


Technische Universität München
Department of Informatics

Bachelor's Thesis in Informatics

# Implementing and analysing the Interface to the Routing System in a Software Defined Network

Daniel Kowatsch


Technische Universität München

Department of Informatics


Bachelor's Thesis in Informatics

# Implementing and analysing the Interface to the Routing System in a Software Defined Network

# Implementierung und Analyse des Interface to the Routing Systems in einem Software Defined Network


| | |
|---|---|
| *Author* | Daniel Kowatsch |
| *Supervisor* | Prof. Dr.-Ing. Georg Carle |
| *Advisor* | Edwin Cordeiro, M. Sc. |
| *Date* | August 16, 2016 |

Informatik VIII
Chair for Network Architectures and Services


I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, August 16, 2016

Signature

**Abstract**

This thesis aims to implement the route-add and route-delete Remote Procedure Calls (RPC) of the Interface to the Routing System (I2RS) in a Software Defined Network (SDN). This is done in order to verify the specification of I2RS and to find potential errors within it. Our Implementation consists of two components, an agent and a client. The client is implemented as an independent Python script. It uses ydk-gen to generate a Python API for I2RS. The agent is a module of the OpenDaylight project. It interacts with other modules like the one for BGP or the one for NETCONF to fulfill the requirements of I2RS. The test environment for the implementation is a SDN. Our implementation of the I2RS client is not yet capable of interacting with an agent while our agent can process the RPCs but still contains a bug regarding the route propagation. The thesis also found two problems with the I2RS specification. The more important one is about a missing structure for storing next hops. It makes creating a working implementation according to the here examined specification of I2RS impossible. But despite these problems, I2RS is developing towards a practicable, standardized interface.

**Zusammenfassung**

Diese Arbeit implementiert die Remote Procedure Calls (RPC) route-add und route-delete für das Interface to the Routing System (I2RS) in einem Software Defined Network (SDN). Das hat den Zweck die Spezifikation von I2RS zu überprüfen und mögliche Fehler zu entdecken. Dafür sind zwei Komponenten erstellt worden. Die erste ist ein Client, die zweite ein Agent. Der Client wird als unabhängiges Pythonskript implementiert. Er nutzt das ydk-gen Tool zur Erstellung einer Python-API für I2RS. Der Agent dagegen ist ein Modul des OpenDaylight Projektes. Er interagiert mit anderen Modulen von OpenDaylight, welche für die Implementierung von BGP und NETCONF zuständig sind. Unsere Implementierung wird anschließend in einem SDN getestet. Der Client ist im Moment noch nicht im Stande mit einem Agenten korrekt zu interagieren. Der Agent kann die RPCs verarbeiten, hat aber noch einen Bug beim Propagieren der Routen zu anderen Geräten. Die Arbeit hat zwei Probleme mit der Spezifikation von I2RS entdeckt. Das schwerwiegendere von beiden befasst sich mit einer fehlenden Datenstruktur zum Speichern von Next Hops. Durch dieses Problem ist es für die Version von I2RS, die wir betrachtet haben, nicht möglich gemäß der Spezifikation zu funktionieren. Trotz dieser Probleme entwickelt sich I2RS jedoch zu einem praktischen Standardinterface.

# Contents

# List of Figures

# Listings

# Chapter 1

# Introduction

The size and complexity of networks is increasing. They consist of a multitude of different hardware developed by different vendors. This heterogeneity makes management and automation of even the simplest tasks difficult.

Additionally, dynamic data-center networking advances and dynamic routing policies are required, but the forwarding decisions are still made based on static routing policies, relying on defined link costs and route costs. So these policies are not capable of providing real-time responsiveness towards security threats and can not use traffic control to deal with them. A more dynamic policy could for example redirect suspicious traffic to analyzers, honey pots or even drop packets. Another point is the paradigm of separating policy-based decision-making from the actual router. But this also requires more dynamic policies and analyzing of the network, its topology, and network traffic statistics.

In order to achieve these goals, a new standardized interface to the routing system needs to be developed. So the IETF created a working group which is currently designing the Interface to the Routing System (I2RS). [2]

## 1.1 Goals of the thesis

The I2RS working group has already written some drafts and RFCs, but the specification of the Interface to the Routing System also requires an implementation. This needs to be done in order to verify the specification, find potential errors and improve the design where necessary. So the goal of this thesis is to create a proof-of-concept implementation of simple functions. The functions include adding new routes to an already existing Routing Information Base (RIB) and deleting existing routes. Because this implementation will be just a proof-of-concept, these features will be limited to only support IPv4 and IPv6.

Additionally, the thesis aims to use the paradigm of separating control logic from actual forwarding. Software Defined Networking achieves this by introducing controllers, which are separated from the actual forwarding device. The implementation will be tested in a Software Defined Network. This way, a separation can be achieved both on layer 2 and layer 3 of the ISO-OSI model. This should enable an easy exchange of the network.

## 1.2 Outline

Chapter 2 provides the state of the art. This includes an introduction to the Interface to the Routing System and an explanation of the concept of Software Defined Networking, an overview on the OpenDaylight project which is an important building block of the implementation, an illustration of network simulation tools and the discussion of related work. Chapter 3 illustrates the methodology, including development, testing and validation methods. Chapter 4 explains the architecture of the implementation and why some design decisions were made. Chapter 5 presents the results of the tests and an evaluation of the I2RS protocol. Chapter 6 discusses the results of the thesis. Chapter 7 provides the conclusion.

# Chapter 2

# State of the Art

This chapter presents the current state of the art. It illustrates the three fundamental components of this thesis. The first one is the Interface to the Routing System (I2RS). Section 2.1 explains the basics of I2RS. This includes its architecture as well as the most important elements for the implementation. The second fundamental component of this thesis is Software Defined Networking (SDN). Section 2.2 introduces the idea of SDN. The third component is the OpenDaylight project (ODL). Section 2.3 provides an overview of ODL. In order to test the implementation, we simulate a network. Section 2.4 shows network simulation tools. This chapter ends with section 2.5, which illustrates connections of this thesis to related work.

## 2.1 Interface to the Routing System

I2RS is a large topic and includes many specifications and drafts. So this section will focus on the relevant ones for the thesis. The first one is the architecture of I2RS. The second one is the RIB module. It describes the Routing Information Base (RIB) in I2RS and which operations are supported.

One thing to keep in mind is that I2RS is still under development. So some drafts and specifications might change. For this reason we will state the version of every draft.

### 2.1.1 Architecture

The architecture of I2RS is defined in RFC 7921 [3]. It consists of two major components. The first one is the *I2RS agent*. It is also referred to as *agent*. The second one is the *I2RS client* or *client*. In order to understand these components we first have to define some basic terms. This thesis uses the same definitions as RFC 7921 section 2 [3]. So first we

introduce these terms. Next we explain the architecture in more detail. For this purpose we use some examples as illustrations.

#### 2.1.1.1 Terminology

An *application* is defined as software which requires to monitor or to change the network for its services.

A *routing element* is an element which implements "some subset of the routing system" [3]. It does not necessarily implement a Forwarding Information Base (FIB). There are different examples for routing elements. The first one is a router with a RIB Manager and a FIB which runs protocols like OSPF or BGP. The second one is a BGP speaker which acts as a *Route Reflector*.

The term *routing and signaling* refers to the part of a routing element which implements a subset of the Internet routing system. This includes the implementation of protocols like BGP or IS-IS as well as the RIB Management layer.

*Local Configuration* describes how the ephemeral state of I2RS is handled. Proposal draft-ietf-i2rs-ephemeral-state-15 [4] explains ephemeral states in I2RS in more detail. This is out of scope of this thesis so we will no further discuss ephemeral states.

The term *Dynamic System State* refers to data which may be required for I2RS but is not part of the routing and signaling system. This includes dynamically changing information like flow data, statistics or counters.

The *Static System State* is like the Dynamic System State not contained in the routing and signaling system. It includes static information like the specification of queuing behavior for interfaces.

*I2RS service* or *service* describes functions and policies to access states within I2RS. In order to represent services we will use data models. One example of such a service is the *RIB service* which is represented by a RIB data model.

Now we can define the I2RS client and the I2RS agent. The client is an implementation of the I2RS protocol which is capable of interacting with I2RS agents inside for example routers. It can read and modify information from agents. The interactions of a client are not limited to only one application or one agent.

An I2RS agent provides services from a routing subsystem. So it interacts with the routing element. It also supports the I2RS protocol and can communicate with I2RS clients. An example of a client is a command line interface an administrator can use to access the routing system.

A client also has an *identity*. It is unique and every client has one. It is used to identify clients. An identity can also be associated with a state and roles. The roles are part of a

role based access control model in I2RS. We will not discuss this any further because it is out of scope of the thesis.

A client can have a *secondary identity*. Since a client can provide services to multiple applications, the secondary identity can be used to associate operations with applications. It is not interpreted by an I2RS agent.

#### 2.1.1.2 Architecture interactions

The architecture defines interactions between clients and agents. Figure 2.1 illustrates one possible interaction between two I2RS clients and an I2RS agent. In this case Client 1 is part of Application 1 and Client 2 is part of Application 2. Interactions between an application and a client may include the usage of command line interfaces or Remote Procedure Calls (RPC). The specification of these interactions is out of scope of I2RS.

Both Client 1 and Client 2 are communicating with Agent 1. The protocols for communication are NETCONF and RESTCONF for now. Proposal draft-ietf-i2rs-protocol-security-requirements-06 [5] specifies security requirements for this communication. The communication includes the client requesting and modifying data and the responses of the agent. It also contains notifications for events. The YANG data models of a service define the structure of requests, responses and data. The communication is asynchronous to support multiple clients and multiple agents at the same time.

