MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Implementation of the protocol RESTCONF in GNU/Linux

DIPLOMA THESIS

Richard Janík

Brno, spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Richard Janík

Advisor: Ing. Mgr. et Mgr. Zdeněk Říha, Ph.D.

Acknowledgement

I would like to thank the supervisor of this thesis Ing. Mgr. et Mgr. Zdeněk Říha, Ph.D. for constantly providing me with feedback and timely help. Next, I'd like to thank my family for being a source of motivation and for reminding me who I aim to be and who I aim not to be. Last, but definitely not least, I would like to thank my consultant RNDr. Radek Krejčí for sharing his technical know-how and for his near-limitless patience with me.

Abstract

Thesis consists of two parts: practical – the implementation of the RESTCONF server itself, which is available with the text of the thesis, and descriptive – the description of technologies used in the process of implementing the server and the following discussion of these technologies, design of the server and its usability.

Keywords

RESTCONF, NETCONF, server, Linux, YANG, XML, JSON, HTTP, RPC

Contents

1 Introduction

Today's networks consist of services and technologies linked together in a manner that allows the network to serve its purpose. Services that networks consist of are often complex as to accommodate for varying properties of their environment and this complexity brings forth the need for their correct configuration. Configuring services manually would require a great deal of effort, thus the necessity to configure services programmatically gave rise to network management and configuration protocols.

There are various network management and configuration protocols of varying scope and complexity being currently used by companies all over the world. One of these is the NETCONF[1] protocol[1]. The NETCONF protocol falls under the standardization process of the IETF organization and is designed mainly for configuring devices. The difference between NETCONF and the, arguably, most well known network management protocol SNMP is, that while the latter is a request-response messaging protocol on top of a connectionless transport, NETCONF is based on an XML messaging protocol using a connection-oriented transport (e.g. the Transmission Control Protocol) [2]. Moreover the SNMP is not optimal for configuration management [3].

NETCONF protocol is based around XML encoded messages and RPCs between clients and a server. The server serves as an entry point to a datastore – a conceptual place to store and access information – which clients use to configure or monitor network based devices. Both client and server applications need to support the NETCONF protocol. Due to the large number of protocol features and useful operations, this solution has many advantages but is somewhat heavy-weight. The Figure number 1 illustrates how this system works.

---

1. NETCONF simply stands for "Network Configuration". More information about NETCONF protocol and tools related to it can also be found in chapters 3 which describes the connection between RESTCONF and NETCONF in detail and 4 where a library implementing useful NETCONF related functions is presented.

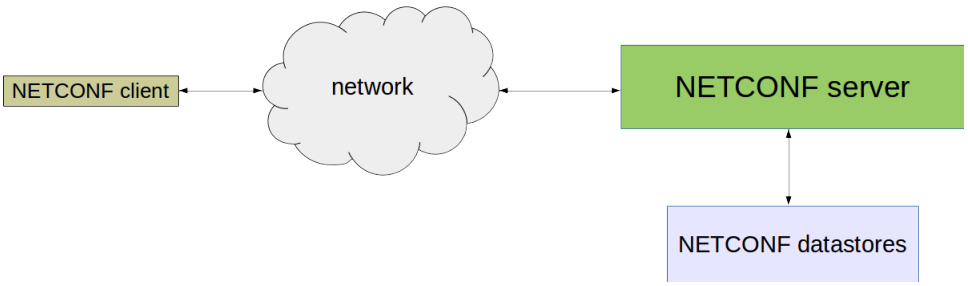

Figure 1.1: A simplified view of the NETCONF stack

The current trend in network management protocols however, sees importance in light-weight protocols that can be used without the need to install/operate complex specific software such as would be NETCONF servers and clients. Such protocols would be ideally easily integrable in exchange for a portion of functionality.

In response to this need a protocol called RESTCONF[2] is being developed by IETF (currently in the draft stage [4]). This protocol aims to supplant the NETCONF protocol in places where a light-weight REST-based client with a subset of functionality is sufficient to configure the network devices (e.g. the well known `curl` software would be sufficient as a RESTCONF client). The only part of the system which would require special software support are the network devices which would have to be managed by a RESTCONF capable server. It is the aim of this thesis to provide an implementation of such server. Furthermore, since RESTCONF still functions as an entry point to NETCONF datastores, it is preferred that this implementation is based upon existing NETCONF technologies.

RESTCONF is a RESTful protocol – that is, it's architecture is client-server, it is stateless, potentially cacheable and layered. [5] The advantages of REST-ful design lie mainly in simple integration into existing management systems. Companies adopting RESTCONF as

2. RESTCONF stands for "REST Configuration" where REST is yet another abbreviation that stands for "Representational State Transfer". How RESTCONF is supposed to operate is explained in Chapter 3.

their network management and configuration protocol of choice can expect the client and server applications to communicate through a well known protocol – the HTTP protocol in this case.

1.1 Thesis objectives

These are the objectives of the thesis:

1. Design and implement a RESTCONF server that will comply with the RESTCONF protocol draft currently available on the IETF web page [4]. The thesis takes into account only a single version of this draft as the draft has changed a few times while the thesis was in progress and is expected to be updated in the future as well.

2. Discuss the advantages and disadvantages of this implementation and provide guidelines for further server development.

As mentioned before, in Chapter 1, the RESTCONF server should have its implementation based upon already existing NETCONF technologies if possible. Thus, our implementation operates as a modification to an already existing open-source NETCONF server. This server is called Netopeer [6].

Netopeer server already implements numerous NETCONF features that serve as a basis for RESTCONF services and using Netopeer as a base upon which a RESTCONF server is built is beneficial to both projects as it expands the user base for the Netopeer project which in turn provides support for NETCONF datastores.

1.2 Chapter distribution

The text of the thesis is laid out in two parts: one introduces the reader to the problem at hand, the other comments on the server design and implementation:

Introductory part: First four chapters (including Chapter 1 – Introduction) of the thesis contemplate the basics upon which the

core of this work is built.

The second chapter – **RESTCONF protocol** – explains what RESTCONF server is and what other technologies are used. The next chapter – **Requirements** – explains what requirements were placed on the practical part of the thesis and lists the features – both supported and unsupported. The fourth chapter – **Tools description** – shows what tools were employed in the development of the server

Server part: The rest of the thesis describes the implementation details and use cases of our server.

The fifth chapter – **Server design** – describes the design of the server and discusses the possible deviations that were considered as the thesis evolved. Chapters six to eight – **YANG Parser**, **XML-JSON converter**, and **RESTCONF Server** – go through the components of our server and present the design decisions, advantages and disadvantages and use cases of these components.

There is a list of used literature as well as a description of what is included on the DVD attached to this thesis at the end of the thesis text after Chapter 9.

The DVD also contains a virtual environment with RESTCONF server and tools to test its functionality. A tutorial on how to start the testing environment and test the servers functionality is provided in Chapter 8.

2 RESTCONF protocol

This chapter for the first time in depth explains what the RESTCONF protocol actually is and in what relationship it is to the existing NETCONF protocol. It also lists and describes other technologies that we have encountered in the process of implementation of the RESTCONF server or with which the RESTCONF protocol works.

2.1 RESTCONF description

As explicitly stated in the above referenced draft, RESTCONF is a REST-like protocol running over HTTP for accessing data described in YANG using datastores defined in NETCONF [4].

HTTP is a well known and flexible protocol that is used for many purposes in many situations, most notable being common Internet traffic. Its wide spread and availability is responsible for it being a good candidate for the RESTCONF protocol which is required to be easily integrable – as easy integration is one of the main reasons for RESTCONF protocol to exist. The HTTP RFC[1] is available at the IETF web page split into multiple documents[7]. NETCONF datastores and YANG language deserve a more complex description and are covered below in sections 2.3 and 2.4.

2.2 Overview

All RESTCONF messages are transmitted through the HTTP protocol. There can be two types of HTTP messages – either requests or responses [8]. An HTTP request is composed of HTTP method (GET, POST, HEAD, ...) identifier, resource identifier (explained in subsection 2.2.1), HTTP protocol version, HTTP headers and finally, HTTP body.

1. RFC – Request For Comments – an online specification for a technology standardized by the IETF.

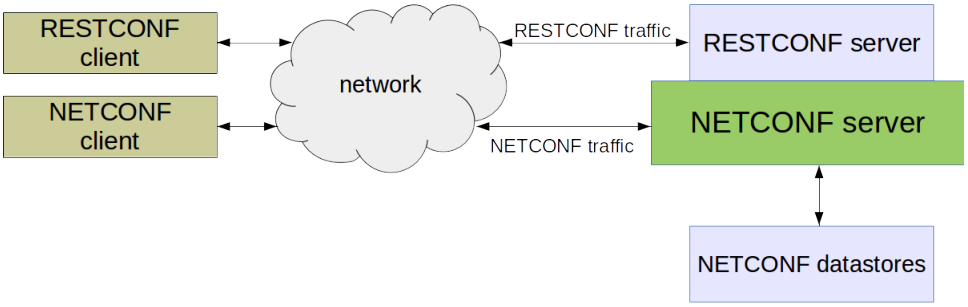

Figure 2.1: NETCONF and RESTCONF interoperability

The headers needed for communication are defined by the draft. Different HTTP headers may be allowed for different resource identifiers thus the HTTP headers are covered in the sections for their respective resource (e.g. requests on the `/restconf/operation` resource may require a `Content-Type` header). The server is also required to return specific HTTP headers, these are described in the resource sections as well since these do differ for each resource.

