks. It has a proper integration with UCI and its functionality is up to par with other modern web servers. It supports TLS (SSL), CGI and Lua. `uHTTPd` is single threaded, but it supports running multiple daemons on different ports [21].

While `uHTTPd` is the standard web server, it is not preinstalled on OpenWrt systems. It can be installed by using the `opkg` package manager with the package name `uhttpd` [21].

3 RESTCONF

RESTCONF is a protocol that was specifically developed for configuration management of network devices and is standardized in RFC 8040. It provides an HTTP interface, that tries to follow REST principles, for configuring data defined in the YANG data modeling language (Section 3.3) using the datastore concepts defined in Network Configuration Protocol (NETCONF). RESTCONF is compatible with servers that implement these datastores and can be co-located with a NETCONF server by protocol interactions [2].

3.1 Configuration Datastore

A configuration datastore is defined as "the complete set of configuration data that is required to get a device from its initial default state into a desired operational state" and when it is modeled by YANG (Section 3.3), it is implemented as an instantiated data tree [10]. The specification of the configuration datastores was originally defined in RFC 6241 [10], but from the experiences that were gained in utilizing the model a new specification was adopted called Network Management Datastore Architecture (NMDA) in RFC 8342. The major change was the definition of two additional datastores that store the intended and operational configuration [7].

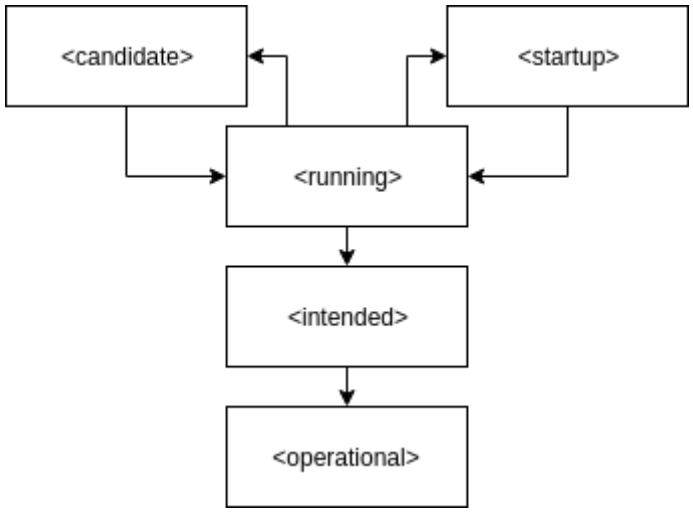

Figure 4: Architectural Model of Datastores [7]

Figure 4 represents the flow between the different datastores which are described in detail below.

- `<startup>`: The startup configuration datastore contains the configuration loaded when the system boots. It is entirely optional and is only present if the device has a separate startup and running configuration. It is copied into the `<running>` datastore [7].
- `<candidate>`: The candidate configuration datastore can be changed without impacting the currently active configuration on the device and can be committed to the devices `<running>` datastore. It should not persist between reboots and in case of

using non-volatile storage, the candidate configuration should be reset to the value of the running datastore [7].

- `<running>`: The current device configuration is stored in the running configuration datastore. It is not the exact copy of the operational configurations because it can contain configuration data that needs further transformations. It always holds a valid configuration [7].
- `<intended>`: The intended configuration datastore is read-only and holds the configuration of the running configuration datastore after all of the transformations have been applied. Therefore, whenever the `<running>` datastore is updated the `<intended>` datastore has to be updated and validated with the newest transformed configuration. If there are no transformations that have to be applied to `<running>`, then both datastores can be considered equal [7].
- `<operational>`: The operational datastore is read-only that contains configuration and state data. This might include the configuration from `<intended>`, learned configuration, the configuration provided by the system and the default values provided by the schema. Any configuration data of functionality that is not enabled or not running on the system should be omitted from this store [7].

3.2 Architecture

RESTCONF is specified to have a subset of NETCONF functionality and in case there is a NETCONF server running it has to communicate with that server to make sure there are no conflicts. However, we will only consider the case that a RESTCONF server is running and the generalized architecture of this system is represented in Figure 5.

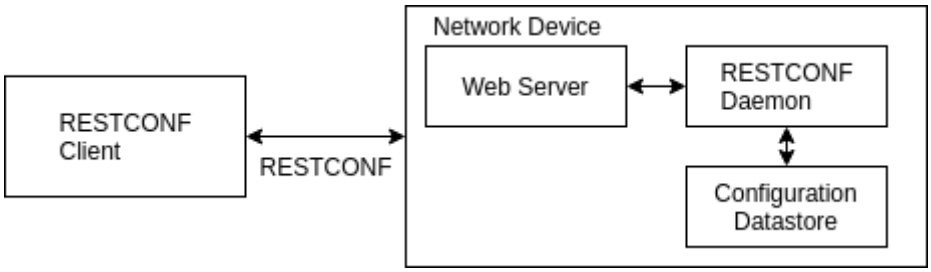

Figure 5: RESTCONF Generalized Architecture [2]

3.3 YANG

YANG is a data modeling languages that was designed to be used for configuration protocols like RESTCONF or NETCONF. It was first defined in RFC 6020 [5], but a maintenance version 1.1 has been defined in RFC 7950 [4]. Initially, only an XML encoding was defined, but additional support for the JavaScript Object Notation (JSON) encoding was added in RFC 7951 [12].

3.3.1 Data Modeling

YANG defines four different data nodes that are used to model the structure of the data:

- **Leaf Node**: contains simple data of a simple data type. It has exactly one value and no child nodes.
- **Leaf-List Node**: contains a sequence of data of one specified simple data types.
- **Container Node**: groups related nodes in a sub-tree and has no own value. It can contain children of any type.
- **List Node**: defines a sequence of list entries that each represents a container. Its defined keys uniquely identify every item.

3.3.2 Tree Representation of YANG Schema Trees

For a simpler presentation, YANG data models can be represented as a tree structure. Configuration data is represented with the prefix ”rw”, while read-only data is prefixed with ”ro”. An optional node and presence container can be represented by suffixing with ”?” and ”!” respectively. ”*” defines a list or leaf-list and the keys of a list are enclosed by ”[” and ”]”. This representation simplifies referring to specific nodes in a schema by using a schema node identifier [6].

3.4 Resources & Methods

RESTCONF is defined by a hierarchy of API resources starting with the root resource. It has a root resource discovery mechanism but no data resource discovery mechanism. The data resource identifiers and operations are derived from the YANG modules that get advertised by the server [2].

```
+---- {+restconf}
       +---- data
       | ...
       +---- operations?
       | ...
       +--ro ds?
       |  +--ro ietf-datastores:operational?
       | ...
       +--ro yang-library-version    string
```

Figure 6: The RESTCONF API resouce tree [2].

The API resource (Figure 6), located at `{+restconf}`, wraps the RESTCONF root resource for the datastore and the operation resources. It represents the accessible resources of a RESTCONF server.

- **data**: combined configuration and state data which can be retrieved by a client. The client does not have access to create or delete this resource.

- **operations**: an optional resource that provides access to the operations supported by the server.
- **yang-library-version**: identifies the revision date of the ”ietf-yang-library” YANG module used on the server.
- **ds/ietf-datastores**: an optional resource for the support of the NMDA

NMDA support for RESTCONF is specified in RFC 8527 [8]. For the support of NMDA, RESTCONF has to have the `{+restconf}/ds/ietf-datastores:operational` resource defined and the `{+restconf}/yang-library-version` should be at least 2019-01-04.

3.4.1 HTTP Methods

The RESTCONF protocol uses the HTTP methods to realize the create, read, update and delete (CRUD) operations that are requested. Its operations can be directly related to the NETCONF protocol operations and it is, therefore, compatible with the NETCONF Access Control Model (NACM) [2]. All of the following methods are required to be implemented for a RESTCONF server.