Agent 1 is part of Routing Element 1. Routing Element 1 also contains components for Routing and Signaling, a Local Configuration as well as a Dynamic and Static System State. Agent 1 interacts with these components in order to collect or modify data. Some components also interact with each other. Arrows illustrate possible interactions between components.

In this case, the agent has to use the identifier of the clients to associate clients with the according operations. This needs to be done in order to achieve traceability. RFC 7922 [6] provides an explanation on how to handle traceability in I2RS exactly.

Figure 2.2 illustrates another possible setup of one client, two applications and two agents. The components for Routing and Signaling, Local Configuration, Dynamic System State and Static System State are omitted. They interact with each other and the according agent like depicted in Figure 2.1. Both Routing Element 1 and Routing Element 2 have an I2RS agent.

In this case Application 1 and Application 2 use Client 1 as a remote client. The interactions between both applications and the client are not part of I2RS. This also includes the security of their communication. Client 1 is responsible for the authenticity of Application 1 and Application 2. In order to associate operations with applications the client may use secondary identities. In this case Client 1 is communicating with two

agents. The communication between Client 1 and Agent 1 uses RESTCONF. The one between Client 1 and Agent 2 uses NETCONF.

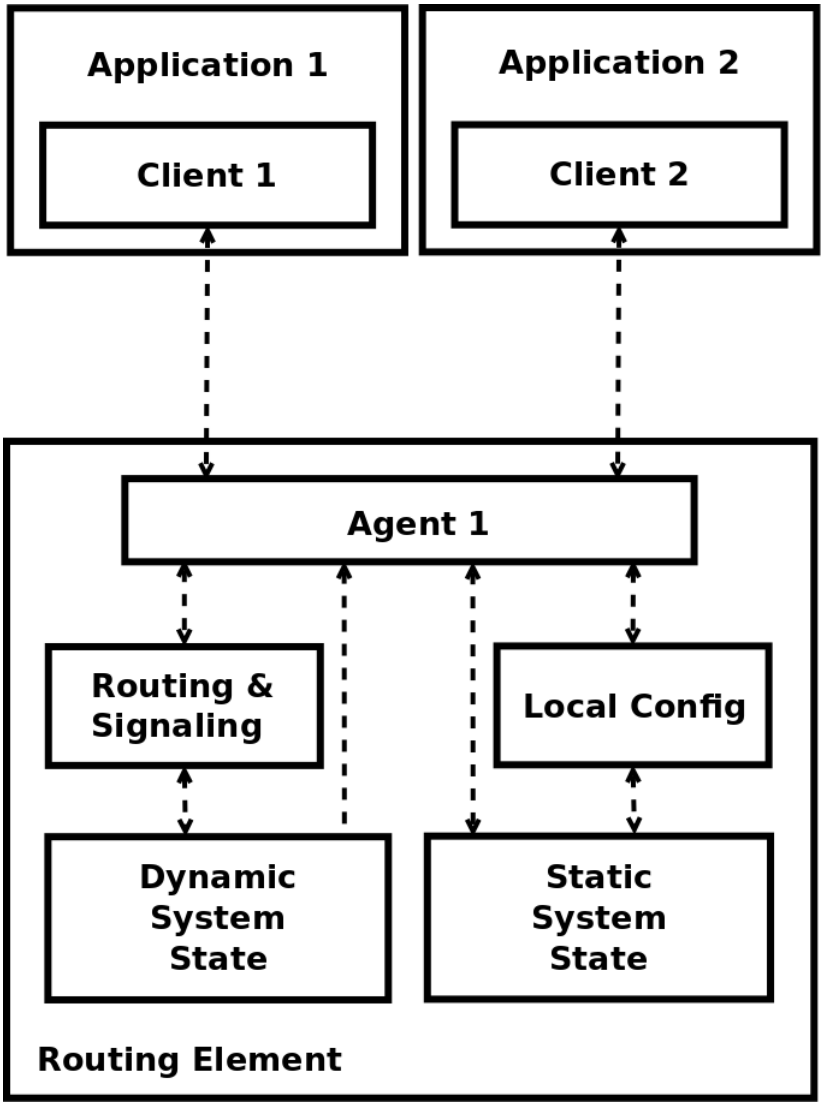


Figure 2.1: Example interaction of two clients with one agent

### 2.1.2 The RIB module

An info model and a data model define the structure of the RIB service. In this thesis we work with draft draft-ietf-i2rs-rib-info-model-08 [7] for the info model and draft draft-ietf-i2rs-rib-data-model-05 [1] for the data model. The info model defines the data structure of the RIB service and its functions. The data model contains a YANG data model derived from the info model. YANG is a data modeling language [8]. It

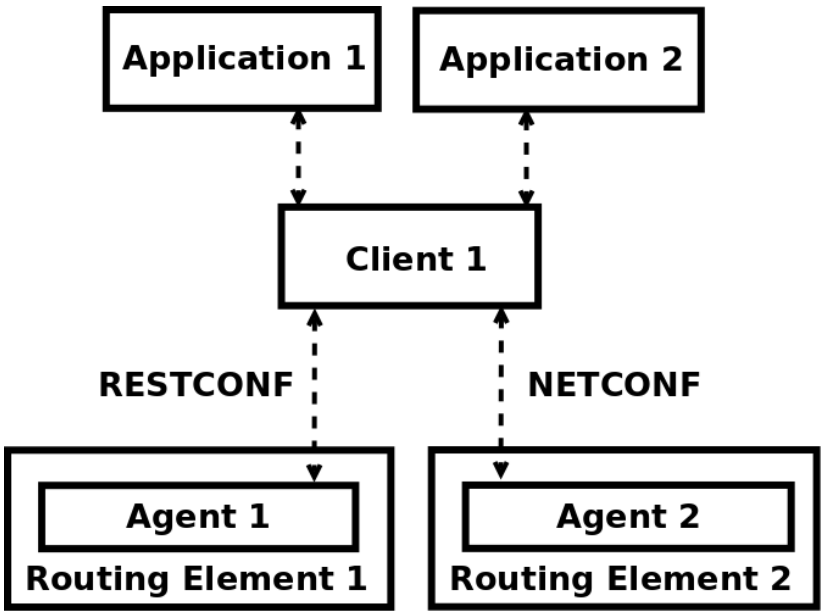


Figure 2.2: Example interaction of a remote client with two applications and two agents

enables modeling structures for configuration and state data. These can be modified with NETCONF.

#### 2.1.2.1 Routing instance

One important part of the RIB data model is the *routing instance*. It represents the data structure for routing information and is managed by the I2RS agent. Listing 2.1 depicts the structure of a routing instance. This includes several components: The first one is a name for the routing instance which is used to identify a routing instance. Therefore it has to be unique. The second one is an optional list of interfaces which can be used in decision making for forwarding packets.

Listing 2.1: Structure of the routing instance [1]

```
+--rw routing-instance
   +--rw name              string
   +--rw interface-list* [name]
   |  +--rw name if:interface-ref
   +--rw router-id?        yang:dotted-quad
   +--rw lookup-limit? uint8
   +--rw rib-list* [name]
      +--rw name            string
      +--rw rib-family      rib-family-def
      +--rw ip-rpf-check?   boolean
      +--rw route-list* [route-index]
         +--rw route-index               uint64
         +--rw match
         |  +--rw (route-type)?
         |     +--:(ipv4)
```

```
|     |  ...
|     +--:(ipv6)
|     |  ...
|     +--:(mpls-route)
|     |  ...
|     +--:(mac-route)
|     |  ...
|     +--:(interface-route)
|        ...
+--rw nexthop
|  +--rw nexthop-id?            uint32
|  +--rw sharing-flag?          boolean
|  +--rw (nexthop-type)?
|     +--:(nexthop-base)
|     |  ...
|     +--:(nexthop-chain) {nexthop-chain}?
|     |  ...
|     +--:(nexthop-replicates) {nexthop-replicates}?
|     |  ...
|     +--:(nexthop-protection) {nexthop-protection}?
|     |  ...
|     +--:(nexthop-load-balance) {nexthop-load-balance}?
|        ...
+--rw route-statistic
|  ...
+--rw route-attributes
|  ...
+--rw route-vendor-attributes
```

It also includes a *router-id* and a *lookup-limit*. A router-id is used to identify the device when interacting with other network devices. It is optional and can be used to differentiate multiple virtual routers in one hardware device. The lookup-limit describes how many levels of look ups can be performed in order to determine if a next hop is reachable.

The last component is a list of RIBs. These can be uniquely identified by their name. A single RIB is also bound to one RIB family. The RIB family describes the type of routes the RIB contains. Some examples are IPv4 or IPv6.

A RIB also contains a list of routes. Each route is identified by its *route-index*. A route has a match condition. A match condition describes which packets this route applies to. The match condition first specifies the type of packet, for example IPv4. Next it defines the direction. So it is possible to match packets based on source address, destination address or both.

Each route must also specify a next hop. A next hop can contain a *nexthop-id* and a *sharing-flag*. The nexthop-id can be used to identify a next hop in case it is shared with other routes. The sharing-flag determines if a next hop is shared. By default it is false. A next hop must contain a *nexthop-type*. The nexthop-type determines what kind of next hop it is. This includes some base nexthop-types like IPv4 which says the next hop is an IPv4 address. It also covers other types like nexthop-chaining or a type for load-balancing.

#### 2.1.2.2 Remote Procedure Calls

Remote Procedure Calls (RPCs) are functions in the data model. The I2RS agent has to provide these methods to clients. The RIB data model defines several which allow interaction with the RIB, including *rib-add, rib-delete, route-add, route-delete, route-update, nh-add* and *nh-delete*.