RESTCONF server is also responsible for keeping record of timestamps and etags[2]. These are mandatory for some resources, optional for other and unsupported for resources where they don't provide any useful information. Thus, they are also covered in sections covering the RESTCONF resource API.

A sample RESTCONF interaction thus consists of an HTTP request sent by a RESTCONF client and an HTTP response sent by the server. Both of these contain a required set of expected HTTP headers and may contain a request or response body. RESTCONF draft states that the message body may be encoded in XML or JSON.

2. Etag is an identifier for a specific version of a resource which is represented by a hexadecimal number. Since it should identify a resource without ambiguity its uniqueness should be guaranteed by the server.

While the XML (EXtensible Markup Language) is a well established specification for encoding documents [9], the JSON (JavaScript Object Notation) language is a more recently standardized lightweight data-interchange format [10].

Our implementation only accepts JSON encoding, since at the time of writing, there is mostly demand for this encoding and the NETCONF server Netopeer already operates with XML encoding (also see Section 3.4 – Future work).

Our server implementation is structured in such a way that it allows for one server to function as both a RESTCONF and NETCONF server. The details of this implementation are given in the Chapter 5 but a simplified scheme is shown in figure 2.1. Both NETCONF and RESTCONF client should be able to specify the same target host, the only difference being the used network ports.

An example of a simple message exchange between a RESTCONF client and a RESTCONF server would be the following:

Client request:

```
GET /restconf/data/ HTTP/1.1
Host: example.com
Accept: application/yang.data+json
```

Client response:

```
HTTP/1.1 200 OK
Date: Tue, 14 Apr 2015 17:01:00 GMT
Server: restconf-impl-server
Content-Type: application/yang.data+json

{
  "data": {
    "system" : {
      // contents taken out for display purposes
    }
  }
}
```

2.2.1 HTTP resources

Terms HTTP resource and HTTP resource identifier have a similar meaning and are often used interchangeably in the text of this thesis. The HTTP resource is an abstract concept of a document, service or another type of resource that a server makes available for a client while HTTP resource identifier is the string which identifies this resource. Since resources are identified by these strings without any ambiguity these terms are roughly convertible.

A RESTCONF HTTP request has to contain an HTTP resource identifier. This resource identifier always has to start with the string "`/restconf`". Any HTTP request which does not specify a resource starting with the well-known entry point "`/restconf`" is not a valid RESTCONF request. The "`/restconf`" resource has a few allowed subresources. All of these are covered in Chapter 3.

2.3 NETCONF datastores

A NETCONF datastore is a conceptual place to store information, often information about the configuration of a device. This could be a simple file store or a specialized database. NETCONF does not differentiate between these and the specific storage method is to be provided by the server implementation.

NETCONF defines several types of datastores. The only mandatory datastore is the *running* configuration datastore – which is the datastore which holds the configuration currently in use by the given device.

Then there is the *candidate* configuration datastore which holds configuration that may eventually become the running configuration but is not yet in effect. This allows this datastore to be manipulated without impacting the configuration of a device and only commit it to the running configuration datastore when the configuration is

ready to be commited. The candidate configuration datastore is not necessary and only some servers may support it.

The last datastore is the *startup* configuration datastore which, in case it is supported by the server, contains the configuration to be loaded by the server when it starts up. The interaction between these datastores is subject to complex locking and transaction mechanisms as defined by the NETCONF RFC.

Nevertheless, the RESTCONF protocol only defines a single unified datastore that is visible to the RESTCONF client. This datastore represents the *running* datastore of the NETCONF protocol and the other two datastores (*startup* and *candidate*) are hiden from the RESTCONF client. How this datastore is implemented is to be decided by the developer of a server implementation and it can be implemented on top of the common NETCONF datastores – which is the way it is implemented in our RESTCONF server.

Datastore support is a part of Netopeer server and there are (or can be implemented) user modules which are used to apply the configuration to a device. The Netopeer and its modules handle all interactions between the NETCONF datastores and devices and our RESTCONF server is made free from the responsibility of datastore management. Thus, the RESTCONF client does not specify a datastore target for his operations, this is handled by the underlying Netopeer server and the RESTCONF client percepts a single unified datastore.

2.4 YANG data modeling language

YANG is a data modeling language standardized by the IETF and its RFC is available at the IETF web page [11]. It has been developed for use by the NETCONF protocol and technologies related to it. It is used to define hierarchical and typed data models which describe the configuration data managed by the RESTCONF and NETCONF protocols. The RFC defines its syntax and available features. One of the more notable features of YANG data modeling language is the

possibility to augment existing YANG data models without the need to modify the original data model.

The body of every HTTP message, be it a request or a response encoded in XML or JSON data must adhere to a hierarchical structure defined within a YANG model document. Each such HTTP message has to be converted inside our RESTCONF server and such conversion requires the information available in the YANG models relevant to the message data and these models thus need to be supported by the server.

Since YANG is critical to model the data transported between the RESTCONF client and RESTCONF server and there is currently no known YANG parser implementation in the C programming language, our server uses its own simple YANG parser. This parser and its possible use cases are described in Chapter 6.

An alternative to the YANG data modeling language is the YIN data modeling language, which specifies a similar set of features but uses XML encoding. YIN is used by the Netopeer server and the `libnetconf` NETCONF library. The RESTCONF draft however uses YANG to model data where possible thus creating incentive to use it instead of the YIN data modeling language.

3 Requirements

The RESTCONF protocol is defined by the RESTCONF draft [1] which is available at the IETF web page [4]. The draft specifies the mandatory and optional features of the RESTCONF protocol. This chapter first describes the idea of the protocol in greater detail and then iterates through the technical requirements placed on RESTCONF implementations. In the end it says which features are supported or unsupported by our implementation and what features could possibly be added during the life of the server.

3.1 HTTP resources

When a RESTCONF client sends an HTTP request to the server a RESTCONF operation is extracted from the URL. A RESTCONF operation contains an HTTP method and a resource identifier. All resource identifiers have to start with the well known `"/restconf"` prefix. What follows after the well known prefix is the subresource identifier which consists of a path, query and a fragment part. The path and query parts are separated by a `"?"` character while the query and fragment parts are separated by a `"#"` character. The fragment part of the resource identifier is however ignored by the RESTCONF protocol as it has currently no use. The figure 3.1 shows the layout of a valid RESTCONF operation.

RESTCONF draft specifies several known HTTP subresources to `/restconf`. Each of these resources makes available a different aspect of the RESTCONF protocol. These are described below in detail. The main application programming interface resource `/restconf` has media type `"application/yang.api+json"` and is used by the RESTCONF client for retrieving the structure of the RESTCONF data. It supports the `"depth"` query parameter to control the level of recursion when returning the children of this resource.

1. An IETF draft is an in development standard which is undergoing the common standardization process employed by IETF.

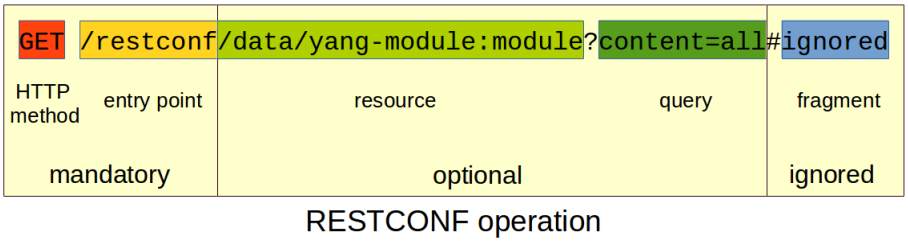

Figure 3.1: RESTCONF operation

3.1.1 `data` and `datastore` resources

The main `/restconf` resource contains a `/data` subresource. This mandatory resource by itself represents the combined configuration and operational data resources that can be accessed by a client. It cannot be created or deleted by the RESTCONF client.

The subtree of the `/data` resource represents the **datastore** resource type – a collection of configuration and operational data nodes. The media type for the **datastore** resource is:

```
application/yang.datastore+json
```

A **datastore** resource can either be read with GET HTTP method or written directly with the HTTP PATCH method – which is defined as optional to support by the RESTCONF draft and is currently not supported by our server. The contents of the datastores can still be written by writing the subresources of a **datastore** resource – the **data** resources. You can use the `depth` query parameter with the **datastore** resource. This limits the size of the output and discloses the available subresources. It is needed that the server keeps track of the last change time (the `Last-Modified` time stamp and `Date` header) and its entity tag for the **datastore** resources.