- **OPTIONS**: This method is used to discover which operations are possible for a specific resource.
- **HEAD**: This method is used to retrieve the header fields that would be returned for a GET request on a specific resource. It has to support the same query parameters as the GET method.
- **GET**: This method is used to retrieve the data and header fields for a specific resource. The only resource not supported is the operation resource. If the user does not have read privileges on the requested data, the server will not return that data.
- **POST**: POST is sent by the client to create a new data resource or to run an operation. The type of action is derived from the target resource type.
  - **Datastore**: creates a top-level configuration data resource
  - **Data**: creates a configuration data child resource
  - **Operation**: runs an operation
- **PUT**: PUT can create new data resources, like POST, but, if the resource already exists, replaces that resource. It supports data and datastore resources. A request on the datastore replaces its whole contents and a request on a data resource only replaces that resource.
- **PATCH**: The PATCH method can be used to execute a partial resource modification. It cannot be used to create a new resource and can only target existing resources. While PUT replaces an entire resource, PATCH can be used to change specific properties of that resource.

Each request can contain query parameters. The specific query parameters that are allowed depend on the resource type and target resource.

3.4.2 Request URI

The RESTCONF operation is determined from the HTTP method and the request URI. It can be conceptualized by the structure in Figure 7.

```
<method> /<root>/<path>?query
```

`<method>` : the HTTP method and mandatory
`<root>` : the RESTCONF root resource and mandatory
`<path>` : the target resource URI
`<query>` : the query parameter list

Figure 7: The RESTCONF request URI structure [2].

3.5 Security

RESTCONF requires the Transport Layer Security (TLS) protocol but does not specify a minimum HTTP version, although, HTTP 1.1 is recommended. It should also prevent excessive memory use in case of many complex requests to prevent a system disruption [2].

4 Design

While other RESTCONF implementations exist, all of them are either built on top of a configuration manager with multiple protocols, and often not lightweight, or not built for resource-constrained systems like OpenWrt (See Appendix F). The implementation should focus on being lightweight and efficient on resource-restricted systems. The implementation should be a standalone RESTCONF implementation without NETCONF interoperability and utilize the UCI system as the configuration datastore.

The UCI file format consists of a flat structure but YANG allows for modeling nested structures. As such, to be able to utilize the UCI system as the configuration datastore, a schema has to be designed that allows for representing YANG structures with the flat file format of UCI. This schema, as well as other design decisions, will be looked at in the following section.

4.1 Mapping YANG and UCI

A basic mapping between YANG and UCI can be developed by comparing the two formats and their common properties. In UCI the two types of leaf-nodes, `option` and `list`, can be used to represent leafs and leaf-lists of the YANG format. Containers can be represented as named config sections and lists as unnamed config sections. For example the following simplified YANG model (Figure 8)

```
module example {
    container state {
        leaf name {
            type string;
        }
        leaf-list namelist {
            type string;
        }
        list person {
            keys "name";
            leaf name {
                type string;
            }
        }
    }
}
```

Figure 8: YANG example module

could be represented in the UCI format, with sample data, as

```
config state state
    option name "Max"
    list namelist "Max"
    list namelist "Max Mustermann"

config person
    option name "Max"
```

While flattening of nested data is often achieved by using randomized identifiers, this method tries to maintain the readability of the configuration files such that other tools, that do not know about the structure, can still be used to modify the configuration. Although this imposes certain limitations to what YANG modules can be represented, the common format and access of the UCI is one of the major aspects of OpenWrt that should be maintained.

4.2 Annotating YANG for UCI

When representing YANG modeled data according to the above format (Section 4.1), the different names and types of sections and options are needed to access the data. These values could be directly derived from the names of YANG nodes, but depending on the structure and naming of nodes in the module, collisions between names or invalid UCI names can occur. To solve this problem and for allowing to utilize existing YANG modules, YANG extensions are defined that can be used to annotate the modules with the needed variables.

The following extensions are added with the *openwrt-uci-extension* module (Appendix A):

- `uci-package`: is used to set the UCI package name of the sub-tree
- `uci-section-type`: is used to set the UCI section type of the sub-tree. This has to be specified for a list.
- `uci-section`: is used to set the UCI section name of the sub-tree. A combination of this and `uci-section-type` is used for containers
- `uci-option`: is used to set the UCI option and list names for leafs and leaf-lists.

When importing *openwrt-uci-extension* with prefix ”ex”, Figure 8 can be annotated to follow this specification (Figure 9).

```
module example {
    ex:uci-package "restconf-example";
    container state {
        ex:uci-section-type "state";
        ex:uci-section "state";
        leaf name {
            ex:uci-option "name";
            ...
        }
        ...
        list person {
            ex:uci-section-type "person";
            ...
        }
    }
}
```

Figure 9: The annotated YANG example module.

For utilizing the annotations, the following six rules apply:

1. `uci-package`, `uci-section-type` and `uci-section` can be overridden in the same sub-tree by specifying them again. This allows for splitting up a complex module into different UCI files and sections.
2. `uci-option` can only be used to annotate leafs or leaf-lists and cannot be overridden.
3. A container has to have both `uci-section-type` and `uci-section` defined (as seen in Figure 9)
4. If just a `uci-section-type` is defined, the `uci-section` is implicitly declared as empty. Either `uci-section-type` and `uci-section` have to be defined or just `uci-section-type`.
5. A list node must have a `uci-section-type` defined but no `uci-section` (as seen in Figure 9).
6. The restrictions of the UCI format, defined in Figure 1, apply to the annotation values

4.3 YANG to JSON parsable format

Since the goal of the implementation is to run on very resource-constrained systems, every library was inspected for its package size. For reading in YANG at runtime the `libyang` library is a common choice, but due to its size, 366kB[3], and the goal of maintaining a lightweight implementation, a different approach has to be chosen. YANG can be converted to an XML format called Yang Independent Notation (YIN) and therefore read at runtime using an XML parser [4]. OpenWrt, however, only comes preinstalled with the `libjson-c` library and no XML library. The `libxml2` library was evaluated for its size in comparison to `libjson-c` and due to its 435kB[4] package size, in comparison to `libjson-c`'s 14kB[5], the decision was made to only utilize JSON. Therefore, to utilize YANG modules, they first have to be converted to be JSON parsable. While the conversion from YANG to XML is specified as YIN, there were no specifications found on converting YANG to JSON. However, a conversion from YANG to JSON can be achieved by a sequential conversion from YANG to YIN and then to JSON.

[3] https://openwrt.org/packages/pkgdata/libyang
[4] https://openwrt.org/packages/pkgdata/libxml2
[5] https://openwrt.org/packages/pkgdata/libjson-c

```
// YANG
module example {
    ...
    ex:uci-package "example";
    container state {
        ex:uci-section-type "example_type";
        ex:uci-section "example-section";
        leaf name {
            ex:uci-option "name";
            type string;
        }
    }
}

// JSON
{
   "@name": "example",
   "uci-package": {
      "@name": "example"
   },
   "container": {
      "@name": "state",
      "uci-section-type": {
         "@name": "example_type"
      },
      "uci-section": {
         "@name": "example-section"
      },
      "leaf": {
         "@name": "name",
         "type": {
            "@name": "string"
         },
         "uci-option": {
             "@name": "name"
          }
      }
   }
}
```

Figure 10: YANG to JSON conversion

Figure 10 shows the direct conversion from YANG to JSON. The issues with the output of direct conversions are:

- all of the single item objects can be mapped directly to the top object key. For example, the `@name` value in ”uci-package”.
- for traversing this JSON according to node names a further search step into each of the children has to be made to check the `@name` attribute
- keywords like `unique` or `key` that is followed by a space separated list could already be converted to an array

The JSON output in Figure 10 can be converted and modified to better suit the usage for

RESTCONF. The output of that modification is

```
{
    "type": "module",
    "uci-package": "example",
    "map": {
        "state": {
            "type": "container",
            "uci-section-type": "example_type",
            "uci-section": "example-section",
            "map": {
                "name": {
                    "type": "leaf",
                    "uci-option": "name",
                    "leaf-type": "string"
                }
            }
        }
    }
}
```

Figure 11: Modified JSON output of Figure 10

This modification makes the model easier to traverse by key and simplifies the access of different attributes. Additionally, it splits up the YANG child nodes and attributes of these nodes. The JSON could then be traversed to the leaf-node ”name” from the root object by `map["state"]/map["name"]`. From this point on, the above format will be referred to as ”YINson”.