For adding or deleting RIBs of a routing instance rib-add and rib-delete have to be used. The nh-add RPC adds new next hops to a RIB. The nh-delete RPC deletes one. In order to add a new route to a RIB, the route-add RPC has to be used. It requires the nh-add RPC to be called first in order to create next hops for the routes. If routes need to be deleted the route-delete RPC has to be called. In case a route needs to be updated route-update is useful. This thesis focuses on an implementation of route-add and route-delete. So these will be explained in more detail. Since the nh-add RPC needs to be called before the route-add RPC we will also examine this one.

Like every RPC, the route-add RPC defines an input and an output. Listing 2.2 depicts the structure of the RPC. It omits the structure of a route-list since it is already depicted in Listing 2.1. The input of the route-add RPC contains a list of routes. These routes are defined like routes for RIBs in a routing instance. The client has to receive the nexthop-id of a next hop inside a route by calling the nh-add RPC first. If one of the values inside a route is invalid, adding the route should fail. Examples for such invalid values are a route type which does not match the RIB family type or an invalid route id. The input also includes a RIB name and a return-failure-detail flag. The RIB name is used to determine which RIB the route should be added to. If the specified RIB does not exist, adding routes should fail in all cases. The return-failure-detail flag determines if the error codes for failed routes should be included in the response. This flag is optional and is set to false by default. The output of a route-add RPC contains two counters. The first one counts how many routes have been successfully added to the RIB. The second one indicates the number of failed routes. If the return-failure-detail flag is set, the output also contains a list with information about the failed routes. It includes the route index of every failed route as well as an error code.

Listing 2.2: Structure of the route-add RPC with route-list omitted [1]

```
+---x route-add
|  +---w input
|  |  +---w return-failure-detail?   boolean
|  |  +---w rib-name                 string
|  |  +---w routes
|  |     +---w route-list* [route-index]
|  |        ...
|  +--ro output
|     +--ro success-count     uint32
|     +--ro failed-count      uint32
|     +--ro failure-detail
|        +--ro failed-routes* [route-index]
|           +--ro route-index uint32
```

```
|          +--ro error-code? uint32
```

The schema for the route-delete RPC is depicted in Listing 2.3. The input of the route-delete RPC is identical to the one of the route-add RPC. The specification of route attributes and a next hop is not required for this RPC. Therefore these fields are omitted from the input. This RPC tries to match the routes given in the input to routes in the specified RIB. If a match is found, it will be deleted. There are some cases in which deletion of a route can fail. An obvious one is if a route simply does not exist. Another one is if the requesting client is not allowed to delete the route. The output is also identical to the output of the route-add RPC.

Listing 2.3: Structure of the route-delete RPC with route-list omitted [1]

```
+---x route-delete
|  +---w input
|  |  +---w return-failure-detail?   boolean
|  |  +---w rib-name                 string
|  |  +---w routes
|  |     +---w route-list* [route-index]
|  |        ...
|  +--ro output
|     +--ro success-count     uint32
|     +--ro failed-count      uint32
|     +--ro failure-detail
|        +--ro failed-routes* [route-index]
|           +--ro route-index uint32
|           +--ro error-code? uint32
```

The structure of the nh-add RPC is illustrated in Listing 2.4. The input of this RPC consists of two to four components. The first one is *rib-name*. It specifies to which RIB the next hop has to be added. If the RIB does not exist the RPC fails. The second one is the *nexthop-id*. So far it is not yet specified what this id does in the context of the nh-add RPC because the purpose of this RPC is to add a new next hop to a RIB and to return the nexthop-id. The *sharing-flag* indicates whether a next hop can be shared by multiple routes. The last component is the *nexthop-type*. It defines what kind of next hop it is. It behaves like the nexthop-type inside a next hop of a route. The output of the nh-add RPC informs the client if the operation was successful. When the field *result* is set to *true* the RPC managed to add a new next hop to the RIB. Then it should also return the nexthop-id of the newly added next hop. If the result field has the value *false* the RPC failed. Then it may also contain the cause for failure in the *reason* field.

Listing 2.4: Structure of the nh-add RPC with details about the nexthop-types omitted [1]

```
+---x nh-add
|  +---w input
|  |  +---w rib-name             string
|  |  +---w nexthop-id?          uint32
|  |  +---w sharing-flag?        boolean
|  |  +---w (nexthop-type)?
|  |     +--:(nexthop-base)
|  |     |  ...
```

```
|  |     +--:(nexthop-chain) {nexthop-chain}?
|  |     |  ...
|  |     +--:(nexthop-replicates) {nexthop-replicates}?
|  |     |  ...
|  |     +--:(nexthop-protection) {nexthop-protection}?
|  |     |  ...
|  |     +--:(nexthop-load-balance) {nexthop-load-balance}?
|  |        ...
|  +--ro output
|     +--ro result        uint32
|     +--ro reason?       string
|     +--ro nexthop-id?   uint32
```

## 2.2 Software-Defined Networking

The term *Software-Defined Networking* (SDN) describes a set of methods. These methods aim to ease design, delivery and operation of network services. This is done in a dynamic and scalable way. The techniques also master components of the service delivery chain, so the methods are capable of providing services that satisfy contracts with customers. [9]

In this thesis the term "Software-Defined Networking" will refer to the concept of separating the process of making forwarding decisions from actual forwarding devices [10]. Figure 2.3 illustrates this concept. It depicts Application 1 inside the *Application Plane*. The Application Plane provides an interface for administrators to insert policies into the *Control Plane*. It also shows two controllers, Controller 1 and Controller 2, inside the Control Plane. These are responsible for making forwarding decisions. The forwarding devices, here the three switches Switch 1 to 3, are inside the *Forwarding Plane*, also referred to as *Data Plane*. Every forwarding device also requires a connection to a controller. These are omitted in Figure 2.3 to not obfuscate the network. Instead it illustrates an abstract communication between the Control Plane and the Forwarding Plane. In this case the protocol for information exchange between Control Plane and Forwarding Plane is OpenFlow [10]. Connections between Control Plane and Data Plane are referred to as *Southbound Connections*. The ones between Control Plane and Application Plane are called *Northbound Connections*.

To better demonstrate the concept we will look at an example. Figure 2.4 illustrates it. We are just concerned with the functionality of SDN. So we assume both Computer 1 and Computer 2 run a network stack comparable to the one of Linux. They also have already stored the layer 2 address of each other. The example starts with Computer 1 sending a frame to Computer 2. First it has to send the frame to Switch 1 because it has only this link to another device. Switch 1 has to forward the frame to the next device. But it is not capable of making forwarding decisions because the logic for decision making is located in the Control Plane. So Switch 1 uses a protocol like OpenFlow

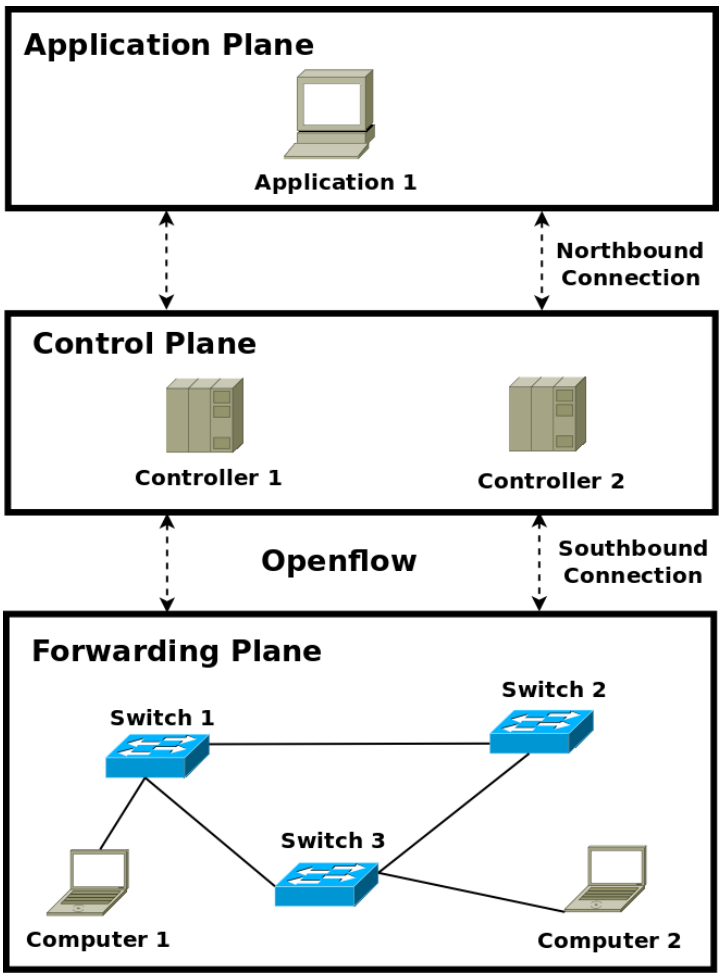


Figure 2.3: Depiction of Control Plane and Forwarding Plane in a software-defined network

to request a decision from its controller. In this example we will use Controller 1 as controller of all switches. Controller 2 is just used if Controller 1 is not reachable. So Controller 1 receives the request and has to decide the next destination of the frame. The decision is based on implemented policies. One policy can be a static configuration which states that all frames sent to Switch 1 from Computer 1 have to be sent to Switch 2. It is also possible for controllers to collect flow data and to direct the frame to a certain destination for load balance. But for this example we try to keep the policy simple. So Controller 1 collects topology data of the network and determines the destination of a frame based on the network graph. More specific the controller tries to use a minimum of links for communication. Controller 1 replies to Switch 1 and states the frame has to be sent to Switch 3 next. The switches do not know the topology of the network, so to be more accurate Controller 1 will tell Switch 1 to sent out the frame on the port which is connected to Switch 3. When Switch 3 receives the frame it will also request a forwarding decision from Controller 1. Our policy is to use a minimum of links and

Switch 3 is directly connected to Computer 2, so Controller 1 answers to send out the frame on the port which is connected to Computer 2. Now Computer 2 receives the frame what ends our example.