A **data** resource represents a YANG data node that is a descendant node of a **datastore** resource. The **data** resource may optionally support a resource entity tag and a last-modified timestamp for configuration data resources. **data** resources can be accessed via the `/restconf/data` entry point and can be created with the POST

method, edited with the PUT method and deleted with the DELETE method. The media type for a **data** resource is:

```
application/yang.data+json
```

A sample request on a **data** resource can be found in Chapter 8 along with all the HTTP headers relevant for communication.

3.1.2 `modules` resource

This resource makes accessible the structure of configuration data maintained by the server without the need to access the data itself. Clients are expected to request this resource when trying to determine the availability of modules on the server.

The `/modules` resource is defined as mandatory by the RESTCONF draft. A last-modified time stamp must be maintained by the server and an entity tag value is defined as optional but preferred. Our implementation supports both the time stamp and entity-tag value for this resource but not its subresources which are defined as optional in the draft.

The `/modules/module` resource which is defined as mandatory contains an entry for each data model supported by the server and there must be an instance of this resource for every model supported by the server. Again, Chapter 8 contains examples on what requests on this resource look like. The server may maintain entity tag values or timestamps for each of these resources but as writing these resources is disabled (see explanation below) it is not necessary to provide these (the last modified time stamp will always be the same as that of the parent `/modules` resource).

There exists the `/modules/module/<module_name>/schema` resource which can be retrieved by the HTTP GET method for each available module. The request returns the YANG module definition to the client. This subresource is defined as optional by the draft, yet supported by our server. The content type for this resource is `application/yang`.

There is one important difference in the draft definition and our

implementation that concerns the `/modules` resource. This resource and its subresources are defined as writable in the draft while our server does not support writing these resources. The reason being the underlying Netopeer server which doesn't fully support writing model files while running.

3.1.3 `operations` resource

This resource is defined as optional by the RESTCONF draft. It provides access to data-model specific operations. These operations are defined by the `rpc` statement inside the YANG module definition. This resource is not supported by our implementation. In servers where this resource is supported the resource may be omitted if no data-model specific operations are not advertised by the server. Invoking operations is done by issuing an HTTP request with the HTTP POST method.

3.1.4 `streams` resource

The `/streams` resource is defined as optional by the RESTCONF draft and currently not supported by our server (but see Section 3.4). It makes available the NETCONF event streams functionality defined in the NETCONF Event Notifications RFC [12]. Media type for this resource is `application/yang.stream` and these resources can only be created and modified by the server. The RESTCONF client can request these resources using the HTTP GET method. There exists a `/restconf/streams/stream` resource for each available streams.

There are a few supported query parameters for the `/streams` resource:

- filter – Boolean notification filter for event-stream resources.
- start-time – Replays buffer start time for event-stream resources.
- stop-time – Replays buffer stop time for event-stream resources.

3.1.5 `version` resource

Finally, the `/version` resource which is defined as mandatory by the draft and is supported by our server returns the version number of the RESTCONF protocol supported by the server implementation. Our implementation always returns "1.0" as the server version accordingly to the RESTCONF draft.

3.2 Secure transport

RESTCONF protocol is expected to be a secure means of configuring devices through the network, however the RESTCONF draft in its current stage does not specify any details to the RESTCONF protocol that would support secure operation over TLS.

Our implementation uses the `stunnel` encryption wrapper to facilitate secure communication over the TCP transport layer protocol.

The TLS – which stands for Transport Layer Security – is a cryptographic protocol which provides privacy and data integrity between two communicating applications on a network. It is layered on top of a reliable protocol – typically TCP (Transmission Control Protocol) – and is composed of several layers of protocols in itself. The TLS protocol Request For Comments is available at the IETF web page at `http://tools.ietf.org/html/rfc5246`.

Although the RESTCONF client needs to have access to a valid client certificate (details available in Chapter 8) the server does not currently differentiate users from the perspective of RESTCONF protocol. The details on how clients authenticate need to be first described in the RESTCONF draft for us to be able to implement a more complex client authentication mechanism.

3.3 Error reporting

The RESTCONF draft defines the success reporting mechanism for the RESTCONF protocol. The success or failure of RESTCONF

HTTP requests is indicated by the status line of the RESTCONF HTTP response. An HTTP status line consists of a status code – an integer that represents the result – and a verbal description of the return code. Status codes that are less than four hundred (<400) indicate a successful completion of the request (or that nothing needed to be done) while status codes of 400 and greater indicate an error in execution. Moreover status code of 500 or greater indicate some kind of internal error in the server and not necessarily an issue on the RESCONF client side.

As the RESTCONF draft states when an error occurs the server should respond with a status code of 400 or greater according to the type of error that occurred. The server also should include a JSON body in the HTTP response. This JSON body contains more information on the cause of the error. The format for these error reports is detailed in the `ietf-restconf` module available from the RESTCONF draft. The Chapter 8 contains examples on the errors that may result from incorrect usage of this server. All the resources that were said to be unsupported result in a "501 Not Implemented" status line in HTTP response.

For the `/operations` resource, which is currently unsupported, there exists a table which maps the NETCONF operation results onto RESTCONF HTTP response status lines. This table is available in the RESTCONF draft.

3.4 Future work

The RESTCONF draft defines several features our server does not support. The reasons are different, sometimes the optimal solution to the problem would require that support would come from tools used for the implementation, sometimes the draft is unclear or prone to change in the given area and often the reason is simply a time constraint combined with optionality of the given feature. The list below explains which features are unsupported and could be added to the server in the future.

- Support for `/operations` resource could be added in the fu-

ture. This resource is unsupported as our YANG parser has not been programmed with the functionality of converting rpc calls in mind. The adaptation should not be too costly however as the parser can be extended relatively easily.

- Support for `/streams` resource could be added in the future. The implementation should be relatively easy as the advertising of stream resources could simply be passed on from the Netopeer server (streams are advertise in a simple get rpc).
- Support for PATCH method modifying the `/data` datastore resource could be added in the future. The support is missing due to needed better support on the side of internal HTTP parser. The PATCH method is declared optional by the RESTCONF draft.
- JSON meta-data[2] is a mechanism which would need support from a deeper level to function. Currently, the meta-data would need to be stored in a different place than where the actual data is stored (datastore) which is prone to errors and inefficient.
- XML HTTP body encoding for RESTCONF could be supported in the future. The reason for JSON support came from higher demand for JSON support on official NETCONF/RESTCONF communication channels.

2. JSON meta-data is data about JSON data transported in HTTP message bodies. RESTCONF draft speculates that such data could be saved along with configurational data through RESTCONF protocol.

4 Tools description

Several tools and programs created by other people are a part of our server implementation. The most important to our implementation is the open-source Netopeer NETCONF server which serves as the basis of the RESTCONF server, providing implementation of NETCONF datastores.

Netopeer makes use of the `libnetconf` library which is a complex tool for implementing NETCONF servers. In case TLS authentication is activated – which is a requirement for our server – the `stunnel` encryption wrapper provides SSL functionality for the Netopeer server. These three tools are described more in depth in the following sections.

4.1 `libnetconf`

The `libnetconf` library is a NETCONF library and the basis of the Netopeer server described below in Section 4.2. It provides functions for establishing connection between a NETCONF client and server via SSH and storing and accessing configuration data in datastores. It is available at its google code web page [13].

From the perspective of our RESTCONF server only `libnetconf` functions that are designed for clients are interesting. The reason for this is gone over in Chapter 5 on server design. The RESTCONF agent uses `libnetconf` to create a dummy NETCONF session in the first place thanks to which it appears as a client to the Netopeer server. The usage of the mentioned NETCONF function is shown below:

```
struct nc_session* netconf_con_dummy =
        nc_session_dummy(nc_session_id, "root",
            NULL, capabilities);
```

This session is not a real session since our RESTCONF server is not a true NETCONF client. And all NETCONF sessions have to be destroyed when they are not needed:

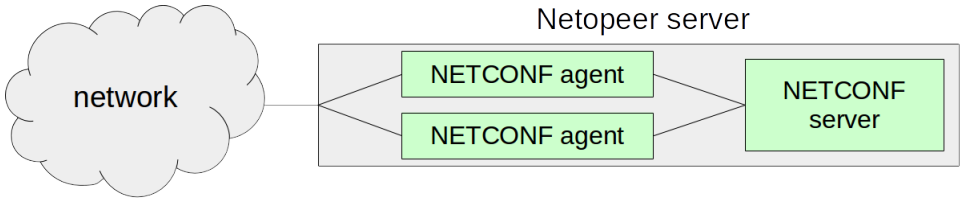

Figure 4.1: Netopeer NETCONF server

```
nc_session_free(netconf_con_dummy);
```

The library is also used for managing NETCONF capabilities:

```
struct nc_cpblts* capabilities =
        nc_session_get_cpblts_default();
nc_cpblts_add(capabilities,
        "urn:ietf:params:xml:ns:yang:ietf-netconf"
        "-monitoring");
...
nc_cpblts_free(capabilities);
```

But the communication with the Netopeer server is managed by functions that were already provided in the `netconf-agent` since this communication goes either over UNIX sockets or the D-Bus message bus system[1]. These are gone through in Section 4.2.