4.4 Configuration Datastores

Since the UCI configuration files are used as the datastore, not all datastores defined in Section 3.1 can be utilized as defined. The `<startup>` and `<running>` datastores are identical as they are represented by the UCI configuration files. The UCI configuration system allows for saving ”deltas”, that is to say, changes to configuration files can be stored in an intermediate format that can, later on, be committed to modify the actual configuration file. This can be utilized in the implementation as the `<candidate>` datastore. The `<intended>` datastore can only be represented based on the individual configuration files if they are converted to another format before being used, as explained in Section 2.3. The OpenWrt system state represents the `<operational>` datastore.

5 Implementation

The goal of the RESTCONF prototype is to allow for configuring basic YANG modeled UCI configuration files[6]. Therefore, the implementation is tightly coupled with the UCI and is focused on using technologies that are accessible on OpenWrt.

The implementation is built as a binary to be executed by the Common Gateway Interface (CGI) and follows the specification in RFC 3874 [24]. It is built for the `uHTTP` server but should be compatible with any CGI enabled web server. The two libraries used are `libuci` and `libjson-c`, which are both preinstalled on OpenWrt 18.06. Since the implementation is a CGI script, it does not maintain state, and certain configurations, such as TLS, have to be enabled in the configuration of `uHTTP`, and do not fall under the scope of the implementation.

The implementation architecture differs from the one specified in Figure 5. Since RESTCONF is implemented as a CGI script there is no constantly running service but instead, the process is only started once the `uHTTP` server receives a request targeted at the script. Therefore, the datastore is represented as the `/etc/config` directory and the possibility of conversion by the `/etc/init.d` script as `/etc`. This is visualized in figure 12.

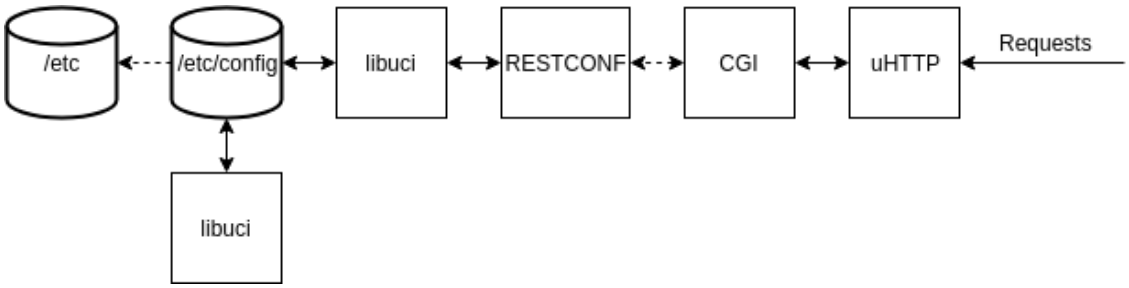

Figure 12: Implementation Architecture

5.1 Adding YINson modules

To reduce the complexity of the implementation, as much data as possible is pre-processed during the building of the binary. One example for this is the conversion of the YANG modules to YINson at build time, optimizing them for the usage with RESTCONF and including them in the code to be compiled. For this purpose, a script has been written that takes a list of file paths to YIN files and a directory where any imported YIN modules are stored and then generates a C header file that contains all of the processed modules. The conversion from YANG to JSON has been outlined in Section 4.3. This header then is included in the source of the implementation and the binary is recompiled. Additionally, this script also pre-processes the defined types (with `typedef`) and the imported types. In order to improve access to these types, the script creates a separate list of type name to type definition.

This step allows for optimizing the YANG modules for this specific use case and allows for strict checking of YANG features used. The prototype does not allow for all possible

[6] https://github.com/mgranderath/OpenWrt_RESTCONF

types and features of YANG and the script can already verify that only supported features are used in the YANG module.

5.2 Reading UCI as JSON

For returning JSON data that can be requested using GET, the existing UCI content has to be mapped to JSON based on the modeling by YANG. From the request URI that is passed to the CGI script, the YINson module can be traversed to find the targeted YANG node. At each step of traversal, the YANG node is checked for the UCI annotations, defined in Section 4.2, and the UCI path stored in an object that is passed on to the next iteration. Once the target node is reached the UCI path up to that node will be defined.

From this node, a recursive depth-first traversal is started that also reads and stores the UCI annotations at each node and passes it on to the children. There are different cases depending on which node is reached during the traversal:

- **leaf** or **leaf-list**: a UCI path should be complete pointing at a specific `option` or `list`. This value is then read using `libuci` and returned to the parent.
- **list**: There is a particular case for lists since they are represented as unnamed sections. As stated in the ABNF for the UCI URI's (Figure 3), unnamed sections can be accessed using an index. Therefore, the number of unnamed sections can be used to iterate through the list, and the index is passed down to the children on continuing the recursion on the sub-nodes.

```
module: restconf-example
  +--rw course
     +--rw name?           string
     ...
     +--rw students* [firstname lastname age]
     |  +--rw firstname     string
     |  +--rw lastname      string
     |  +--rw age           uint8
     ...
     +--rw instructor
        +--rw name?     string
        ...
```

Figure 13: Simplified *restconf-example* YANG module tree representation.

Taking the module from Figure 13 (full module in Appendix B) as an example of this process, with the URI `/data/restconf-example:course`, the following traversal will be executed:

1. **name** $\rightarrow$ read UCI value
2. **instructors** $\rightarrow$ read UCI value
3. **semester** $\rightarrow$ read UCI value
4. **students** $\rightarrow$ for i in list_length (assuming list_length = 1)
   i = 0

   (a) **firstname** → read UCI value
   (b) **lastname** → read UCI value
   (c) **age** → read UCI value
5. **instructor**
   (a) **name** → read UCI value

This method proved to be relatively quick, even reading across different UCI configuration files but could be further optimized. In case multiple options from a section are needed, batch reads of sections could be executed and then extracting the needed options, since this method currently will perform a separate read per `option` or `list` in a section. At every step of the traversal, the return value of the child will be combined with the key from YANG to produce JSON.

5.3 Writing JSON as UCI

While for reading the UCI as JSON just a traversal through YANG has to be executed, for writing the JSON as UCI, a parallel traversal of YANG and JSON has to be run. For every object in the JSON, it has to be compared to the node in YANG. Similarly to the reading of UCI as JSON, the request URI is also used to traverse to the target node. During this, the individual nodes can be checked for existence depending on whether it is a PUT or POST request.

After traversing to the target node, the root key of the sent JSON should represent the YANG resource that is targeted or its parent. From this, a depth-first traversal can be run on the combined JSON and YANG tree. Two main scenarios exist:

In case the JSON sent is not targeted at adding a list item, the content has to be flattened according to the UCI annotations and no knowledge about the existing file content has to be known except if they exist. The JSON key-value pairs will be converted to UCI path, value and type triples that can then be written after traversal. The return cases of this recursive depth-first traversal are as follows:

- **leaf**: a triple of the UCI path, JSON value and type ”option” will be returned
- **leaf-list**: a list structure of triples of the UCI path, JSON value and type ”list” will be returned
- **list**: for each item in the list the recursive traversal is continued but with the index passed in the UCI path object. The results are combined into a list structure and returned.
- **container**: the traversal is continued, but an additional triple is added of UCI path, no value and type ”container”. This is later used to create the named section.

After this step, the JSON has essentially already been flattened into a list of UCI paths to values. The changes can then be written using `libuci` and iterating through the list.

```
{
    "restconf-example:course": {
        "name": "Thesis",
        "students": [{
            "firstname": "Max",
            "lastname": "Mustermann",
            "age": 21
        }],
        "instructor": {
            "name": "Example Professor"
        }
    }
}
```

Figure 14: Example JSON request for *restconf-example* module.

For example, the request URI `/data/` with the JSON content in Figure 14 will output the following list when processed together with the *restconf-example* module:

1. ”restconf-example.course”, container
2. ”restconf-example.course.name”, option, ”Thesis”
3. ”restconf-example.@student[0].firstname”, option, ”Max”
4. ”restconf-example.@student[0].lastname”, option, ”Mustermann”
5. ”restconf-example.@student[0].age”, option, 21
6. ”restconf-example.instructor.name”, option, ”Example Professor”

In case a list item is to be added to an already existing list the above principles still apply, but instead of initializing the list index at zero, it will be initialized with the list length.