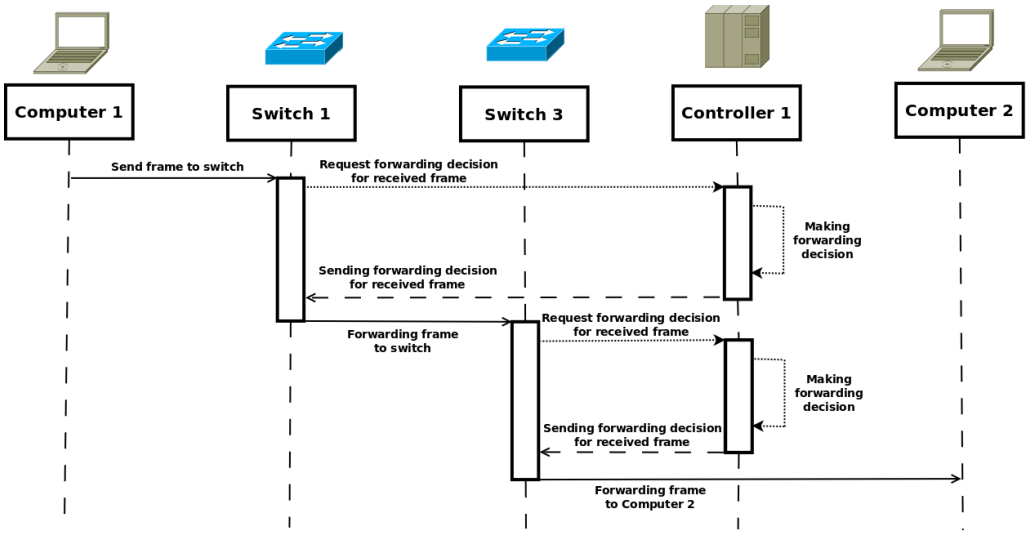


Figure 2.4: Sequence diagram of the example for SDN

## 2.3 OpenDaylight

OpenDaylight (ODL) is a modular open source platform for SDN [11]. It is based on a microservice architecture. This refers to the model-driven approach of ODL. Every module in ODL defines its data and service structure in YANG modules. This enables the use of a Model Driven Service Abstraction Layer (MD-SAL). Data structures can be stored and interacted with in one common MD-SAL store. The structure of installed services is well-defined in YANG modules. Therefore other services can easily use the API of another one. This enables ODL to create a chain of services. These services can then be bundled to ease deployment. ODL also allows users to just load services in the controller which are required for their use case. ODL's microservice architecture is especially useful because of the set of already implemented protocols and services. The OpenDaylight web page provides documentation for all official modules [12]. The most notable ones include support for RESTCONF, NETCONF, OpenFlow and BGP. It also uses the yangtools project [13] to auto-generate Java APIs from YANG modules.

The OpenDaylight controller supports a northbound implementation of both RESTCONF and NETCONF. A northbound implementation refers to a server capable of using the protocols. The RESTCONF server is listening on port 8181 and also provides an configuration interface. This allows both reading from and writing to the data store by using a browser. It also allows to call RPCs implemented and registered by services in

ODL. For NETCONF ODL runs two servers, one on port 1830 to access the configuration subsystem, another one on port 2830 to interact with the MD-SAL data store. The server which listens on port 2830 supports ssh and basic password authentication. The configuration files also allow to change these ports or enable an additional server which listens on port 2831. This additional server supports TCP. For interactions with other NETCONF enabled devices ODL supports a NETCONF southbound implementation. So ODL is also able to act as a NETCONF client. This feature is not part of the controller's odl-mdsal-all bundle. Instead it is supported in the odl-netconf-mdsal bundle which is not included in the controller.

OpenDaylight includes a L2 Switch project [14]. This uses ODL's OpenFlow library to control OpenFlow enabled switches. It provides support for flow tables. Based on these a user can enforce his policies in regards of forwarding. The DLUX project is useful to illustrate a network consisting of switches controlled by ODL.

The BGPCEP project [15] of ODL supports a southbound implementation of BGP and allows ODL to run a *BGP Route Reflector*. So it is capable of connecting to other BGP peers and to listen for incoming connections, but it is not able to forward packets. Instead a Route Reflector simply propagates routes to connected peers and helps keeping the RIBs homogeneously. BGPCEP supports an application-peer. An application-peer provides an interface which allows an administrator to insert routes into or delete routes from the RIB. All features like BGP peers, supported RIBs and application-peers can be configured.

## 2.4 Network Simulation

Network simulation allows to create virtual networks with specific conditions like ideal transmission rates or transmission errors. It can also be used to test software and protocols in networks, which are too expensive to be created with hardware. A variation of tools has been created to simulate networks. One network simulation tool is the Common Open Research Emulator (CORE) [16]. It provides a graphical user interface which allows users to create networks using drag and drop. The supported services can be configured for every element inside a network. These include routing protocols like OSPF or BGP and configurations for default gateways and static routing. Even if a service is not directly supported, CORE allows to run scripts for adding missing ones. CORE also allows to track traffic in real time with tools like tcpdump or wireshark.

Another tool for network simulation is Mininet [17]. It focuses on simulating SDNs with OpenFlow. To interact with the network during run time Mininet provides a command line interface. It has its own Python API for creating Mininet networks. The API allows users to create switches and hosts. All of them can be put into their own namespace to prevent problems when the same service is running on multiple simulated nodes. A user can also execute commands on nodes to create a start configuration. In regards of SDN,

Mininet allows users to specify the controller of every switch inside the network. The API does not support services of higher layers of the ISO-OSI model directly, but allows to run them inside a Linux machine. The only exception is setting of IP addresses.

## 2.5 Related Work

Even though I2RS is still under development it is already a large topic. This thesis only covers a part of a single service. So in order to put the work of this thesis in perspective, we first discuss other related parts of I2RS. One of these is the filter-based RIB. The proposal draft-ietf-i2rs-fb-rib-info-model-00 [18] defines its current version. Instead of destination-oriented routing like the RIB module defined in section 2.1.2, it provides routing decisions based on filters. These can match on headers of multiple layers. An administrator can enforce policies based on size, time or packet/frame count, too. It is possible that both a filter-based RIB and a RIB contain a matching route for a packet. To ensure determinism it is not possible to use both at the same time, but a destination-based RIB can be utilized as default RIB in cases where a filter-based RIB has no matches for a packet.

I2RS is based on a client-agent architecture. A client may need to keep track of changes in a network. Because other clients or routing protocols can trigger change events, RFC 7923 [19] defines requirements for a pub/sub service. It allows a client to subscribe to a set of YANG data objects in a data store and get notified when an event occurs. The YANG PUBSUB project of OpenDaylight [20] is one attempt to implement a service fulfilling the requirements of RFC 7923.

Another important part of I2RS are security requirements. This thesis focuses on a proof-of-concept implementation of some features of the I2RS RIB service. Therefore it does not make a detailed analysis of the security aspects of I2RS, but for deployment in a business network, it is necessary to keep risks low. This is considered by draft-ietf-i2rs-protocol-security-requirements-06 [5] that specifies what security requirements a deployable implementation of the I2RS protocol has to fulfill. These include, but are not limited to, a mutual authentication of client and agent, data integrity and data confidentiality.

In this thesis when we use the term *Software Defined Network* we primarily refer to the separation of Control Plane and Forwarding Plane in layer two of the ISO-OSI model. But as Susan Hares et al. [21] points out is the Interface to the Routing System also an approach of Software Defined Networking. Therefore protocols like OpenFlow of the Open Networking Foundation [22] are similar to I2RS. OpenFlow is a protocol responsible for the communication between a controller and OpenFlow enabled switches. So it takes care of the connection between the Control Plane and the Data Plane. Since it is specified to work with switches its purpose is to allow Software Defined Networking

on layer two. I2RS is also responsible for communication between control elements, here the I2RS clients, and elements which are responsible for forwarding, here the routing elements containing an I2RS agent. The difference in the approaches is that in a SDN with OpenFlow enables switches, these do not make any forwarding decision. Instead the devices request one from the controller. In I2RS a routing element is still able to make forwarding decisions by itself, but the policies and information base which is used for making forwarding decisions is influenced by clients.

# Chapter 3

# Methodology

This chapter illustrates the network in which we have tested our implementation of I2RS. Since this thesis' aim is a proof-of-concept implementation rather than a deployable one the network is a small one. Nonetheless, it is intended to reflect the paradigm of separating the control plane from the forwarding plane. So the main sub-network will implement this paradigm. First section 3.1 states the software used in our tests. Section 3.2 illustrates the overall topology of the test network. Then section 3.3 will focus on the control plane of the main sub-network while section 3.4 explains the forwarding plane in the setup. At last section 3.5 describes additional sub-networks required for testing.

## 3.1 Software

In order to enable reproducibility we will state the software used in the tests. The forwarding plane runs inside a virtual machine. To be more specific we use Mininet 2.2.1 on Ubuntu 14.04 LTS - 64 bit [23]. It is emulated by VirtualBox [24]. The used build is VirtualBox build 5.0.14_Ubuntu r105127. The virtual machine has installed Mininet 2.2.1 [25] for network virtualisation of a SDN. Quagga 0.99.22.4 [26] implements the BGP routers inside the networks.