From the perspective of the RESTCONF server the `libnetconf` library has to be compiled with TLS support. This is done by adding the option `--enable-tls` when compiling the library.

4.2 Netopeer

The Netopeer project is a set of tools built upon the `libnetconf` NETCONF library. It is used to connect to devices using the NETCONF protocol, control them and manage them. It contains two implementations of NETCONF clients – a command line interface as well as a graphical user interface – and an implementation of a NET-

1. D-Bus is a program which allows interprocess communication in UNIX type systems by allowing them to send messages to each other.

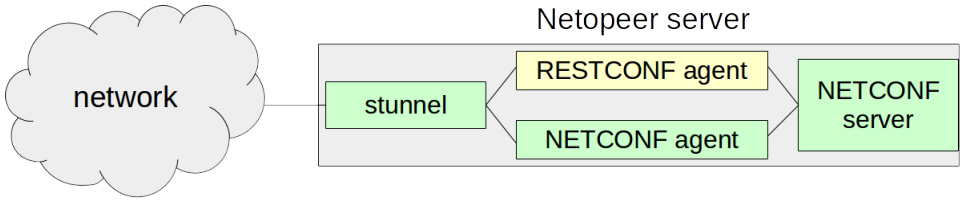

Figure 4.2: RESTCONF integration into the Netopeer server

CONF server. The NETCONF server implemented in the Netopeer project (here simply called the Netopeer server) is the basis for this implementation of RESTCONF server.

4.2.1 Netopeer server architecture

The Netopeer project provides a whole stack of applications for communication via the NETCONF protocol. A client willing to configure a NETCONF-enabled network device would use one of two applications available in the project: either `netopeer-cli` which is a command line text based interface or `Netopeer-GUI` which is a graphical user interface. These client applications are capable of connecting to a remote NETCONF server.

The server part of the stack consists of an SSL subsystem - in our case managed by the `stunnel` described in Section 4.3 which starts so called Netopeer agents and the `netopeer-server` program itself which takes care of configuration datastores and TransAPI modules. TransAPI modules have been lightly mentioned before – these are the user implemented modules that take configuration data from a datastore (a file structure) and apply this data onto devices. A sample TransAPI module is implemented as part of the Netopeer project – the `cfgsystem` module which implements the `ietf-system` data model.

A `netopeer-agent` is started for each NETCONF client session and this agent facilitates the access to the Netopeer server for the

client as well as the communication with a client for the server. The means via which the Netopeer NETCONF agent and server communicate can be chosen at the time of compilation by options to the `./configure` script.

The Netopeer server architecture, which is shown in Figure 4.1, was deemed fit for use by our RESTCONF server which makes use of it by implementing a custom agent launched by the `stunnel` encryption wrapper. The architecture with integrated RESTCONF agent is shown in Figure 4.2. A more detailed description is available in Chapter 5.

4.2.2 Functions for server communication

The `netopeer-agent` or in our case the `restconf-agent` has to communicate with the `netconf-server` as explained in the 4.2.1 subsection. This communication can take two paths in the current implementation of the Netopeer server:

UNIX sockets – UNIX sockets are communication pipes used by processes to communicate in both directions on the same UNIX system [14]. This RESTCONF server implementation is targeted to Linux systems which support UNIX sockets. A RESTCONF or NETCONF agent can use UNIX sockets to communicate with NETCONF server. These are primarily used on systems where D-Bus system is unavailable. The sockets can be utilized by using common UNIX functions for socket communication like `socket(), bind(), listen()` and others.

D-Bus system – D-Bus is an interprocess communication mechanism which is a part of the FreeDesktop.org project. It was designed to replace CORBA[2] and DCOP[3]. It is specifically designed for desktop Linux systems and built upon the idea of binary messages composed of a header and a body [15].

Both of these options are provided by the `comm.h` header file. The D-Bus mechanism is used as a default and UNIX sockets can

2. CORBA stands for Common Object Request Broker Architecture
3. DCOP stands for Desktop Communication Protocol

be activated instead of it by compiling the NETCONF server with the option `-disable-dbus`. For more information, see the Chapter 8. The communication functions of NETCONF agent are utilized by our RESTCONF agent by using functions such as:

```
conn_t* comm_connect();
char** comm_get_srv_cpblts(conn_t* conn);
int comm_session_info_send(conn_t* conn, const
        char* username, const char* sid, int
        cpblts_count, struct nc_cpblts* cpblts);
nc_reply* comm_operation(conn_t* conn,
        const nc_rpc *rpc);
```

The `comm_connect()` function is used for establishing connection with NETCONF server, `comm_get_srv_cblts()` is used for retrieving server NETCONF capabilities which are often useful to the RESTCONF agent, `comm_session_info_send` is used for sending information about our NETCONF session to the NETCONF server and `comm_operation` is used for sending a rpc to the NETCONF server for execution.

4.3 stunnel

`stunnel` is a proxy which adds the TLS encryption functionality to the Netopeer server stack. It uses the open source OpenSSL library for cryptography and is thus fairly modular since many different cryptographic algorithms can be compiled into this library.

The `stunnel` wrapper is an alternative to the Netopeer server as only OpenSSL is normally used to facilitate SSL encryption for the server. TLS encryption is enabled by compiling Netopeer server and the `libnetconf` library with the `--enable-tls` option.

The `stunnel` program is configured to listen on port `6513` and when accepting communication on that port is configured to create an instance of the `netopeer-agent` program. This port is assigned to the NETCONF protocol by the Internet Assigned Numbers Authority.

The same functionality is used by our RESTCONF server, the only difference being that when accepting communication on port `8080` it is told to create a `restconf-agent` instance. The RESTCONF protocol does not have a port number assigned by IANA[4] and this port number has been chosen for the purpose of this thesis only.

More information on how Netopeer and `stunnel` work together is shown in the Figure 5.1 on NETCONF and RESTCONF interoperability in Chapter 5.

4. IANA – which stands for Internet Assiged Numbers Authority is an organization which assigns various numbers – in our case port numbers – to services which are well known by the Internet community

5 Server design

This chapter explains how the RESTCONF server is implemented on top of the Netopeer server. It explains how the technologies from Chapter 4 and our components from chapters 6 and 7 fit together to create a RESTCONF server.

5.1 Server architecture

As was shown in the Section 4.2, the Netopeer server consists of a NETCONF server and a NETCONF agent. This agent receives the communication from NETCONF clients and either forwards it to the NETCONF server for execution or, if capable, evaluates the requests by itself and sends responses back.

The necessary change to this scheme to implement a RESTCONF server is to implement a RESTCONF agent in addition to the original NETCONF agent. This agent will serve as a transformation point for RESTCONF HTTP requests where they will either get evaluated or where they will be converted to NETCONF rpc messages and forwarded to the NETCONF server.

It is necessary to note that the term "NETCONF agent" refers to the Netopeer NETCONF agent – the piece of software implemented as part of the Netopeer project to serve as a proxy to the Netopeer NETCONF server. The "RESTCONF agent", likewise, represents the Netopeer RESTCONF agent. The term "RESTCONF server" refers to the Netopeer server augmented by the Netopeer RESTCONF agent.

The final RESTCONF server works as both a NETCONF and a RESTCONF server. This is made possible by implementing the RESTCONF agent as a separate agent of the server in addition to the original NETCONF agent, both of which are started by the encryption wrapper `stunnel`. The `stunnel` program decides which agent to start based on which port the communication is comming to.

This is depicted in Figure 5.1 where the whole system is shown

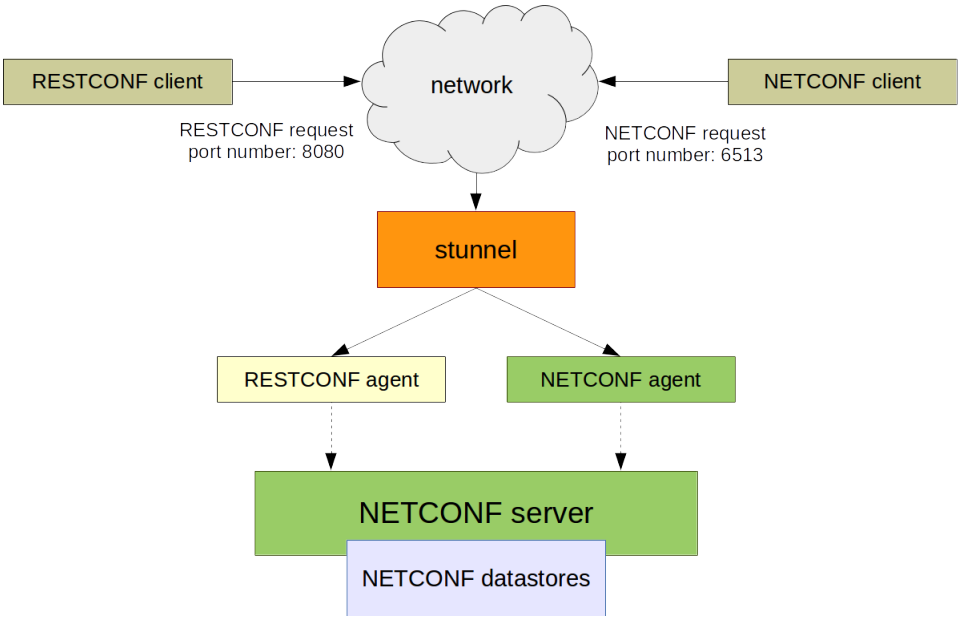

Figure 5.1: NETCONF and RESTCONF interoperability 2

in full. The port number for NETCONF communication is 6513 as established by IANA while the port number for RESTCONF communication – 8080 – was chosen by us because RESTCONF has not been assigned a port number by IANA.