5.4 Verifying JSON according to YANG

One of the features of YANG and RESTCONF is the verification of input data according to the model. The implementation does not verify the data when reading from UCI since it is assumed that if there are other changes to the configuration files, they will be verified by the entity or tool causing those changes. However, for writing JSON as UCI, a subset of verification features has been implemented. The general verification of the JSON structure is completed while traversing through the tree.

As described in Section 5.1, the imported types and `typedef`s are added to a separate list that can then be searched for the type and definition. This is then used as a fallback in case the type declared in the YANG is not one of the YANG built-in types [4]. The verification of types is interconnected in the process of writing JSON as UCI, so whenever a leaf or leaf-list is reached during traversal, the content in the JSON is verified according to the following steps:

1. The type declaration is retrieved either directly from YANG or the list of `typedef`s and imported types

2. The content is compared to the lexical representation of the base built-in types defined in RFC 7951 [12].
3. In case a `pattern` or `range` statement is defined, the content is also verified against these statements

Verification of list and leaf-list `unique` and `key` are also implemented. For lists, the `key` and `unique` statement values are first extracted from YINson and then a simple iterative comparison is done on all items. In the case that a list item is added, the value of that item has to be compared to the already existing values, so in that case, the existing items are first read from the UCI and added into a list that is then verified. This is similar to the verification of leaf-lists, where only single values have to be compared.

5.5 Limitations

The GET, POST, PUT and DELETE methods are implemented, but PATCH should be further added. Furthermore, the implementation focuses on the `data` sub-tree of Figure 6 and functionality for other resources was only implemented sparsely. Additionally, only a limited number of built-in YANG types are supported, such as `string` or `uint8`. Built-in types like `enum` or references are not supported. Only a subset of YANG keywords are allowed but can be added in an iterative process. For this not to be a hidden issue the build-time script already detects these un-implemented features in YANG and will not process the YANG module. This, however, restricts the possible YANG modules that can be used together with the Prototype implementation.

The main restriction that is imposed by the difference in UCI and nested data structures is the nesting of a `container` or `list` in another `list`. The problem in the current implementation is to show which child item, of a list item, belongs to each list item. This can easily be represented in the following example. If a container ”nested” is added to the *students* list in the ”restconf-example” module (Appendix B) a UCI configuration file might look like this:

```
config student
    option firstname "Max"
    ...

config student
    option firstname "John"
    ...

config nested nested
```

Figure 15: Nested Container in List item.

In Figure 15 the ”nested” container can belong to either of the *student* list items. This could be solved by changing the name of the section to contain a reference to the list item it belongs to.

6 OpenWrt YANG modules

The first intention was to utilize the existing YANG modules such as *ietf-system* or *ietf-interfaces*, but after evaluation, the decision was made to write new modules that specify the available configuration options in the UCI files. The options defined in the existing modules have major differences with the available ones and, therefore, modules would have to be majorly augmented [1, 3]. Only exposing the options that are available in UCI ensures that there is consistent configuration state on startup.

6.1 System

The system module represents the `/etc/config/system` configuration file. It contains settings that apply to the basic functions of the operating system, such as logging or the hostname [19]. It consists of the system section and NTP section. The possible options and descriptions were taken from OpenWrt documentation and can be found in the complete YANG module in Appendix C [19].

```
module: openwrt-system
  +--rw system
  |  +--rw hostname?           string
  |  +--rw buffersize?         uint32
  |  +--rw conloglevel?        uint8
  |  +--rw cronloglevel?       uint8
  |  +--rw klogconloglevel?    uint8
  |  +--rw log_buffer_size?    uint32
  |  +--rw log_file?           string
  |  +--rw log_hostname?       string
  |  +--rw log_ip?             inet:ipv4-address
  |  +--rw log_port?           uint32
  |  +--rw log_prefix?         string
  |  +--rw log_proto?          string
  |  +--rw log_remote?         boolean
  |  +--rw log_size?           uint32
  |  +--rw log_trailer_null?   boolean
  |  +--rw log_type?           string
  |  +--rw urandom_seed?       string
  |  +--rw timezone?           string
  |  +--rw zonename?           string
  +--rw ntp
     +--rw enabled?         string
     +--rw enable_server?   string
     +--rw server*          string
```

6.2 Network

The network data model represents the `/etc/config/network` configuration file. The network configuration file allows for configuring interfaces, routing, rules and switches [16, 20]. The complete module can be found in Appendix D and the possible options and descriptions were taken from the OpenWrt documentation [16].

6.2.1 Globals

The module has a globals section that is used to configure interface-independent configuration options, but currently, there is only one option available which is the IPv6 ULA prefix.

```
+--rw globals
  |  +--rw ula_prefix?   inet:ipv6-prefix
```

6.2.2 Interfaces

The interfaces list in OpenWrt allows for the usage of many different protocols and different options are available depending on the other declared options. In the current implementation, these conditions cannot be declared and therefore they have to be verified manually. The following sub-tree represents the options for interface configuration:

```
+--rw interfaces
  |  +--rw interface* [ifname]
  |     +--rw ifname               string
  |     +--rw iftype?              string
  |     +--rw stp?                 openwrt-bool
  |     +--rw bridge_empty?        openwrt-bool
  |     +--rw igmp_snooping?       openwrt-bool
  |     +--rw multicast_querier?   openwrt-bool
  |     +--rw macaddr?             yang:phys-address
  |     +--rw mtu?                 int64
  |     +--rw auto?                openwrt-bool
  |     +--rw ipv6?                openwrt-bool
  |     +--rw force_link?          openwrt-bool
  |     +--rw disabled?            openwrt-bool
  |     +--rw ip4table?            string
  |     +--rw ip6table?            string
  |     +--rw proto?               string
  |     +--rw ipaddr?              inet:ipv4-address
  |     +--rw netmask?             inet:ipv4-prefix
  |     +--rw gateway?             inet:ipv4-address
  |     +--rw broadcast?           inet:ipv4-address
  |     +--rw ip6addr?             inet:ipv6-address
  |     +--rw ip6gw?               inet:ipv6-address
  |     +--rw dns*                 inet:ipv4-address
  |     +--rw layer?               uint8
```

A future improvement that would improve the organization of this section is adding specific lists per protocol to UCI or adding conditional sub-statements.

6.2.3 Rules

The rules section allows for specifying IP rules for both IPv4 and IPv6. They have the same possible configuration options, but because of different UCI `section-type` names and different types they are split into two lists,

```
+--rw rules
|  +--rw ip4rules* [mark]
|  |  +--rw mark         string
|  |  +--rw in?          string
|  |  +--rw out?         string
|  |  +--rw src?         inet:ipv4-prefix
|  |  +--rw dest?        inet:ipv4-prefix
|  |  +--rw tos?         uint64
|  |  +--rw invert?      openwrt-bool
|  |  +--rw priority?    uint64
|  |  +--rw lookup?      string
|  |  +--rw goto?        uint64
|  |  +--rw action?      string
|  +--rw ip6rules* [mark]
|     +--rw mark         string
|     +--rw in?          string
|     +--rw out?         string
|     +--rw src?         inet:ipv6-prefix
|     +--rw dest?        inet:ipv6-prefix
|     +--rw tos?         uint64
|     +--rw invert?      openwrt-bool
|     +--rw priority?    uint64
|     +--rw lookup?      string
|     +--rw goto?        uint64
|     +--rw action?      string
```

6.2.4 Routes

The routes sections can be used to declare custom routes. This is also possible for IPv4 and IPv6. Similar to the rules section there are only small differences between the IPv4 and IPv6 routes.

```
+--rw routes
|  +--rw ip4routes* [name]
|  |  +--rw name          string
|  |  +--rw interface     string
|  |  +--rw target?       inet:ipv4-address
|  |  +--rw netmask?      inet:ipv4-prefix
|  |  +--rw gateway?      inet:ipv4-address
|  |  +--rw metric?       uint64
|  |  +--rw mtu?          uint64
|  |  +--rw table?        string
|  |  +--rw source?       inet:ipv4-address
|  |  +--rw onlink?       openwrt-bool
|  |  +--rw rtype?        string
|  +--rw ip6routes* [name]
|     +--rw name          string
|     +--rw interface     string
|     +--rw target?       inet:ipv6-address
|     +--rw gateway?      inet:ipv6-address
|     +--rw metric?       uint64
|     +--rw mtu?          uint64
|     +--rw table?        string
|     +--rw source?       inet:ipv4-address
```

```
|      +--rw onlink?       openwrt-bool
|      +--rw rtype?        string
```