The control plane is run on a host machine. This host machine runs the virtual machine for the data plane. The I2RS agent is a ODL module. We deployed it in the Beryllium-SR2 stable build of ODL [12]. For the tests ODL runs in the OpenJDK Runtime Environment (build 1.8.0_91-8u91-b14-3ubuntu1 16.04.1-b14).

## 3.2 Network Topology

The network topology is an augmentation of a network created by Edwin Cordeiro. It can be found on github [27]. We adapted it to better fit our requirements. So we removed OSPF, substituted some routers with switches and connected the network to our control plane.

In order to better understand under which conditions our network is running we will first focus on network virtualisation. Figure 3.1 illustrates this. The entire network is run on a single hardware machine. We will now use the term *Host machine* or *HM* to describe it. To keep testing conditions of the network constant the HM runs a virtual machine. From now on we will refer to the virtual machine as *VM*. Other methods like docker container are also possible but we decided to use a virtual machine because Mininet already provides one and performance is not important in this thesis. The VM runs Mininet to simulate all hosts, switches and routers in the network. Mininet is not depicted in Figure 3.1 because it includes the entire network like the VM. Except from the VM the HM also runs ODL including the I2RS agent module and the I2RS client. These components are not inside the virtual machine so we can have a separation between the control plane and the forwarding plane. For constant test conditions we rely on a stable build of ODL which runs inside the Java Virtual Environment and a Python virtual environment for the client.

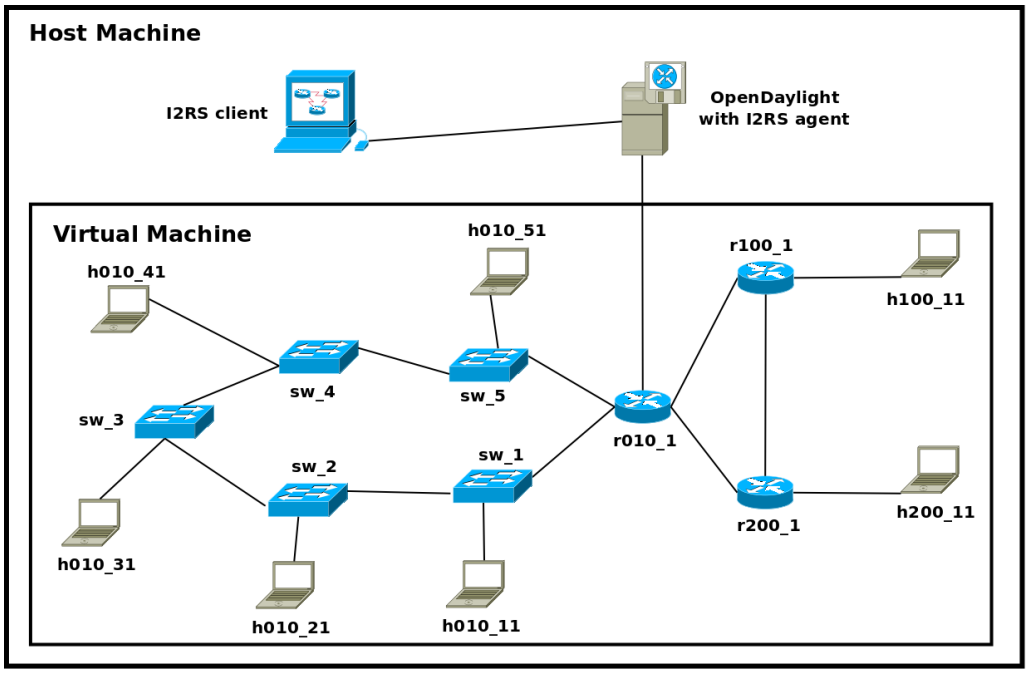


Figure 3.1: Illustration of virtualisation of the networks

Since I2RS is about access to the routing system the used protocols and interactions

between routers are important. This network only uses BGP as its routing protocol. Figure 3.2 depicts the different autonomous systems (AS) in the network. AS 10 contains a SDN consisting of five switches and the router r010_1 in a circle. It also includes the hosts h010_11 to h010_51. Router r010_1 is connected to ODL. ODL acts as a route reflector inside of AS 10. AS 100 consists of the router r100_1 and the host h100_1. The router r200_1 and the host h200_1 form AS 200. All three autonomous systems are connected with each other. This is achieved by linking r010_1, r100_1 and r200_1 in a circle. Interface eth2 of router r010_1 connects the network in the VM with ODL on the HM. This is achieved by running r010_1 in the root name space of the VM. Router r100_1 and router r200_1 run in their own name space to prevent problems with the BGP daemons. The VM and the HM communicate with each other by using a Host-Only adapter in VirtualBox.

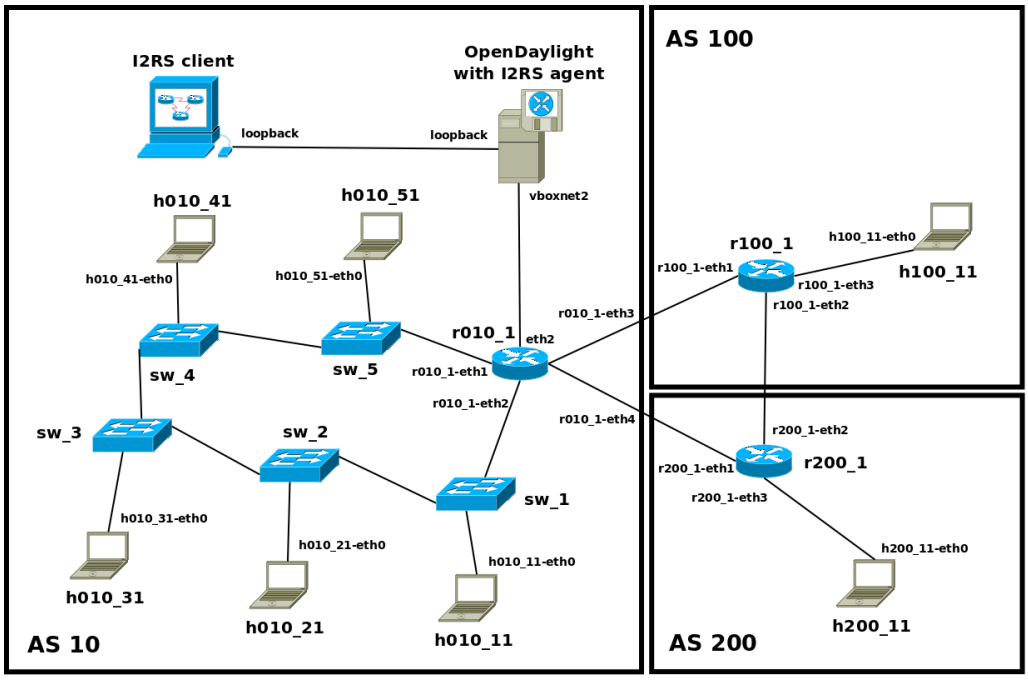


Figure 3.2: Illustration of the different Autonomous Systems inside the test network

Since this thesis focuses on I2RS in a SDN, we also separate AS 10 into the control plane and the forwarding plane. AS 100 and AS 200 are additional networks for testing purpose. Figure 3.3 illustrates this structure. Router r010_1 connects all components. So it is, to some extend, part of each one. The following sections explain these components in more detail.

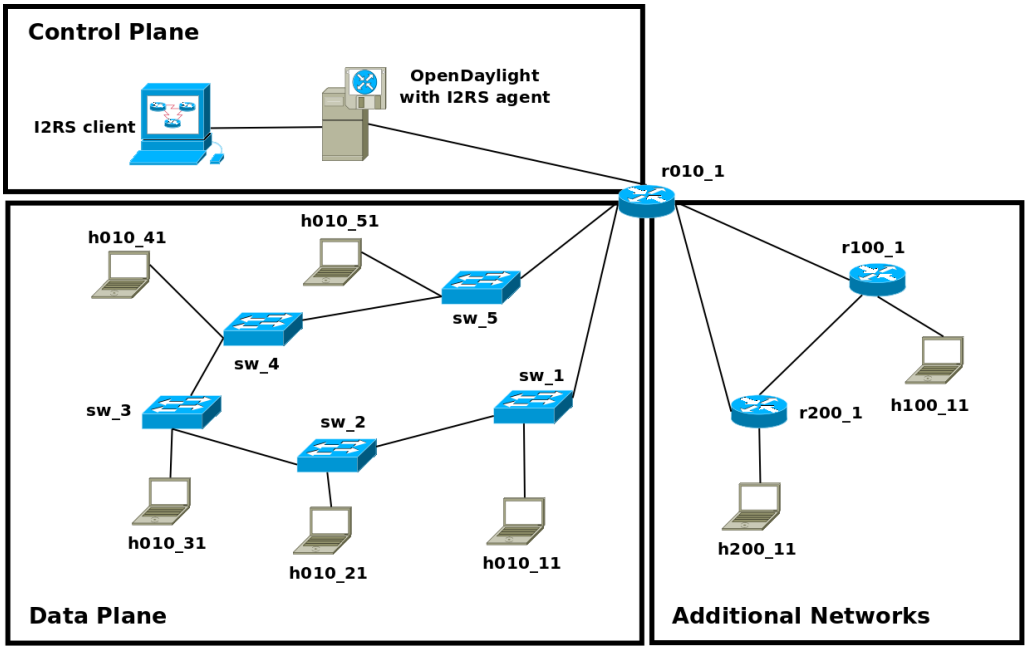


Figure 3.3: Depiction of the network separated into control and data plane

## 3.3 Control Plane

The control plane is responsible for making forwarding decisions. It includes the I2RS client, ODL and the Quagga router r010_1. The Quagga router is also responsible for forwarding. So the control plane and the data plane are not clearly separated. We did not investigate in finding a solution to this problem because the main focus of the thesis is I2RS.