5.2 Message flow

How the whole RESTCONF server works internally is shown in detail in Figure 5.2. It depicts how the messages are received by RESTCONF server internal components and handed over to other components. The flow can be described in multiple steps as follows:

1. `stunnel` receives encrypted NETCONF of RESTCONF message from the network and depending on which port the message has been received it starts either a RESTCONF or NETCONF agent. It does not manipulate the data itself in any way, the only exception being decrypting the data and putting them on standard input of the given agent.

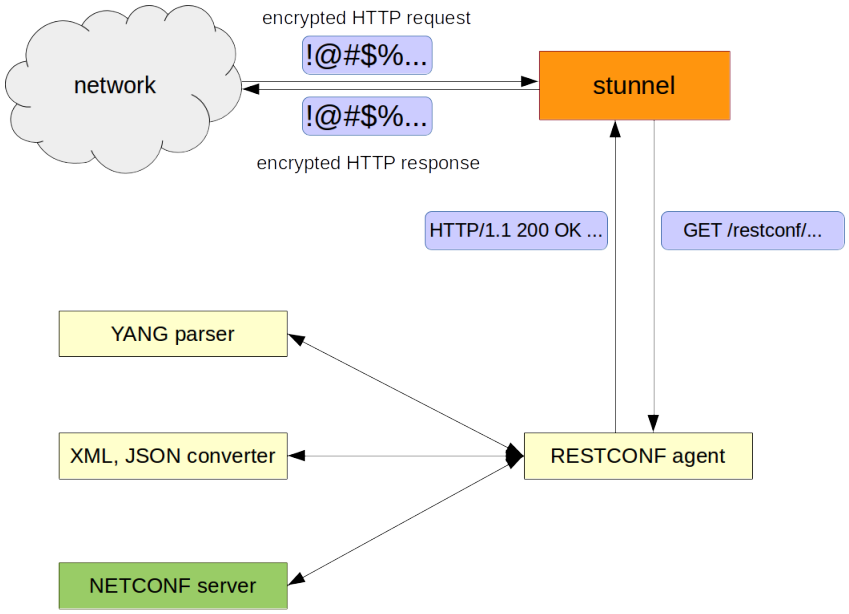

Figure 5.2: Message flow

2. In this step we assume that the RESTCONF agent has been started by `stunnel` in the first step. RESTCONF agent reads and parses the decrypted HTTP message into a format understood by the RESTCONF agent internals – this is a structure containing the type of HTTP method, information about the resource identifier and HTTP headers and a string containing the HTTP message body.

The RESTCONF agent then analyzes the information in the HTTP request structure. Based on this information, it either chooses to create RESTCONF HTTP response and send it back to the client immediately or it decides that it is necessary to communicate further with the NETCONF server. The following steps describe only the latter option as the former option execution ends here.

3. Since the RESTCONF agent needs to communicate with the NETCONF server the HTTP RESTCONF request needs to be converted into NETCONF rpc message. If the HTTP request does not contain a body, the RESTCONF agent will not need

to convert that body from JSON into XML and the request is resolved without the services of XML–JSON converter. If the RESTCONF request does contain a body it will have to be converted from JSON to XML.

To successfully convert it we need to know what YANG models will be necessary for the conversion. This is done by reading the module entries in the HTTP request body which is written in the JSON format.

Then the schemes of the required YANG modules are requested from the NETCONF server by using the `get-scheme` NETCONF operation and these schemes are parsed by the YANG parser component of the RESTCONF agent. More information on how the parser operates can be found in Chapter 6. The result of parsing the YANG schemes is a list of structures which represent the YANG models, these are later passed to the JSON–XML converter.

If the HTTP request contains no body, the JSON–XML converter will not be required in this step, though it might still be required later when the result of NETCONF rpc is to be converted back to JSON format.

4. Next, the JSON message body has to be converted into XML NETCONF rpc body with the help of parsed YANG modules. The XML–JSON converter component takes care of this conversion. For more information on how the converter works see the Chapter 7.

The result of this conversion is the body of a NETCONF rpc request for the server. The RESTCONF agent has to compose the NETCONF rpc from the HTTP request, HTTP resource identifier, HTTP request headers and this NETCONF rpc message body which is now in the XML format.

5. The NETCONF rpc created by RESTCONF agent is sent to the NETCONF server either via UNIX sockets or the D-Bus communication mechanism using the functions provided by the Netopeer server. These were described in the previous Chapter 4. The result of this operation is a structure representing a

NETCONF reply.

6. As a next step the NETCONF rpc reply body – if any – has to be converted back from XML into JSON. It is passed to the XML–JSON converter along with any YANG modules required for its conversion (these are mentioned in the namespace attributes in the XML body of the response) and a JSON document is the result of this operation.

7. Last, the final RESTCONF HTTP response is created based on the result of the NETCONF operation and the parameters of the HTTP RESTCONF request. If an error occurred an error message is generated with an explanation instead and sent back to the client. The format of error messages is described in the RESTCONF draft and more information on it can also be found in Chapter 3.

5.3 Legacy ideas about server design

Throughout the planning stage of the thesis the ideas about implementation of the RESTCONF server have changed. The original idea about how the RESTCONF server implementation was intended to look like included other tools as well as a different overall strategy.

The original design of the RESTCONF server consisted of the Apache `httpd` web server which would serve as a container for our user implemented `httpd` module and a local NETCONF server. The RESTCONF module would have been started for any incoming RESTCONF communication and would have communicated with the local NETCONF server through the use of UNIX sockets or the D-Bus system much like the agents in the scheme of the Netopeer server. This design is described best by the Figure 5.3.

This server design would have otherwise operated similarly to the current implementation – parsing the incoming RESTCONF communication, translating it into NETCONF rpcs and forwarding them to the local NETCONF server while translating the NETCONF responses back into RESTCONF response and sending them back to

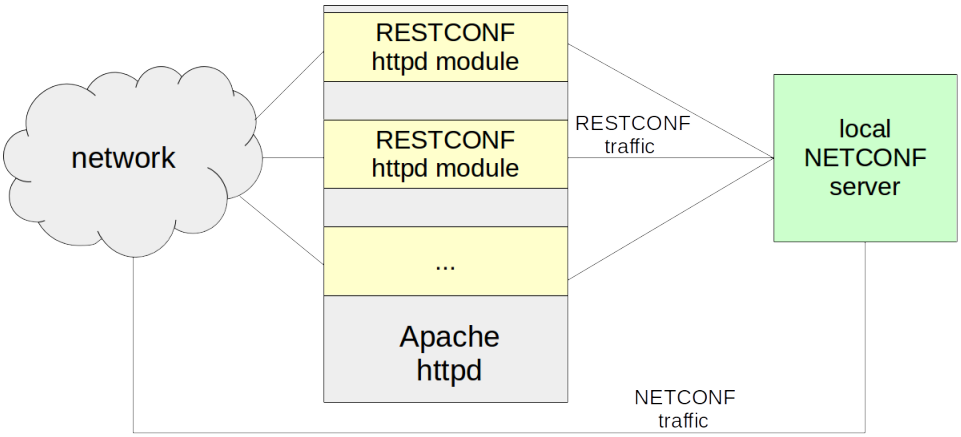

Figure 5.3: RESTCONF legacy implementation design

the client.

6 YANG parser

This chapter describes how our YANG parser works and comments on how it could have been implemented better. In the end, it also goes through the usage of the parser and shows how to run the examples available from the attached DVD.

As mentioned before in the Chapter 2, at the time of writing this thesis there were no parsers to be found for YANG language implemented in the C programming language. There are however parsers in other languages, for example the `pyang` parser which is written in the Python programming language and is also used by the JNC – Java NETCONF client. Before implementing the YANG parser we have considered utilizing the `pyang` parser by writing a set of C bindings for it. The reason this method has been discarded is that it would have been too complicated for our purpose – which is only converting JSON messages into XML messages based on a data model defined in YANG.

Our parser does not need to understand all aspects of the YANG data modeling language. It only needs to support basic expressions, for instance the `container` or `list` statements, support groupings which are often used in data models associated with the NETCONF protocol and support augmenting existing data models by using the `augment` directive. It might be convenient to have other features supported but that would be a convenience for mostly other applications of the YANG parser.