6.2.5 Switches

The final section of the network configuration files is the configuration of the switches. There are 3 different sections that are part of this configuration:

- **switch**: can be used to change general switch related configuration options, such as the address-resolution table's aging time [16].
- **switch_vlan**: can be used to configure the VLAN configuration of a switch [16]
- **switch_port**: can be used to configure the ports of a switch [16]

The sub-tree representing these configuration options is below.

```
+--rw switches
    +--rw switch* [name]
    |  +--rw name                    string
    |  +--rw reset?                  openwrt-bool
    |  +--rw enable_vlan?            openwrt-bool
    |  +--rw enable_mirror_rx?       openwrt-bool
    |  +--rw enable_mirror_tx?       openwrt-bool
    |  +--rw mirror_monitor_port?    uint64
    |  +--rw mirror_source_port?     uint64
    |  +--rw arl_age_time?           uint64
    |  +--rw igmp_snooping?          openwrt-bool
    |  +--rw igmp_v3?                openwrt-bool
    +--rw switch_vlan* [device vlan]
    |  +--rw device    string
    |  +--rw vlan      uint64
    |  +--rw vid?      uint64
    |  +--rw ports*    string
    +--rw switch_port* [device port]
       +--rw device           string
       +--rw port             uint64
       +--rw pvid?            uint64
       +--rw enable_eee?      openwrt-bool
       +--rw igmp_snooping?   openwrt-bool
       +--rw igmp_v3?         openwrt-bool
```

7 Testing

To test the implementation and its features, a YANG module was developed called *restconf-example*. The module can be found in Appendix B and can be used to test almost all implemented features. The module was changed with every new feature implementation and in the same manner, new tests were added. Additionally, memory usage was tested using the `valgrind` utility. The development and testing was done on a virtual environment with 64MB of RAM and x86_64 architecture. This was used because the virtual hardware and system limitations can easily be changed for a virtual environment.

7.1 Performance

Since one of the goals is for the implementation to run on very resource-constrained systems, to evaluate this goal the performance is tested. The implementation depends on a couple of libraries and the total package size of dependencies can be found in Table 1.

Library	**Size**
libjson-c	14KB
libuci	13KB
libubox	16KB
Total	43KB / 0.043MB

Table 1: Total package library size

The total sum of libraries is kept at an absolute minimum with not even requiring 0.05MB of the flash. In the official documentation, it is stated that a minimum of 4MB of Flash is needed for installation of OpenWrt and a minimum of 32MB of RAM is recommended [18].

The binary size when compiling with the *restconf-example*, *openwrt-system* and *openwrt-network* modules and using the `gcc` (version 7.4.0) compiler is 0,2943MB. A future possibility to reduce the size of the binary would be to read the YINson at runtime from the filesystem instead of importing it at build-time but this is a tradeoff between time and space.

For testing the memory consumption and execution time of the CGI script, it was tested on the *restconf-example* module with the UCI file contents for the GET, POST and PUT request that can be found in Appendix E. The following cases and results were found utilizing the `/usr/bin/time` utility and the `valgrind` *massif* tool to detect the maximum heap size:

- GET `/data/restconf-example:course` → Elapsed time: 0.00s and maximum heap size: 1100 kilobytes (kbytes)
- POST `/data/restconf-example:course` → Elapsed time: 0.07s and maximum heap: 1100 kbytes
- PUT `/data/restconf-example:course` (Replacing all content) → Elapsed time: 0.07s and maximum heap: 1100 kbytes

- DELETE `/data/restconf-example:course` → Elapsed time: 0.01s and maximum heap size: 1100 kbytes

The average memory consumption for any execution is around 1100 kbytes (1.1 MB) and is well below the 32MB RAM that are stated as a minimum for a system. When increasing the number of list items to be read to 30 the *GET* method has a maximum heap size of 1.2MB.

7.2 Test cases

For demonstrating and testing functionality a couple of basic test cases are described in this section, but tests for almost all cases were written and can be found in the implementation source. The test cases are one complete flow. They depend on the order they are executed. The following test cases are implemented:

1. POST — Error on malformed JSON content
2. POST — Valid content for creation
3. GET — Check that expected content exists
4. DELETE — Delete leaf-list node
5. POST — Create new leaf-list
6. DELETE — Delete leaf
7. POST, PUT — Check range statement
8. POST, PUT — Check valid leaf created
9. POST, PUT — Check duplicate and unique for lists
10. DELETE — Deleting specific list item
11. POST, PUT — Verify with pattern

The above test cases are a subset of all tests implemented and since they are testing the complete system more functionality than stated is tested with every execution.

8 Conclusion

The accessibility of configuring OpenWrt utilizing a lightweight and accessible protocol poses benefits for both enterprises and single users. An implementation of RESTCONF has been designed by closely following the official specification while following design principles that make OpenWrt different from other operating systems.

Close attention was paid to using only pre-installed libraries and the architecture of the implementation allows for utilization with multiple different web servers. One of the significant aspects of this implementation is the focus on using the UCI as a configuration datastore. This maintains a consistent configuration state but also imposes some limitations such as the configurable options or YANG modules that can be used.

While this prototype implementation only implements a subset of the defined features of RESTCONF and YANG, it has been proven to be an efficient base that can be used to extend the implementation further. It is implemented in the C language and can utilize the shared libraries. It utilizes only 12.5% of the flash that is stated as a minimum requirement of any OpenWrt system and its peak memory consumption during testing was also never higher than 12.5% of the minimum memory requirement. Additionally, it was implemented as a CGI script and as such only runs whenever a request is sent. This shows that it can be used on almost every supported device without hindering other networking essential applications.

8.1 Future work

There are many ways the prototype implementation can be improved. Many of the features such as the PATCH method have not been implemented due to time constraints. Any additional feature implementations would, therefore, improve the implementation.

One of the major challenges encountered during the implementation was the lack of documentation and specification of the UCI configuration options. There are different variants of documentation of the options that can be found online. This also means that together with work from the OpenWrt community the developed YANG modules could be improved with more options or descriptions and can act as this standardized documentation.

The main reason that the *ietf-system* and *ietf-interfaces* modules were not used is the difference in configurable options between UCI and the schemas. Therefore, adapting the configuration options available in UCI, towards the IETF standards, would generally allow for a more convenient usage with configuration protocols.

References

[1] A. Bierman and M. Bjorklund. *A YANG Data Model for System Management. RFC 7317*. Tech. rep. Aug. 2014. DOI: 10.17487/rfc7317. URL: https://doi.org/10.17487/rfc7317.

[2] A. Bierman, M. Bjorklund, and K. Watsen. *RESTCONF Protocol. RFC 8040.* Tech. rep. Jan. 2017. DOI: 10.17487/rfc8040.

[3] M. Bjorklund. *A YANG Data Model for Interface Management. RFC 8343*. Tech. rep. Mar. 2018. DOI: 10.17487/rfc8343. URL: https://doi.org/10.17487/rfc8343.

[4] *The YANG 1.1 Data Modeling Language. RFC 7950.* Tech. rep. Aug. 2016. DOI: 10.17487/rfc7950.

[5] *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020.* Tech. rep. Oct. 2010. DOI: 10.17487/rfc6020.

[6] M. Bjorklund. *YANG Tree Diagrams. RFC 8340.* Tech. rep. Mar. 2018. DOI: 10.17487/rfc8340.

[7] M. Bjorklund et al. *Network Management Datastore Architecture (NMDA). RFC 8342.* Tech. rep. Mar. 2018. DOI: 10.17487/rfc8342.

[8] M. Bjorklund et al. *RESTCONF Extensions to Support the Network Management Datastore Architecture*. RFC 8527. Feb. 2019. URL: https://www.rfc-editor.org/rfc/authors/rfc8527.txt.

[9] *clicon/clixon: Clixon automatically generates interactive CLI, NETCONF, RESTCONF and embedded databases with transaction support from a YANG specification.* Feb. 12, 2019. URL: https://github.com/clicon/clixon (visited on 02/12/2019).