### 3.3.1 I2RS client

The function of the client in the control plane is to allow administrators to access the routing system. This is achieved by interacting with the I2RS agent via the I2RS protocol. It provides a command line interface (CLI) for an user. The supported commands allow to connect to an agent on a specified address and port and to send RPCs to the agent. In the setup both client and agent are located on the same machine. So the loopback interface is used for communication. I2RS supports NETCONF and RESTCONF. The client in this example uses NETCONF. To add and delete routes via RESTCONF the API doc interface of ODL can be used.

### 3.3.2 OpenDaylight

ODL serves three functions in the control plane. The first one is as a controller for switches in the SDN. So in regards of layer 2 of the ISO-OSI model, it implements the entire control plane. For control on layer 3, it has to interact with the client and the Quagga router r010_1. The second function is handling interactions with clients. The I2RS agent module of ODL is responsible for this task. It listens for both RESTCONF and NETCONF clients to connect. Once a client has established a communication, the agent provides its services. For now they only include writing and deleting routes from a RIB. When a client calls a RPC, the agent changes the RIB accordingly if possible. The third function ODL provides is propagating route changes. ODLs BGP module acts as a BGP Route Reflector and sends route updates to its peers. Since ODL does not support any forwarding functionality this is necessary for I2RS to have effects on the network. Figure 3.3 shows that ODL has only one peer, the Quagga router r010_1.

### 3.3.3 Quagga

All routers in the network run Quagga's BGP daemon for routing. The router r010_1 in the control plane uses it to make forwarding decisions based on received route updates from its peers. These include the router r100_1 and r200_1 from the additional networks. Router r010_1 also communicates with ODL and implements routes inserted by I2RS. Quagga is a traditional router implementation. So it does not have a separation of the control plane and the forwarding plane. Instead the control logic resides inside the same machine as the kernel which is responsible for forwarding. Therefore it is also, to some extend, part of the data plane.

## 3.4 Forwarding Plane

The forwarding plane consists of five switches and partially the router r010_1. Figure 3.3 depicts this scenario. ODL controls the switches in the forwarding planes and makes forwarding decisions. Therefore the switches request forwarding decisions from ODL. In the network, these switches are responsible for the actual forwarding. The test setup includes the data plane so we have a separation between control logic and forwarding on layer 2 of the ISO-OSI model. I2RS is responsible for controlling the forwarding decisions on layer 3.

## 3.5 Additional Networks

Figure 3.3 illustrates the additional networks. When considering Figure 3.2 we can recognize that the additional networks are the autonomous systems AS 100 and AS 200. They are intended to emulate external networks so we can better check if added routes are actually applied and used. Both ODL and Quagga's BGP daemon state if prefixes are included in their RIB but sending ICMP echo requests and tracking traffic with wireshark or tcpdump is a more reliable source of information. Besides it is also advisable to test if routes are propagated beyond directly connected BGP peers.

# Chapter 4

# Implementation

This chapter explains the implementation of the I2RS route-add and route-delete RPCs. First section 4.1 lists the used software and explains decisions regarding their use. Section 4.2 illustrates the implementation of the I2RS agent. Section 4.3 provides an overview on the implementation of the I2RS client. This chapter ends with section 4.4 which describes some tests of the implementation.

## 4.1 Software

The I2RS agent is a module for ODL. The usage of ODL is suggested by the I2RS working group. Except from that ODL provides modules which provide important services like auto-generated YANG bindings or NETCONF support. We decided to use it for the agent and not the client because ODL also provides a module for BGP and can already be used as a controller of a SDN. Our implementation uses the ODL Beryllium-SR2 release. The target Java version of the implementation is OpenJDK version "1.8.0_91". The client uses ydk-gen [28] to generate a Python infrastructure. It allows to create Python modules based on YANG files, to connect to servers and includes a NETCONF Provider. The client is written for Python 2.7.12. Compatibility with Python 3 is not tested.

## 4.2 I2RS Agent

The I2RS agent is responsible for interacting with the RIB manager. Therefore it listens on the preconfigured port 2830. Clients can connect via ssh and use the NETCONF protocol for communication. It also allows to change this port and to enable an additional one for TCP connections. Besides it listens on port 8181 for RESTCONF clients. When a client has connected, it provides implementations for the route-add and the route-delete

RPC to modify the RIB. It does not provide a working implementation of other RPCs like route-update or nh-add.

### 4.2.1 Architecture

The architecture of the I2RS agent module is based on the Startup Project Archetype [29]. It consists of different subprojects. The most important ones are *api, impl, features* and *karaf*.

The api project includes the YANG data module for the RIB service. It uses ODL's yangtools project to generate a Java API from the YANG modules. Auto-generated classes can also be found inside the project. The exact method on how a Java API is generated from YANG modules is described in the yangtools documentation [30].

The subproject impl contains the implementations of the RPCs. When the module is installed into ODL, it registers the RPCs as services into the MD-SAL data store. This allows to use ODL's modules for RESTCONF and NETCONF servers. Because of this ODL's controller is also able to delegate the RPCs to the correct implementation.

In order to make the I2RS agent module deployable in ODL, the features subproject is required. It defines bundles which can be installed into ODL and is responsible for dependency management. Currently four features are bundled. The first one is *odl-i2rsagent-api*. It loads just the api project of the I2RS agent and OpenDaylight's MD-SAL models into ODL. The second one is *odl-i2rsagent*. This feature installs odl-i2rsagent-api, the impl project of the I2RS agent and the feature for BGP support. It also enables access to the MD-SAL data store and installs the NETCONF server. The third feature is *odl-i2rsagent-rest*. It includes the odl-i2rsagent feature and the one for RESTCONF support. The last one is *odl-i2rsagent-ui*. In addition to the odl-i2rsagent-rest feature it also installs ODL's MD-SAL API docs.

The karaf subproject contains a ODL version for testing of the module as a single feature. Therefore it just installs modules which are required according to features. It also loads the default configuration for modules when provided. In the case of I2RS this especially includes configurations for BGP peers, speakers and RIBs. In order to run the I2RS agent correctly the BGP configuration has to be adapted in this local instance of ODL.

### 4.2.2 Data structure

The I2RS agent does not yet implement the *routing instance* of the I2RS RIB service. So a separate RIB is not supported. Instead, it uses the *application peer* of the BGP module to interact with its RIB. It also does not yet support a configuration file. Because of these conditions, it is currently limited to only work with the BGP module, a correctly configured application peer, and predefined RIB names. Since the BGP module does not

support saving next hops separated from routes which include them, RPCs like nh-add can not be implemented. This structural limitation is also the reason why the route-add RPC processes next hops directly and does not rely on the nh-add RPC.

The data structure of input and output of the route-add RPC and the route-delete RPC consist of classes auto-generated from the YANG module. The implementation will try to map the input data to route information for the RIB of the BGP module. Since a route for example does not necessarily have an autonomous system number, this is not always possible. So it requires additional configuration data which is hard coded for now. Also the other way around the RIB of the BGP module does not support every information which can be included in the RPCs input data. One example for this is the route-index which is sent with every route. For now this data will be ignored and its function inside of I2RS is not supported.

### 4.2.3 Remote Procedure Calls

RPCs can be called via NETCONF and RESTCONF. ODL's controller checks if provided input is consistent with data defined in the according YANG module. Then it will delegate the input to the registered implementation of the service. It calls the method belonging to the RPC in the implementation and hands off the input formatted as Java classes. The Java classes are the ones auto-generated from the YANG module by ODL's yangtools module.

The I2RS RIB service implementation will then check if input data like the RIB name is valid and call a RPC handler to process the data. The handler will extract information from the input and map it to a route instance for the BGP RIB. If an error occurs because of an invalid value or a not yet supported type of route, the RPC handler will throw an exception containing the error code. Error codes are not specified in any document so they are arbitrarily ordered. The implementation does not yet have a feedback loop from BGP's RIB or its peers. So everything which could be successfully put into the MD-SAL data store will be regarded as a successful operation regardless of what was actually propagated to other peers. Once all routes are either added to the data store or have failed during processing, the output will be generated. It already supports the return-failure-details flag. This process is illustrated in Figure 4.1. How the BGP module interacts with the changes inside the MD-SAL data store and how it handles propagating routes to its peers is not depicted. This is part of the BGP module and is explained in its documentation [31].