6.1 Parser design

The YANG parser described in this chapter is written by hand. The implementation is rather simple and the workflow consists of sequential word reading and filling in an internal structure that represents the YANG model. The workflow is shown in Figure 6.1. The reading is also recursive – the parser takes into account sections of text which are enclosed in quotes, brackets or parenthesis.

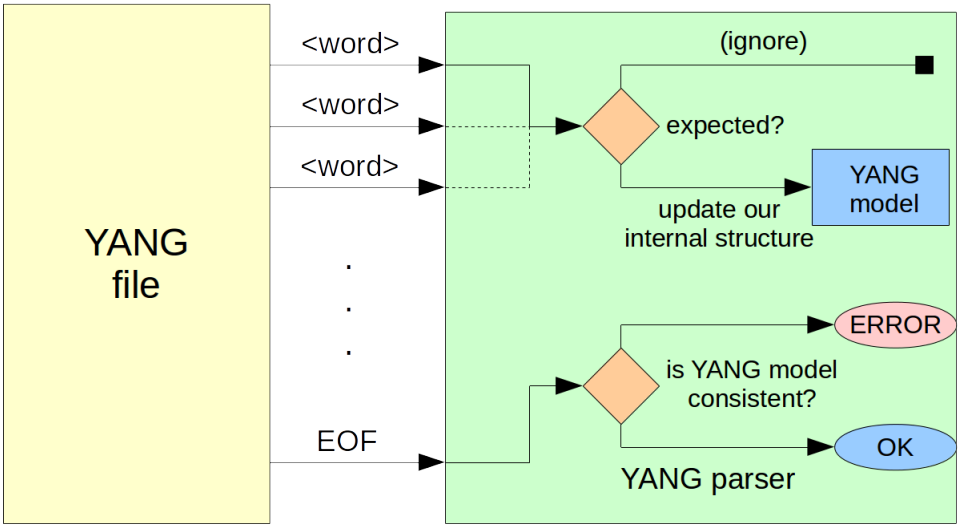

Figure 6.1: YANG parser design

The implementation consists of three sets of functions:

1. **Parsing functions** – these take care of reading the YANG model file word by word. They contain various utility functions for reading words, lines, YANG expressions and other functions for working with file streams. There also exist functions for parsing data models both from strings and files. An example of a function which belongs in this category would be the following:

   ```
   module* read_module_from_file(FILE* file);
   ```

2. **Modeling functions** – this set of functions is used for modeling the internal YANG data structures which represent a parsed data model. Such internal YANG structure is the result of parsing a YANG model. There exists functions for creating module structures, groupings structures and other utility structures as well as functions for copying and cleaning up these structures. An example function from this category would be the function for creating a YANG node (a YANG node in this context might

be a `container` declaration):

```
yang_node* create_yang_node(char* name,
        node_type type, char* value);
```

3. **Utility functions** – the rest of functions implemented as part of the YANG parser serve mainly for logging and debugging purposes. There are functions for printing the internal structures in a simple format to the standard output or logging them with the `syslog` program. An example of a function which belongs in this category would be:

```
void print_module(const module* mod);
```

6.2 Notes about parser implementations

This section describes what we learned about the methods of parser implementation. There are multiple methods commonly employed when creating parsers for formal languages. Each of these methods carries some advantages and some disadvantages. The following list attempts to shed some light on the features of these methods:

Method 1. – Implementing a parser by hand is a slow and possibly error-prone method. However, the outcome of such parser implementation might be an efficient parser with clean design especially if such parser is written by a programmer experienced in this area. Our YANG parser is written purely by hand though the reasons for this approach are different. These are described later in this section.

Method 2. – Generating a parser from a formal grammar. This is a method often used with simpler languages (though not at all restricted to simple languages only). It yields a quickly generated parser that can be used by other applications to parse a language that is defined by some kind of formal grammar – often the BNF grammar[1]. The YANG modeling language features an ABNF grammar – an Augmented BNF grammar for syntax specifications [17].

1. BNF – the Backus-Naur Form or the Backus Normal Form is one of the two main formal grammars used today.[16]

Method 3. – Modify a generated parser by hand. A method which might yield a quick first parser implementation though the subsequent parser modification may take time depending on the complexity of the generated parser.

As has been mentioned above our YANG parser has been created by hand. When deciding which one of the above described methods we should use to implement the parser we have investigated our options and we made the following observations:

- That the set of YANG features our parser needs to support in order for XML-JSON converter to perform its functions properly is much smaller than the set of all features of the YANG modeling language. This meant that generating a parser might not be the best option due to the set of unused features in the parser. This also rules out the option number three – modifying a generated parser by hand.

- That the process of generating a parser requires an input in the form of a formal grammar. This formal grammar has to be one that is supported by the tool or set of tools of choice. The YANG modeling language comes with an ABNF grammar provided as has been mentioned before.

  We have found that there are no suitable tools for generating parsers in the C language from the formal grammar provided by the YANG RFC document. As we later found, this observation has been erroneous. After the YANG parser has already been implemented we've found that not only there are tools that are capable of generating parsers in the C language based on an ABNF grammar – one such is listed briefly below – but also that the grammar can be converted into another format which we could utilize to generate a parser with different set of tools. Such conversion would preferably be done by a software tool as the formal grammar is rather lengthy and converting it by hand would be prone to errors.

Among the tools capable of generating a parser in the C programming language with the input of an ABNF grammar is the tool APG available at:

`http://www.coasttocoastresearch.com/docc/index.html`

As a last note in this section it remains to be said that our perception of how parsers should be implemented has changed since the parser has been finished. Even when implementing a parser by hand the naive approach to read the model file word by word and filling an internal data structure continuously has not worked out very well limiting the functionality and extensibility of the parser. A better approach would be to first tokenize the input file – assign a token to each syntactically important standalone piece of information in the file – and then parse the sequence of tokens into an internal structure based on the formal definition of the language.

6.3 Parser use cases

The YANG parser is also available as a standalone library though for the purpose of the RESTCONF server it is integrated in the RESTCONF agent and the source files are available from the Netopeer project in the RESTCONF development branch. The Netopeer project and specifically the Netopeer server part also contains examples that demonstrate how the parser works. This chapter goes through these examples – how to obtain, compile and run them – and also shows how to pass a custom YANG module to the YANG parser to see how the parser operates on real data.

To obtain the Netopeer project which contains the examples follow the steps in Chapter 8 Section 8.1. Alternatively, the Netopeer server with the YANG parser examples is also available from the virtual image file appended to the thesis. Chapter 8 Section **??** explains how to get the virtual image file operational and also explains where the RESTCONF server files are available.

Once in a directory which has available the `yang_example` program it can be executed without parameters to print a help message or can accept a single parameter to demonstrate the functionality of the parser as shown below:

```
$ ./yang_example example1
...
$ ./yang_example my_data_model.yang
...
```

The `yang_example` program accepts either a prepared example identifier – a string in the form of "exampleX" where 'X' is a number of 1 to 3 – or a path to a yang data model file.

The YANG parser is a library and the `yang_parser` program is merely an example program which uses the library. To test the library to the extremes it is necessary to implement a program which uses the parser in the way the user wishes to use it.

There are available three distinct examples for the above program labeled `example1` to `example3`. These take YANG data models of different complexity and parse them using the YANG parser. Then each of them prints the internal parser structure that shows how the parser understands the YANG data model file and which YANG statements have been ignored. Each example also prints a short description of what it does.

It is also possible to supply the example program with a file name that leads to a YANG data model. The example will attempt to parse the data model file and will print the resulting internal structure.

6.4 Supported and unsupported features

Due to the limited requirements of the JSON and XML converter, the parser does not support a number of YANG features defined in the YANG RFC document. This chapter details which YANG features are recognized and which are not.

The following YANG features are recognized by our parser:

- **Module declarations** – the parser recognizes the basic module declarations. Only the name of the module, its namespace and prefix are important for the purpose of XML-JSON converter.

- **All node types** – The parser supports all node types defined by the YANG RFC – the `container`, `list`, `leaf` and `leaf-list` nodes. These are crucial for basic data modeling.

- **Data types** – The parser takes notice of all data types of `leaf` or `leaf-list` nodes. These are necessary for the XML-JSON converter since JSON saves the type of variables while for XML documents the type is only denoted by the YANG data model. Thus, the data types are the main reason for building a custom XML-JSON converter that depends on YANG data models.

- **Reusable node groups** – Groupings are constructions that define often used clusters of node declarations that can be referenced anywhere in the data model by using the `uses` statement.

- **Choices** – The `choice` and `case` statements are used to define and choose between two incompatible node sets in a data model.

- **Augment declaration** – This declaration is used to augment existing data models without the need to modify them. It is commonly used in various data models defined by IETF and is thus necessary for us to support. An example of a data model which augments the `ietf-system.yang` data model might be the `ietf-system-tls-auth.yang` model.

The following YANG features defined in the RFC document are currently unsupported by YANG parser mostly because of large time requirements and complexity:

- **Submodules** – Submodules define modules inside other modules. This is an uncommonly used functionality.

- **Derived type definitions** – This allows the creator of a YANG module to define his own types from the existing types by using the `typedef` directive. This functionality would require a change in how the YANG parser understands types and most derived types are currently handled as string types by the XML-JSON parser.