[10] *Network Configuration Protocol (NETCONF). RFC 6241.* Tech. rep. June 2011. DOI: 10.17487/rfc6241.

[11] *jetconf architecture - GitLab*. Feb. 12, 2019. URL: https://gitlab.labs.nic.cz/labs/jetconf/wikis/jetconf-architecture (visited on 02/12/2019).

[12] L. Lhotka. *JSON Encoding of Data Modeled with YANG. RFC 7951.* Tech. rep. Aug. 2016. DOI: 10.17487/rfc7951.

[13] *OpenDaylight Controller:MD-SAL:Explained - OpenDaylight Project*. Feb. 13, 2019. URL: https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Explained (visited on 02/13/2019).

[14] *OpenWrt Project*. URL: https://openwrt.org/.

[15] *OpenWrt Project: Init Scripts*. Feb. 20, 2019. URL: https://openwrt.org/docs/techref/initscripts (visited on 02/20/2019).

[16] *OpenWrt Project: Network basics /etc/config/network*. Apr. 26, 2019. URL: https://openwrt.org/docs/guide-user/base-system/basic-networking (visited on 04/26/2019).

[17] *OpenWrt Project: Reasons to Use OpenWrt*. Feb. 20, 2019. URL: https://openwrt.org/reasons_to_use_openwrt (visited on 02/20/2019).

[18] *OpenWrt Project: Supported devices*. Apr. 26, 2019. URL: https://openwrt.org/supported_devices (visited on 04/26/2019).

[19] *OpenWrt Project: System configuration /etc/config/system*. Apr. 25, 2019. URL: https://openwrt.org/docs/guide-user/base-system/system_configuration (visited on 04/25/2019).

[20] *OpenWrt Project: UCI networking options cheatsheet*. Apr. 26, 2019. URL: https://openwrt.org/docs/guide-user/network/ucicheatsheet (visited on 04/26/2019).

[21] *OpenWrt Project: uHTTPd webserver*. Jan. 29, 2019. URL: https://openwrt.org/docs/guide-user/services/webserver/http.uhttpd (visited on 01/29/2019).

[22] *OpenWrt Version History*. Jan. 2019. URL: https://openwrt.org/about/history.

[23] P. Overell. *Augmented BNF for Syntax Specifications: ABNF. RFC 5234.* Tech. rep. Jan. 2008. DOI: 10.17487/rfc5234.

[24] D. Robinson and K. Coar. *The Common Gateway Interface (CGI) Version 1.1. RFC 3875*. Tech. rep. Oct. 2004. DOI: 10.17487/rfc3875. URL: https://doi.org/10.17487/rfc3875.

[25] *The UCI System*. Oct. 2018. URL: https://openwrt.org/docs/guide-user/base-system/uci.

A YANG UCI extension module

```
module openwrt-uci-extension {
    namespace "urn:jacobs:yang:openwrt-uci";
    prefix "ouci";

    contact "Malte Granderath <m.granderath@jacobs-university.de
       >";
    revision 2019-04-24 {
      description "initial revision";
    }

    extension uci-package {
        argument name;
    }

    extension uci-section {
        argument name;
    }

    extension uci-option {
        argument name;
    }

    extension uci-section-type {
        argument name;
    }
}
```

B RESTCONF-Example YANG module

```
module restconf-example {
  namespace "http://example.org/example-last-modified";
  prefix "ex";

  import openwrt-uci-extension {
    prefix uci;
  }

  typedef grade {
    type uint8 {
      range "0..100";
    }
  }

  typedef email {
    type string {
      pattern "[A-Za-z0-9]*@university.de";
    }
  }

  uci:uci-package "restconf-example";
  container course {
    uci:uci-section "course";
    uci:uci-section-type "course";

    leaf name {
      uci:uci-option "name";
      type string;
      description "name of the course";
    }

    leaf-list instructors {
      uci:uci-option "instructors";
      type string;
      description "list of names of instructors";
    }

    leaf semester {
      uci:uci-option "semester";
      type uint8 {
        range "1..6";
      }
    }

    list students {
      uci:uci-section-type "student";

      key "firstname lastname age";
      leaf firstname {
        uci:uci-option "firstname";
        type string;
      }
```

```
      leaf lastname {
        uci:uci-option "lastname";
        type string;
      }

      leaf age {
        uci:uci-option "age";
        type uint8 {
          range "0..120";
        }
      }

      leaf major {
        uci:uci-option "major";
        type string {
          pattern "(CS|IMS)";
        }
      }

      leaf grade {
        uci:uci-option "grade";
        type grade;
      }
    }

    container instructor {
      uci:uci-section "instructor";
      uci:uci-section-type "instructor";

      leaf name {
        uci:uci-option "name";
        type string;
      }

      leaf email {
        uci:uci-option "email";
        type email;
      }
    }
  }
}
```

C OpenWrt System data model

```
module openwrt-system {
  namespace "urn:jacobs:yang:openwrt-system";
  prefix "os";

  import ietf-inet-types {
    prefix inet;
  }

  import openwrt-uci-extension {
    prefix ex;
  }

  contact "Malte Granderath <m.granderath@jacobs-university.de>";
  revision 2019-04-24 {
    description "initial revision";
  }

  ex:uci-package "system";
  container system {
    ex:uci-section "OpenWrt";
    ex:uci-section-type "system";

    leaf hostname {
      ex:uci-option "hostname";
      type string;
      description "The hostname of the system";
    }

    leaf buffersize {
      ex:uci-option "buffersize";
      type uint32;
      description "Size of the kernel message buffer.";
    }

    leaf conloglevel {
      ex:uci-option "conloglevel";
      type uint8 {
        range "0..7";
      }
      default 7;
      description
        "Number between 1-8. The maximum log level for kernel
           messages to be logged to the
        console. Only messages with a level lower than this will
           be printed to the console.
        Higher level messages have lower log level number.
           Highest level messages are ones with
        log level 0. If you want more verbose messages in console
            put conloglevel to 8
        if you want less messages lower conloglevel to 4 or even
           less.";
    }
```

```
leaf cronloglevel {
  ex:uci-option "cronloglevel";
  type uint8;
  default 5;
  description
    "The minimum level for cron messages to be logged to
       syslog. 0 will print all debug messages,
     8 will log command executions, and 9 or higher will only
         log error messages.";
}

leaf klogconloglevel {
  ex:uci-option "klogconloglevel";
  type uint8;
  default 7;
  description
    "The maximum log level for kernel messages to be logged
       to the console. Only messages with a level
     lower than this will be printed to the console.
        Identical to conloglevel and will override it.";
}

leaf log_buffer_size {
  ex:uci-option "lof_buffer_size";
  type uint32;
  default 16;
  description
    "Size of the log buffer of the   n e w   procd based
       system log, that is output by the logread command";
}

leaf log_file {
  ex:uci-option "log_file";
  type string;
  description
    "File to write log messages to (type file). The default
       is to not write a log in a file. The most
     often used location for a system log file is /var/log/
        messages.";
}

leaf log_hostname {
  type string;
  description
    "Hostname to send to remote syslog. If none is provided,
       the actual hostname is send.";
}

leaf log_ip {
  ex:uci-option "log_ip";
  type inet:ipv4-address;
  description
    "IP address of a syslog server to which the log messages
       should be sent in addition to the local destination
       .";
}
```

```
leaf log_port {
  ex:uci-option "log_port";
  type uint32;
  description
    "Port number of the remote syslog server";
}

leaf log_prefix {
  ex:uci-option "log_prefix";
  type string;
  description
    "Adds a prefix to all log messages send over network";
}

leaf log_proto {
  ex:uci-option "log_proto";
  type string {
    pattern "(udp|tcp)";
  }
  default "udp";
  description
    "Sets the protocol to use for the connection";
}

leaf log_remote {
  ex:uci-option "log_remote";
  type boolean;
  default false;
  description "Enables remote logging.";
}

leaf log_size {
  ex:uci-option "log_size";
  type uint32;
  default 16;
  description "Size of the file or circular memory buffer in
     KiB.";
}

leaf log_trailer_null {
  ex:uci-option "log_trailer_null";
  type boolean;
  default false;
  description 'Use \0 instead of \n as trailer when using TCP
     .';
}

leaf log_type {
  ex:uci-option "log_type";
  type string {
    pattern "(circular|file)";
  }
  description "Either circular or file.";
}
```

```
    leaf urandom_seed {
      ex:uci-option "urandom_seed";
      type string;
      default "0";
      description "Path of the seed. Enables saving a new seed on
          each boot.";
    }

    leaf timezone {
      ex:uci-option "timezone";
      type string;
      description "The time zone that date and time should be
         rendered in by default.";
    }

    leaf zonename {
      ex:uci-option "zonename";
      type string;
      description "Only useful when using glibc and zoneinfo!";
    }
  }

  container ntp {
    ex:uci-section-type "timeserver";
    ex:uci-section "ntp";

    leaf enabled {
      ex:uci-option "enabled";
      type string {
        pattern "(0|1)";
      }
      description "enable the ntp client";
    }

    leaf enable_server {
      ex:uci-option "enable_server";
      type string {
        pattern "(0|1)";
      }
      description "enable the timeserver";
    }

    leaf-list server {
      ex:uci-option "server";
      type string;
      description "list of ntp servers";
    }
  }
}
```