Only route-add and route-delete RPC are currently supported. Both are also limited to IPv4 and IPv6 routes with a destination prefix as the match condition. Matching on source prefix, source and destination or other types of routes can not be processed and will throw an exception. Next hops are also limited to IPv4 and IPv6 addresses. Other base-types of next hops or more complex next hops like ones for next hop chaining or

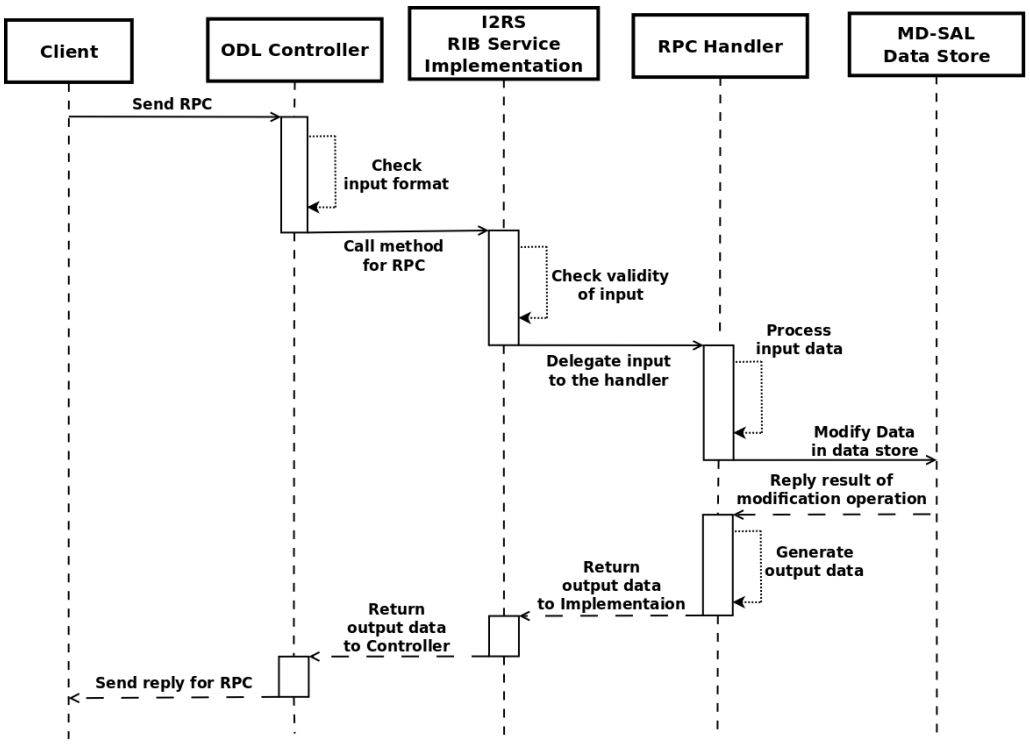


Figure 4.1: Sequence diagram on how a RPC is processed

load balance are not supported. If these are used the I2RS agent throws an exception. All exceptions based on not supported or invalid input are caught by the I2RS agent's RIB service implementation. It marks the corresponding route as failed and will adjust the output. This means it will increase the failed-count and create a failed-route if the return-failure-detail flag is set.

## 4.3 I2RS Client

The I2RS client is responsible for communicating with I2RS agents. It also provides a user interface to send RPCs to an agent.

### 4.3.1 Architecture

The I2RS client uses a Python module generated by ydk-gen as the base of the implementation. The module contains class representations of the data structure defined in the listed YANG files during the generation. In this case we created only modules for the RIB service data module and all of its dependencies. The generated modules also include Service Providers, which can be used to connect to servers via NETCONF, and

a CRUD service for reading and writing of data.

The client itself consists of different classes representing the client and its commands. So different implementations of commands are easily interchangeable. Except from this, the client does not have any additional internal structure because of its small size.

### 4.3.2 Functionality

This implementation of an I2RS client provides a command-line-like interface for users. So it is not possible to start it with parameters but once started a user can type in different commands and the client will request required information step by step. It also informs users if a value is optional or has a default value. In this case skipping the insertion of input for a field will lead to the client ignoring the field or the usage of default values.

At the moment the client supports just basic commands. These include *help, exit, connect, route-add, route-delete* and *test*. The help command will provide information about the client itself and supported commands. For leaving the client exit can be used. This will disconnect from all agents. The connect command specifies the parameters which should be used to connect to an agent. For now, the client is only capable of connecting to one agent at a time and supports only the NETCONF protocol. Thus session management is not provided. The commands route-add and route-delete call the RPCs on the agent. This requires prior use of the connect command. The test command can be used to run predefined tests on an agent. These basically call RPCs on the agent with predefined values. The tests include only functionality tests and no ones for evaluating performance.

## 4.4 Testing the setup

The tests consist only of integration tests. Unit tests inside an implementation of the agent or the client are not included. The tests, which are also defined in the client's test command, include several scenarios. The first one is adding a single IPv4 route with an IPv4 next hop. Second is adding multiple IPv4 routes with IPv4 next hops. The next ones each add a single IPv4 route with different invalid parameters. The tests are also repeated with IPv6 next hops and IPv6 routes, that are verified by reading from the RIB. The tests also include deleting the routes again and deleting routes, which do not exist. The test RPCs are first sent to the agent using a browser and ODL's RESTCONF API docs. After confirming functionality inside the agent the tests are repeated but this time they are sent from a NETCONF client. As independent NETCONF client we use yangcli-pro [32]. It supports connecting to clients via ssh, session management,

loading YANG files and using the RPCs of loaded YANG files. Last we repeat the tests by sending the RPCs from our I2RS client.

Listing 4.1 depicts an example input of the route-add RPC as XML file. This example sets the return-failure-detail to true. The target RIB has the name *example-app-rib*. It adds just one route with route-index 600. This route matches every packet with a destination address in the *110.3.0.0/24* network. The route attributes include a preference of seven. Vendor specific attributes are not defined. The next hop has the IPv4 address *10.0.10.2*. Listing 4.2 depicts the input for a route-delete RPC which deletes this route again.

Listing 4.1: Example of input data for a route-add RPC as XML file

```
<route-add xmlns="urn:ietf:params:xml:ns:yang:ietf-i2rs-rib">
    <return-failure-detail>true</return-failure-detail>
    <rib-name>example-app-rib</rib-name>
    <routes>
        <route-list>
            <route-index>600</route-index>
            <match>
                <ipv4>
                    <dest-ipv4-prefix>110.3.0.0/24</dest-ipv4-prefix>
                </ipv4>
            </match>
            <route-attributes>
                <route-preference>7</route-preference>
                <local-only>false</local-only>
                <address-family-route-attributes/>
            </route-attributes>
            <route-vendor-attributes/>
            <nexthop>
                <nexthop-id>637</nexthop-id>
                <sharing-flag>false</sharing-flag>
                <nexthop-base>
                    <ipv4-address>10.0.10.2</ipv4-address>
                </nexthop-base>
            </nexthop>
        </route-list>
    </routes>
</route-add>
```

Listing 4.2: Example of input data for a route-delete RPC as XML file

```
<route-delete xmlns="urn:ietf:params:xml:ns:yang:ietf-i2rs-rib">
    <return-failure-detail>true</return-failure-detail>
    <rib-name>example-app-rib</rib-name>
    <routes>
        <route-list>
            <route-index>600</route-index>
            <match>
                <ipv4>
                    <dest-ipv4-prefix>110.3.0.0/24</dest-ipv4-prefix>
                </ipv4>
            </match>
        </route-list>
    </routes>
</route-add>
```

Listing 4.3 illustrates how multiple routes can be added by calling one RPC. In order to achieve this, multiple instances of *route-list* are defined in the data structure. Each of these route-lists represents a route which should be added. Besides, this input data does not specify the return-failure-detail flag. Therefore the agent has to use its default value false.

Listing 4.3: Example of input data for adding multiple routes with the route-add RPC formatted as a XML file

```
<route-add xmlns="urn:ietf:params:xml:ns:yang:ietf-i2rs-rib">
    <rib-name>example-app-rib</rib-name>
    <routes>
        <route-list>
            <route-index>500</route-index>
            <match>
                <ipv4>
                    <dest-ipv4-prefix>10.2.0.0/24</dest-ipv4-prefix>
                </ipv4>
            </match>
            <route-attributes>
                <route-preference>42</route-preference>
                <local-only>false</local-only>
                <address-family-route-attributes/>
            </route-attributes>
            <route-vendor-attributes/>
            <nexthop>
                <nexthop-id>543</nexthop-id>
                <sharing-flag>true</sharing-flag>
                <nexthop-base>
                    <ipv4-address>10.10.0.6</ipv4-address>
                </nexthop-base>
            </nexthop>
        </route-list>
        <route-list>
            <route-index>5300</route-index>
            <match>
                <ipv4>
                    <dest-ipv4-prefix>10.13.0.0/24</dest-ipv4-prefix>
                </ipv4>
            </match>
            <route-attributes>
                <route-preference>17</route-preference>
                <local-only>false</local-only>
                <address-family-route-attributes/>
            </route-attributes>
            <route-vendor-attributes/>
            <nexthop>
                <nexthop-id>542</nexthop-id>
                <sharing-flag>true</sharing-flag>
                <nexthop-base>
                    <ipv4-address>10.10.0.2</ipv4-address>
                </nexthop-base>
            </nexthop>
        </route-list>
    </routes>
</route-add>
```

# Chapter 5

# Results

This chapter presents the results of the thesis. First section 5.1 states the functionality of I2RS. Section 5.2 demonstrates problems encountered while developing the I2RS agent and client.

## 5.1 Functionality of I2RS

The implementation of I2RS consists of a client and an agent. The current implementation of the client is capable of reading input from the command line. It creates a representation of the RPC as Python class object and inserts the given values. Because of time restrictions full recognition is not yet implemented into the client. If invalid input is given to a function it may lead the client to throw an exception and terminate afterwards. The client can create a route-add and a route-delete RPC. It can also connect to a NETCONF server on a specified port and supports the usage of ssh and basic password authentication. When sending a RPC to the server it throws an exception because of a timeout while waiting for the reply. The problem is on the client-side. Because of this bug all connectivity tests fail for the client.

The agent supports a northbound interface for both NETCONF and RESTCONF. If input data has an invalid structure ODL sends an error code to the client. This is handled by the ODL controller and not by our implementation of the I2RS agent. If an error occurs while processing the RPC, the agent marks the operation as failed. This is communicated to the client by using the failed-count and optionally the failure-details in the output of the RPC. With both protocols, NETCONF and RESTCONF, the agent was able to insert valid routes into the MD-SAL data store and to delete them afterwards. So all test cases were successful in regards of operating on the data store. The agent interacts with the BGP module in order to advertise routes to its peers. These routes include ones learned by other peers and ones inserted by the agent. The BGP module sends BGP update

messages containing the added routes. So far no invalid fields inside an update message could be found. In previous versions of the I2RS agent the BGP module has sometimes thrown warnings because of a lack of attributes in the routes. This problem has been resolved in the current version. Quagga's BGP daemon marks routes sent by ODL as valid and as best route, but crashes immediately after trying to install the route into the RIB. So for now, it is not possible to insert routes into the routing system.