- **RPC definitions** – The parsing of these definitions would be necessary to implement the optional `/operations` resource of the RESTCONF server.
- **Augment specifications** – The keyword `when` is ignored in the augment statements. This means that the augmenting statements are always in effect for our parser.
- **Descriptions, revisions and other elements** – YANG parser does not keep track of many pieces of information often present in the YANG data models. An example would be the description element which is used to append a description to a module or a node. This feature is not needed for the functionality of the RESTCONF server.

7 XML and JSON converter

This chapter outlines the methods used in development of XML-JSON converter for our purposes, that is, the purpose of converting RESTCONF requests into NETCONF requests and NETCONF responses into RESTCONF responses.

7.1 Converter basics

The XML-JSON converter is a library which requires a YANG data model to work. The core of the library is a pair of recursive algorithms which receive either JSON or XML, RESTCONF or NETCONF message respectively, and traverse the document trees converting the messages from the innermost elements out to the document root. With some elements of the document the converter consults the parsed YANG data model to see what type a value should be or what namespace an element should be assigned.

To consult the YANG data model the converter needs to specify a node inside the model structure. For this purpose, the converter constructs a string based on its current location in the document it is parsing. Below is an example of how this string is constructed when converting an XML response to JSON response:

```
<!-- simple example representing an XML message -->
<system>
  <authentication>
    <tls>
      <cert-maps>
        <cert-to-name>
          <id>1</id>
          <!-- the rest has been omitted -->
        </cert-to-name>
      </cert-maps>
    </tls>
  </authentication>
</system>
```

To specify the `id` element of the above message the XML-JSON converter would construct the following string:
`system:authetication:tls:cert-maps:cert-to-name:id`

This string would then be passed to function which would return a structure which contains the type of node the element is (e.g. a `leaf` or a `leaf-list` node type) the type of that node (e.g. a `string` or `uint8`). The converter then converts the element to a fitting JSON object type based on the obtained information.

As a design oversight, the node identifier string does not differentiate between nodes from different modules (put into a single module with `augment` declaration) which have the same name. However, augmenting modules that add nodes with the same name are rare. It is recommended that the format of this node identifier be replaced with a format which differentiates between module namespaces (e.g. format similar to XPath where `':'` are replaced with `'/'`).

7.2 Converter use cases

As with the YANG parser there is an example program ready to test the XML-JSON converter inside the source directory of the RESTCONF server. To access this directory or to compile the source code the directions in Chapter 8 can be followed as with the YANG parser examples.

Once in the directory which contains the `converter_example` program it can be executed without parameters to print a help message or can accept a single parameter to demonstrate the functionality of the converter as shown below:

```
$ ./converter_example example1
...
```

The `converter_example` accepts a single parameter – a string in the form of "exampleX" where 'X' is a number of 1 to 3.

As with the YANG parser, the converter is a library and user is

encouraged to construct his own examples for conversion.

7.3 Supported and unsupported features

The set of rules that should be utilized for conversion between JSON and XML documents based on YANG data models is defined by a draft available at the IETF website: Modeling JSON Text with YANG [18].

The basic rules defined in the draft are supported by the XML-JSON converter. The cases not covered by our converter stem from nonoptimal YANG parser design – an example might be bad conversion of a `leaf` node, that has a complex type defined with the `typedef` directive in the YANG data model.

8 RESTCONF server

The previous chapters described in detail the components of the RESTCONF server. This chapter takes the RESTCONF server as a whole and shows how to start and operate the server. It is a quick start guide for those who aim to set up their server locally and also describes how to simulate a simple RESTCONF client.

Those who would want to avoid the installation and server set up can make use of the virtual image present on the DVD attached to this thesis. The instructions on how to start up the virtual image and how to run a prepared set of examples are available in Appendix B.

8.1 RESTCONF server installation

To start and operate the server locally, it first has to be downloaded. As there is no prepared distribution for the Netopeer project, it has to be compiled and installed manually. The `git` program has to be installed in order to download the source code of the project. To install Netopeer, the `libnetconf` library and the `stunnel` program must be installed on the system.

Files available in Netopeer source code will be required to patch the `stunnel` program. These can be obtained by issuing the following commands:

```
$ git clone https://code.google.com/p/netopeer/
$ cd netopeer
$ git checkout restconf-impl
```

If problems occur, consult the *checkout* web page at the project site:
`https://code.google.com/p/netopeer/source/checkout`

8.1.1 `libnetconf` installation

To compile the Netopeer server, first, the `libnetconf` library needs to be installed. The source code to `libnetconf` library can

be cloned and then installed by issuing the following commands:

```
$ git clone https://code.google.com/p/libnetconf/
$ cd libnetconf
$ ./configure --enable-tls
$ make
# make install
```

Any issues will be announced by the `configure` script. These need to be resolved in order to compile the library. Consulting the `README` and `INSTALL` files in the project directory is also recommended. The `make install` step requires root privileges, which is signified by the `'#'` character in front of the command.

8.1.2 `stunnel` installation

The source code of the `stunnel` encryption wrapper is available for download at:
`https://www.stunnel.org/downloads.html`

To install `stunnel`, it has to be patched to work with the Netopeer project. Follow the instructions in the `stunnel/README` file in the Netopeer server directory (the `server` directory under the Netopeer project). This file and the `README` file in the `stunnel` project directory will also lead the user through the installation process.

8.1.3 Netopeer installation

To compile the Netopeer server make sure to read the `README` file provided in the directory `netopeer/server` which contains the list of packages required for Netopeer installation. The installation of these packages is platform specific and beyond the scope of this chapter. We will be compiling the server with the `--enable-tls` option enabled and thus we will need the packages required for TLS authentication.

The `jansson` library is also needed in addition to the libraries mentioned in `README` files. To install it for Ubuntu-based systems, use this command:

```
# apt-get install libjansson-dev
```

When all the necessary packages are installed, compile the server by issuing the following commands:

```
$ ./configure --enable-tls
$ make
# make install
```

The `configure` script checks whether all the necessary tools and libraries are available. The `make` commands then compile and install the server into the correct location. Note that the last part of the command: `make install` requires superuser privileges. It is necessary to resolve any problems which arise when configuring the program – the `configure` script notifies the user of any that occur (e.g. patching the `stunnel` program if not done in the previous step) – and when the installation finishes, the RESTCONF server is almost ready to be used.

The `cfgsystem` transAPI Netopeer module needs to be installed after the Netopeer server for the examples to work. Execute the following commands from the `netopeer` directory:

```
$ cd transAPI/cfgsystem
$ ./configure
$ make
# make install
```

All missing packages reported by the `configure` script need to be installed.

Correct certificate setup is necessary for the RESTCONF server to work. To configure certificates, it is recommended to read the `README` files in the `stunnel` and Netopeer projects. For the purpose of RESTCONF examples, two files are provided in the `cert` directory on the attached DVD. The `client.pem` file should be copied into the `~/.netopeer-cli` directory and the `ca_rootCA.pem` file should be copied into the `~/.netopeer-cli/certs` directory for the examples to work.

8.2 Startup

To make use of the RESTCONF server implementation, the NETCONF server has to be started. Use the following command to start the Netopeer server:

```
# netopeer-server -d
```

The server accepts multiple options that can be seen by issuing the `-h` option and it must be run as a superuser. The `-d` option shown in the example above daemonizes the server process, putting it into background. To increase the amount of logging done by the server, the `-v X` option can be used, where 'X' is a number from 0 to 3 that specifies the logging level.

By default the server and its agents log the events that occur by using the `syslog` program. The log file location varies by operating system, but for Ubuntu-based operating systems it can usually be found at `/var/log/syslog`.

The Netopeer agents (both RESTCONF and NETCONF) are never started manually. They are started by the `stunnel` program when a connection is accepted at one of their configured ports.

8.3 Shutdown

If the server has been started without the `-d` option, then shutting it down consists of issuing a SIGINT signal to the program by pressing `Ctrl+C` on the keyboard. This terminates the program while giving it the chance to clean up all resources required by the server.

If the server has been started with the `-d` option, its process ID has to be found by this command:

```
$ ps -efl | grep netopeer-server
```

and then it can be killed by

```
# kill <netopeer-server-process-id>
```

In the unlikely case that the shutdown is unsuccessful or if the program has been terminated by the SIGTERM signal (e.g. by issu-

ing a `kill -9` command), the server may quit without releasing all shared resources. This may cause errors the next time the server is started, since it requires several shared resources for its functioning. In the case that the server fails to start, try manually deleting the following files:

```
/dev/shm/sem.NCDS_FLOCK_*
/var/lib/libnetconf/libnetconf_sessions.bin
```

All applications using the `libnetconf` library should be stopped before manually removing these files.

8.4 RESTCONF clients

Any program capable of creating HTTP requests that gives the user the ability to specify request parameters (e.g. HTTP method, resource identifier, HTTP headers and body), and that can handle TLS authentication can serve as a RESTCONF client. The popular HTTP client program `curl`[19] was used in the development of the server.