D OpenWrt Network data model

```
module openwrt-network {
  namespace "urn:jacobs:yang:openwrt-network";
  prefix "on";

  import ietf-inet-types {
    prefix inet;
  }

  import ietf-yang-types {
    prefix yang;
  }

  import openwrt-uci-extension {
    prefix uci;
  }

  contact "Malte Granderath <m.granderath@jacobs-university.de>";
  revision 2019-04-24 {
    description "initial revision";
  }

  typedef openwrt-bool {
    type string {
      pattern "(0|1)";
    }
  }

  uci:uci-package "network";
  container globals {
    uci:uci-section-type "globals";
    uci:uci-section "globals";

    leaf ula_prefix {
      uci:uci-option "ula_prefix";
      type inet:ipv6-prefix;
      description "IPv6 prefix for device";
    }
  }

  container interfaces {
    list interface {
      uci:uci-section-type "interface";
      key "ifname";

      leaf ifname {
        uci:uci-option "ifname";
        type string;
        mandatory true;
        description "Physical interface name";
      }

      leaf iftype {
        uci:uci-option "type";
```

```
  type string;
  description "Type of interface";
}

leaf stp {
  uci:uci-option "stp";
  type openwrt-bool;
  description "Enable spanning tree protocol";
}

leaf bridge_empty {
  uci:uci-option "bridge_empty";
  type openwrt-bool;
  description "Enable creating empty bridge";
}

leaf igmp_snooping {
  uci:uci-option "imgp_snooping";
  type openwrt-bool;
  description "Enable multicast_snooping";
}

leaf multicast_querier {
  uci:uci-option "multicast_querier";
  type openwrt-bool;
  description "Enable mutlicast_queries";
}

leaf macaddr {
  uci:uci-option "macaddr";
  type yang:phys-address;
  description "Override the MAC address of the interface";
}

leaf mtu {
  uci:uci-option "mtu";
  type int64;
  description "Override the default MTU of the interface";
}

leaf auto {
  uci:uci-option "auto";
  type openwrt-bool;
  description "Bring up interface at boot";
}

leaf ipv6 {
  uci:uci-option "ipv6";
  type openwrt-bool;
  description "Enable or disable IPv6";
}

leaf force_link {
  uci:uci-option "force_link";
  type openwrt-bool;
  description
```

```
    "Specifies whether ip address, route, and optionally
       gateway are assigned to the interface
  regardless of the link being active ('1') or only after
     the link has become active ('0');";
}

leaf disabled {
  uci:uci-option "disabled";
  type openwrt-bool;
  description "Enable or disable the interface";
}

leaf ip4table {
  uci:uci-option "ip4table";
  type string;
  description "IPv4 routing table for this interface";
}

leaf ip6table {
  uci:uci-option "ip6table";
  type string;
  description "IPv6 routing table for this interface";
}

leaf proto {
  uci:uci-option "proto";
  type string;
  description "The protocol of the interface";
}

leaf ipaddr {
  uci:uci-option "ipaddr";
  type inet:ipv4-address;
  description "Alias IP address";
}

leaf netmask {
  uci:uci-option "netmask";
  type inet:ipv4-prefix;
  description "Alias Netmask";
}

leaf gateway {
  uci:uci-option "gateway";
  type inet:ipv4-address;
  description "Default gateway";
}

leaf broadcast {
  uci:uci-option "broadcast";
  type inet:ipv4-address;
  description "Broadcast address (autogenerated if not
     setup)";
}

leaf ip6addr {
```

```
      uci:uci-option "ip6addr";
      type inet:ipv6-address;
      description "IPv6 address";
    }

    leaf ip6gw {
      uci:uci-option "ip6gw";
      type inet:ipv6-address;
      description "IPv6 default gateway";
    }

    leaf-list dns {
      uci:uci-option "dns";
      type inet:ipv4-address;
      description "list of dns servers";
    }

    leaf layer {
      uci:uci-option "layer";
      type uint8 {
        range "1..3";
      }
      description "Selects the interface to attach to for
         stacked protocols";
    }
  }
}

container rules {
  list ip4rules {
    uci:uci-section-type "rule";
    key "mark";

    leaf mark {
      uci:uci-option "mark";
      type string;
      description "Specifies the fwmark";
    }

    leaf in {
      uci:uci-option "in";
      type string;
      description "Specifies the incoming logical interface
         name";
    }

    leaf out {
      uci:uci-option "out";
      type string;
      description "Specifies the outgoing logical interface
         name";
    }

    leaf src {
      uci:uci-option "src";
      type inet:ipv4-prefix;
```

```
    description "Specifies the source subnet to match";
  }

  leaf dest {
    uci:uci-option "dest";
    type inet:ipv4-prefix;
    description "Specifies the destination subnet to match";
  }

  leaf tos {
    uci:uci-option "tos";
    type uint64;
    description "Specifies the TOS value to match in the IP
       headers";
  }

  leaf invert {
    uci:uci-option "invert";
    type openwrt-bool;
    description "Inverts the meaning of match options";
  }

  leaf priority {
    uci:uci-option "priority";
    type uint64;
    description "Declares the priority of the rule";
  }

  leaf lookup {
    uci:uci-option "lookup";
    type string;
    description "The rule lookup target";
  }

  leaf goto {
    uci:uci-option "goto";
    type uint64;
    description "The rule target is another rule with
       priority";
  }

  leaf action {
    uci:uci-option "action";
    type string;
  }
}

list ip6rules {
  uci:uci-section-type "rule";
  key "mark";

  leaf mark {
    uci:uci-option "mark";
    type string;
    description "Specifies the fwmark";
  }
```

```
leaf in {
  uci:uci-option "in";
  type string;
  description "Specifies the incoming logical interface
     name";
}

leaf out {
  uci:uci-option "out";
  type string;
  description "Specifies the outgoing logical interface
     name";
}

leaf src {
  uci:uci-option "src";
  type inet:ipv6-prefix;
  description "Specifies the source subnet to match";
}

leaf dest {
  uci:uci-option "dest";
  type inet:ipv6-prefix;
  description "Specifies the destination subnet to match";
}

leaf tos {
  uci:uci-option "tos";
  type uint64;
  description "Specifies the TOS value to match in the IP
     headers";
}

leaf invert {
  uci:uci-option "invert";
  type openwrt-bool;
  description "Inverts the meaning of match options";
}

leaf priority {
  uci:uci-option "priority";
  type uint64;
  description "Declares the priority of the rule";
}

leaf lookup {
  uci:uci-option "lookup";
  type string;
  description "The rule lookup target";
}

leaf goto {
  uci:uci-option "goto";
  type uint64;
  description "The rule target is another rule with
```

```
          priority";
      }

      leaf action {
        uci:uci-option "action";
        type string;
      }
    }
  }

  container routes {
    list ip4routes {
      uci:uci-section-type "route";
      key "name";

      leaf name {
        uci:uci-option "name";
        type string;
        description "Name of rule";
      }

      leaf interface {
        uci:uci-option "interface";
        type string;
        mandatory true;
        description "Specifies the interface name this route
           belongs to";
      }

      leaf target {
        uci:uci-option "target";
        type inet:ipv4-address;
        description "The target network address";
      }

      leaf netmask {
        uci:uci-option "netmask";
        type inet:ipv4-prefix;
        description "The route netmask";
      }

      leaf gateway {
        uci:uci-option "gateway";
        type inet:ipv4-address;
        description "The network gateway";
      }

      leaf metric {
        uci:uci-option "metric";
        type uint64;
        description "Specifies the route metric to use";
      }

      leaf mtu {
        uci:uci-option "mtu";
        type uint64;
```

```
    description "Specifies a MTU for this route";
  }

  leaf table {
    uci:uci-option "table";
    type string;
    description "Specifies the table id to use for route";
  }

  leaf source {
    uci:uci-option "source";
    type inet:ipv4-address;
    description "The preferred source address when sending to
        destinations";
  }

  leaf onlink {
    uci:uci-option "onlink";
    type openwrt-bool;
    description "Enable gateway on link, even if gateway does
        not match any interface prefix";
  }

  leaf rtype {
    uci:uci-option "type";
    type string;
  }
}

list ip6routes {
  uci:uci-section-type "route6";
  key "name";

  leaf name {
    uci:uci-option "name";
    type string;
    description "Name of rule";
  }

  leaf interface {
    uci:uci-option "interface";
    type string;
    mandatory true;
    description "Specifies the interface name this route
       belongs to";
  }

  leaf target {
    uci:uci-option "target";
    type inet:ipv6-address;
    description "The target network address";
  }

  leaf gateway {
    uci:uci-option "gateway";
    type inet:ipv6-address;
```

```
      description "The network gateway";
    }

    leaf metric {
      uci:uci-option "metric";
      type uint64;
      description "Specifies the route metric to use";
    }

    leaf mtu {
      uci:uci-option "mtu";
      type uint64;
      description "Specifies a MTU for this route";
    }

    leaf table {
      uci:uci-option "table";
      type string;
      description "Specifies the table id to use for route";
    }

    leaf source {
      uci:uci-option "source";
      type inet:ipv4-address;
      description "The preferred source address when sending to
          destinations";
    }

    leaf onlink {
      uci:uci-option "onlink";
      type openwrt-bool;
      description "Enable gateway on link, even if gateway does
          not match any interface prefix";
    }

    leaf rtype {
      uci:uci-option "type";
      type string;
    }
  }
}

container switches {
  list switch {
    uci:uci-section-type "switch";
    key "name";

    leaf name {
      uci:uci-option "name";
      type string;
      description "The name of the switch to configure";
    }

    leaf reset {
      uci:uci-option "reset";
      type openwrt-bool;
```

```
  }

  leaf enable_vlan {
    uci:uci-option "enable_vlan";
    type openwrt-bool;
  }

  leaf enable_mirror_rx {
    uci:uci-option "enable_mirror_rx";
    type openwrt-bool;
    description "Mirror received packets from the
       mirror_source_port to the mirror_monitor_port";
  }

  leaf enable_mirror_tx {
    uci:uci-option "enable_mirror_tx";
    type openwrt-bool;
    description "Mirror transmitted packets from the
       mirror_source_port to the  mirror_monitor_port";
  }

  leaf mirror_monitor_port {
    uci:uci-option "mirror_monitor_port";
    type uint64;
    description "Switch port to which the packets are
       mirrored";
  }

  leaf mirror_source_port {
    uci:uci-option "mirror_source_port";
    type uint64;
    description "Switch port from which packets are mirrored
       ";
  }

  leaf arl_age_time {
    uci:uci-option "arl_age_time";
    type uint64;
    description "Adjust the address resolution table's aging
       time";
  }

  leaf igmp_snooping {
    uci:uci-option "igmp_snooping";
    type openwrt-bool;
  }

  leaf igmp_v3 {
    uci:uci-option "igmp_v3";
    type openwrt-bool;
  }
}

list switch_vlan {
  uci:uci-section-type "switch_vlan";
  key "device vlan";
```

```
  leaf device {
    uci:uci-option "device";
    type string;
    description "The device to configure";
  }

  leaf vlan {
    uci:uci-option "vlan";
    type uint64;
    description "The VLAN table index to configure";
  }

  leaf vid {
    uci:uci-option "vid";
    type uint64;
    description "The VLAN tag number to use";
  }

  leaf-list ports {
    uci:uci-option "ports";
    type string;
    description "List of port indices that should be
       associated with VLAN";
  }
}

list switch_port {
  uci:uci-section-type "switch_port";
  key "device port";

  leaf device {
    uci:uci-option "device";
    type string;
    description "The device to configure";
  }

  leaf port {
    uci:uci-option "port";
    type uint64;
    description "The port index to configure";
  }

  leaf pvid {
    uci:uci-option "pvid";
    type uint64;
    description "The port PVID";
  }

  leaf enable_eee {
    uci:uci-option "enable_eee";
    type openwrt-bool;
    description "Enable power-saving features";
  }

  leaf igmp_snooping {
```

```
        uci:uci-option "igmp_snooping";
        type openwrt-bool;
      }

      leaf igmp_v3 {
        uci:uci-option "igmp_v3";
        type openwrt-bool;
      }
    }
  }
}
```

E UCI testing content

```
config course 'course'
        option name 'Computer Networks'
        option semester '5'
        list instructors 'Replaced'
        list instructors 'Added item'

config instructor 'instructor'
        option name 'Example Lecturer'
        option email 'example@university.de'

config student
        option firstname 'test'
        option lastname 'student'
        option age '21'
        option major 'CS'
        option grade '75'

config student
        option firstname 'test2'
        option lastname 'student'
        option age '21'
        option major 'IMS'
        option grade '60'
```

F Existing RESTCONF implementations

F.1 JetConf

JetConf is a pure RESTCONF implementation that supports all mandatory features of the specification. It only supports the JSON encoding and only HTTP/2 transport because it is using a python web server. There is an existing in-memory datastore, but JetConf exposes an API for extending the core with custom implementations. It has several non-standard library dependencies that have to be installed separately [11].

There is a python 3 package available on OpenWrt, however, running a python application usually is relatively slow. The required packages can be precompiled and included as `.pyc` files, but that requires more amount of memory space since the whole modules will be loaded into memory. Additionally, the python package would have to be installed before running the server or come preinstalled with the image.

F.2 MD-SAL

Model-Driven Service Adaptation Layer (MD-SAL) is part of the OpenDaylight (ODL) Project, which is a modular platform for configuring and automating networks. MD-SAL is an infrastructure component of ODL that, based on user-defined data and interface models, provides messaging and data storage. It supports several transport and payload formats with one of them being RESTCONF [13].

RESTCONF is part of an MD-SAL artifact and is installed using Apache Karaf. MD-SAL is written in Java and running the JVM on OpenWrt has been shown to have low performance. Furthermore, only specially modified versions of the JVM run on OpenWrt and require additional installation steps.

F.3 Clixon

Clixon is a YANG based configuration manager with both NETCONF and RESTCONF interfaces and an interactive CLI tool. Clixon provides the core system and can be used as-provided but exposes an API for plugins, such that additional functionality can be implemented. NETCONF and RESTCONF are enabled by utilizing the extension API. The RESTCONF implementation is based on the FastCGI C-API and the recommended web server is NGINX [9].

Clixon utilizes dynamic datastore plugins that are loaded at runtime. The goal of this is simplifying the implementation of additional datastores. Clixon supports all three of the defined configuration datastores: *running*, *startup* and *candidate* [9].

F.4 Limitations

OpenWrt is available on a variety of different devices with different architectures, memory and storage limitations. While Clixon can use the shared libraries and is already used on a variety of different software for embedded devices [9], it is mainly a configuration

manager and provides interfaces for the protocols and can be too complex for a system that requires one of the protocols. Additionally, there are still some problems with running Clixon on different architectures[7]. MD-SAL is an enterprise solution and will be very inefficient on systems with limited resources. JetConf could be optimized for running on OpenWrt, but its memory footprint, due to not using shared libraries, prevents it from running on highly resource constrained systems.

[7] https://github.com/clicon/clixon/issues/58