The examples provided with the source codes of the server utilize the `curl` program and the usage can be seen by opening the shell scripts. An example of `curl` usage would be:

```
curl -k -1 -E ~/.netopeer-cli/client.pem \
    --cacert ~/.netopeer-cli/certs/ca_rootCA.pem \
    --capath ~/.netopeer-cli/certs/ \
    https://127.0.0.1:8080/restconf/data
```

9 Conclusion

At the beginning of the thesis, it has been explained that there is need for a network management protocol that is simpler than NETCONF and provides a subset of its capabilities. This protocol is RESTCONF and the thesis has attempted to design a RESTCONF server, implement it and discuss the results of our implementation.

The first RESTCONF server design has featured a standalone NETCONF server and a RESTCONF agent that would be implemented as a module to the Apache `httpd` web server. This design has later been redone due to uncertainty about authentication specification (which is, as of now, missing in the RESTCONF draft) and the possibility of code duplication. In the final design, both the standalone NETCONF server and Apache `httpd` have been swapped for the open source Netopeer NETCONF server and the `stunnel` encryption wrapper provides the TLS functionality that is required.

We have implemented the RESTCONF server as an add-on to the existing Netopeer server. The server consists of three components and these are: the YANG parser, the XML-YANG converter and the RESTCONF Netopeer agent.

The RESTCONF agent is the main component which serves RESTCONF requests and the YANG parser along with the XML-JSON converter help with translating the RESTCONF requests into the NETCONF protocol and back. The functionality of the XML-JSON converter depends on the YANG parser.

Overall, the application can be downloaded as part of the Netopeer project and when built, it provides the capabilities of both the RESTCONF server and the original Netopeer NETCONF server. The implementation is thus part of a successful open source project and provides a foundation for future RESTCONF related technologies.

As a last objective, both, the design and the implementation of the server have been discussed, and guidelines have been provided

on future and better development of a RESTCONF server. The guidelines for server components are in their respective chapters and the Section 9.1.

9.1 Future of the project

The discussion of design and implementation of the RESTCONF server and its parts, has helped to identify areas suitable for improvement and parts of our software solution that might benefit from implementing additional optional features.

One of these is the YANG parser component of our RESTCONF server. The parser design has been found problematic and is the best candidate for improvement. For future implementations it is recommended that the parser is implemented not only as a library but also as a standalone component to the server, which loads data models at startup and then only serves their internal representations to the RESTCONF agents when requested.

Additional areas for improvement are discussed in the Section 3.4, which contains the list of optional RESTCONF features and reasons why they were left unimplemented, and at the end of Chapters 6 and 7, which contain the lists of supported and unsupported features of RESTCONF server components.

Bibliography

[1] R. Enns, M. Bjorklund, J. Schoenwaelder and A. Bierman. *Network Configuration Protocol (NETCONF)*. IETF, June 2011. [online] [cit 14.4.2015]. Available at <https://tools.ietf.org/html/rfc6241>.

[2] et al. Brian Hedstrom. Protocol Efficiencies of NETCONF versus SNMP for Configuration Management Functions. Master's thesis, University of Colorado, Boulder, 2011.

[3] et al. Stefan Wallin. Automating Network and Service Configuration Using NETCONF and YANG. In *25th Large Installation System Administration Conference*. USENIX.

[4] A. Bierman and NETCONF Working Group. *RESTCONF Protocol*. IETF, February 2014. [online] [cit 14.4. 2015]. Available at <https://tools.ietf.org/html/draft-bierman-netconf-restconf-04>.

[5] FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures* [online] [cit. 15.3. 2015]. Available at <https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>.

[6] R. Krejčí and CESNET. *Netopeer* [online] [cit. 14.4. 2015]. Available at <https://code.google.com/p/netopeer/>.

[7] R. Fielding, et al. *Hypertext Transfer Protocol (HTTP/1.1)*. IETF, June 2014. [online] [cit 24.4.2015]. Specification is split into multiple documents available at
<https://tools.ietf.org/html/rfc7230>
<https://tools.ietf.org/html/rfc7231>
<https://tools.ietf.org/html/rfc7232>
<https://tools.ietf.org/html/rfc7233>
<https://tools.ietf.org/html/rfc7234>
<https://tools.ietf.org/html/rfc7235>.

[8] David Gourley et al. *HTTP: The Definitive Guide*. O'Reilly Media, 2002.

[9] Joe Fawcett et al. *Beginning XML*. Wrox, 2012.

[10] Srai Srinivas Sriparasa. *JavaScript and JSON Essentials*. Packt Publishing, 2013.

[11] M. Bjorklund and Tail-f systems. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. IETF, October 2010. [online] [cit 28.4.2015]. Available at <https://tools.ietf.org/html/rfc6241>.

[12] S. Chisholm, et al. *NETCONF Event Notifications*. IETF, July 2008. [online] [cit 28.4.2015]. Available at <https://tools.ietf.org/html/rfc5277>.

[13] R. Krejčí and CESNET. *libnetconf* [online] [cit. 28.4. 2015]. Available at <https://code.google.com/p/libnetconf/>.

[14] David A. Wheeler. *Secure Programming HOWTO* [online] [cit. 4.5. 2015]. Available at <http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/sockets.html>.

[15] Richard Blum Jon Masters. *Professional Linux Programming*. Wiley / Wrox, 2007.

[16] Dick Grune. *Parsing Techniques: A Practical Guide*. US: Springer, 1999.

[17] D. Crocker, et al. *Augmented BNF for Syntax Specifications: ABNF* [online] [cit. 20.5. 2015]. Available at <https://tools.ietf.org/html/rfc5234>.

[18] L. Lhotka and CZ.NIC. *libnetconf* [online] [cit. 17.5. 2015]. Available at <http://tools.ietf.org/html/draft-lhotka-netmod-yang-json-02>.

[19] curl development team. *curl tutorial* [online] [cit. 14.4. 2015]. Available at <http://curl.haxx.se/docs/manual.html>.

A DVD attachment contents

The attached DVD contains the source codes of the modified Netopeer server as well as the virtual image with the RESTCONF server already set up. Apart from that it also contains the text of the thesis and the pictures used in the thesis. Everything is contained in the following directories:

- *virtual-image* – Contains the virtual image files. To learn how to start this virtual image up, see the Appendix B.
- *netopeer* – Contains the source codes for the RESTCONF server. These are present for documentation purposes. To successfully install the RESTCONF server, see the instructions in chapter 8.
- *cert* – Contains certificate files for the client.
- *thesis* – Contains the thesis text in the `.pdf` and `.tex` format plus pictures used in the thesis in the `.png` format.

B Virtual image instructions

A virtual image in the form of a virtual hard drive file is available on the DVD attached to the thesis. The *VirtualBox* software is required to make use of it. To install virtual box software on the Ubuntu Linux operating system, issue the command:
`sudo apt-get install virtualbox`
or its equivalent for other Linux distributions. Note that a password may be required to install the program.

Once the program is installed, find the `Thesis.zip` file on the attached DVD. Unzip this file into a directory of choice. The archive `Thesis.zip` only contains a single `Thesis.ova` virtual archive. Its location is of importance as it will be utilized in the following steps.

Start the *VirtualBox* software. Click 'File' and select 'Import Appliance' from the drop-down menu. On the pop-up screen select the location of the `Thesis.ova` file and then click 'Next' and 'Import'.

The virtual machine is now ready. Changes to its configuration can be made by clicking on the 'Settings' button in the upper panel and the virtual machine can be started by pressing the 'Start' button next to the 'Settings' button.

Once started, the virtual machine will greet the user with a login screen. The preconfigured 'Thesis' user is the one with the files for the RESTCONF server and superuser privileges:
`username:` `thesis`
`password:` `thesis1`

After logging in successfully, open the terminal window by double-clicking on the terminal icon on the desktop. Navigate to the Netopeer server directory by issuing the command:

```
$ cd /home/thesis/netopeer/server
```

This directory contains the compiled Netopeer server and the examples to demonstrate the functionality of YANG parser, XML-JSON

converter and the RESTCONF server itself. The examples for YANG parser and XML-JSON converter are compiled C programs while the examples for the RESTCONF server are in the form of `bash` scripts.

To run the examples for the YANG parser, simply enter the following command:

```
$ ./yang_example exampleX
```

where the 'X' should be replaced by number in the interval of 1 to 3. Each of these examples explains what input it takes and then

Similarly, to run the examples for the XML-JSON converter enter the following command:

```
$ ./converter_example exampleX
```

The `converter_examples` program provides the same example naming as the `yang_example` program.

To run the examples on the RESTCONF server, make sure the server is running by issuing the following command (and enter the 'thesis1' password):

```
# netopeer-server -d 3
```

Then, the RESTCONF examples can be executed. The example scripts are in the form `restconf_example_<case>.sh` where <case> is the use case the given example demonstrates. To run a given example, issue the following command:

```
$ ./restconf_example_simple_get.sh
```

The RESTCONF examples support the `-h` option which explains in detail what functionality the example demonstrates.