## 5.2 Encountered Problems

During development we encountered two problems with the I2RS protocol. The first one is a different *route-index* size. Proposal draft-ietf-i2rs-rib-data-model-05 defines the route-add RPC. It contains a list of routes. A route is represented by a route-prefix which includes a route-index to be uniquely identifiable. The route-index of a route-prefix is of the type uint64. The output of the route-add RPC is a route-operation-state. It may have a container *failure-detail*. This container is supposed to give more information about failed routes. So it includes a list of failed-routes. A failed-route only contains two elements. The first one is an error code, which specifies the failure reason. The second one is a route-index. The route-index is used to match failed-routes to the routes which should be added. In this specification the problem is that the route-index of a failed-route is of the type uint32 and not uint64 like the route-index of a regular route. So we have a mismatch of types. This makes representation as failed-route impossible for every route which is added.

The second problem is a missing data structure for next hops. The route-add RPC specifies that the nh-add RPC has to be called first. The nh-add RPC is supposed to add a next hop to a RIB and return its hexthop-id. Then the route-add RPC uses the nexthop-ids for the next hops of the routes which should be added.

A normal workflow for adding a route in I2RS is supposed to look like this: A client has some routes it wants to add. So first it sends a nh-add RPC to an agent. It includes the next hops of the routes, the client wants to add to a RIB. The agent looks if the next hops already exist. If the next hops do not exist, the agent will create them and assign an unique nexthop-id to every single one. Next the agent sends a reply to the client. This reply contains the nexthop-ids of the requested next hops. The client can now use these nexthop-ids to substitute the next hop inside the routes. Next the client sends a route-add RPC to the agent. It contains the routes, the client wants to add and every route contains a nexthop-id representing a next hop. The agent receives this RPC and can add the routes to the RIB. The next hops can be resolved by matching the nexthop-id of a next hop in a route and one in the next hop storage of the RIB.

The problem with this process is that neither a routing instance nor a RIB contains any storage for next hops. An additional storage which can be mapped to a RIB is not defined,

too. This makes it impossible for an agent to keep track of already used nexthop-ids or to resolve a next hop based on its id. Since this is required by the specification of the nh-add RPC and the route-add RPC, these can not be implemented correctly.

# Chapter 6

# Discussion

This chapter discusses the results of the thesis. First section 6.1 explains the state of the implementation of I2RS presented in this thesis. Next section 6.2 discusses the I2RS protocol. Section 6.3 states the future work.

## 6.1 Discussion of the implementation

The first part of the implementation is the I2RS client. Currently, when the client connects to an agent and sends a RPC it times out while waiting for the reply from the agent. We used other NETCONF clients like yangcli-pro to send RPCs to the agent. In these cases the agent replied immediately. Therefore we can be sure it is a bug in the client, not in the agent, but the reason for this bug is not known. It requires further investigation and debugging of the client. Nevertheless, this bug leads to the fact that the client is not working. Therefore it is a severe one.

For now the client only has to be able to send RPCs via NETCONF like every other NETCONF client. So currently it is possible to substitute the I2RS client by a NETCONF client like yangcli-pro. Later the client is supposed to combine the route-add RPC with the nh-add RPC. This can not be done easily by a simple NETCONF client. Therefore, in our opinion, it is still advisable to create a running I2RS client for implementing future features.

Our implementation of the I2RS agent can receive route-add and route-delete RPCs via NETCONF and RESTCONF. It is also capable of modifying the MD-SAL data store of OpenDaylight. The agent can interact with the BGP module of OpenDaylight and insert routes into its RIB. So this part of the implementation is working as supposed. The BGP module acts as a Route Reflector and therefore advertises the inserted routes to its peers. In the test networks we used Quagga for its peers. Quagga crashed when it tried to insert a route into its RIB. The exact reason for this problem is not known. In earlier

stages of development our setup used CORE for network simulation. In this setup we also observed a similar misbehavior of Quagga. We were able to link it to the fact that our virtual machine does not support IPv6 but it tries to install IPv6 routes into the RIB. In our current setup Quagga also runs inside a virtual machine which does not support IPv6. Because of this correlation we assume that the misbehavior of Quagga is due to a lack of IPv6 support. But at the moment a bug in the implementation or a wrong configuration of our agent can not be excluded. So it requires further investigation to solve this problem.

## 6.2 Discussion of the Interface to the Routing System

This thesis also introduced the basics of I2RS. Even though the implementation of the Interface to the Routing System does not yet work correctly, this thesis has shown that I2RS is on a good way. It can be used to interact with the routing system and allows automation of tasks. This is achieved because of the YANG data modules and already established protocols like NETCONF. In the thesis for example we were able to substitute our not correctly working client with an already existing NETCONF client. But it is also notable that there is still a lot of work to do. This includes fixing our implementations of both, the I2RS agent and the I2RS client, and an implementation of further specification of the I2RS protocol.

In this thesis we found two errors in draft-ietf-i2rs-rib-data-model-05. The first one is a type mismatch in the route-index of a regular route and a failed route. This problem makes sending always correct output impossible because not every route can be mapped on a failed route. This error can be solved by changing the type of a field and does not require any structural change. Therefore we consider it to be a minor one, but draft-ietf-i2rs-rib-data-model-06 [33] has not yet addressed this problem. Our suggestion is to change the types of both, the route index of a regular route and the one of a failed route, to uint64. This solves the problem and allows a huge number of different routes to be installed.

The second issue is about a missing data structure for next hops in the routing-instance and in the RIBs. It makes storing next hops impossible. Because of this problem it is not possible to implement the route-add RPC or the nh-add RPC according to the specification. Therefore we consider it to be a severe problem. In draft-ietf-i2rs-rib-data-model-06 this has been addressed and is solved by adding a nexthop-list to every RIB. The nexthop-list allows to reference next hops based on the their nexthop-id and therefore enables an agent to manage next hops. This looks like a good approach to us, but further evaluation is required in order to determine if it solves all related problems and does not generate new ones.

## 6.3 Future Work

I2RS includes many services and requirements. Therefore a lot of work has to be done. This thesis works with draft-ietf-i2rs-rib-data-model-05, but because of our findings this proposal has been updated by the IETF working group. So the first step should be to update to draft-ietf-i2rs-rib-data-model-06. An evaluation of this new specification is necessary, especially in regards of the data structure for next hops. The next step should be solving the problem with the routers. Because if other routers are not implementing routes advertised by the agent, the agent can not be used as supposed. Therefore a setup which supports IPv6 can be used in order to test whether the lack of IPv6 support causes the problem. If this does not solve the issue Quagga's logs and the sent BGP update messages should be reexamined to find more clues about this bug.

Afterwards the implementation of a routing-instance would be advisable. It allows to create a separate RIB which enables the agent to support other protocols than BGP. Also it makes it possible to manage next hops as supposed by I2RS. Therefore nh-add, nh-delete, rib-add and rib-delete RPC should be implemented. Also the route-add RPC has to be adapted to work with the nh-add RPC and the routing-instance. Other features of the RIB Service of I2RS are the route-update RPC and notifications for events. These have to be implemented over time, too. Besides the implementation of a configuration system is necessary. This allows to work with more than one specified RIB of the BGP module.

Except from that I2RS defines other services like filter-based RIBs. An implementation of these is also part of future work. Before deployment the security requirements and a traceability frame work need to be implemented. So only for I2RS a lot of work has to be done. But creating a working implementation of an I2RS client is also an important part. Tools like yangcli-pro work for I2RS at the moment but when I2RS implements features beyond NETCONF connections and calling RPCs, usage of these tools might not be possible any more. Some of these features include multiple RPCs working together like the nh-add RPC and the route-add RPC or even adding features to NETCONF [34]. An automation of this interaction in a client is advisable. So future implementations of I2RS clients can be based on our client, implement a new client by using adapted libraries for NETCONF or even be based on already existing tools like yangcli-pro.

# Chapter 7

# Conclusion

In this thesis we presented a proof-of-concept implementation of an I2RS client and an I2RS agent. These modules support the route-add and route-delete RPCs of draft-ietf-i2rs-rib-data-model-05.

Our I2RS client can read input data from a command line interface and is capable of creating Python class representations of RPCs. It can also connect to an agent on specified ports. Currently the client gets a time out when it is sending a RPC to an agent. Therefore it is not working correctly and further investigation is required.

Our I2RS agent can receive and process the route-add RPC and the route-delete RPC. In our test setup, when the agent propagates inserted routes to Quagga, its BGP daemon crashes. The reason for this bug is currently not known. We assume that it is linked to a lack of IPv6 support in our test environment. This assumption is based on experiences with similar problems in previous test setups. Nevertheless this bug requires further investigation to achieve a working implementation.

The development lead to the discovery of two errors in the specification which form our main contribution to the development of I2RS. The first one is a type mismatch which makes creating an always correct reply to the route-add RPC and the route-delete RPC impossible. A more severe one is the missing of a data structure for storing next hops. This causes that the nh-add RPC is not usable and the route-add RPC can not be implemented according to its specification. The second error is solved in draft-ietf-i2rs-rib-data-model-06, but it is required that the functionality of this specification is evaluated, too.

Because of these results, we conclude that I2RS still requires more development before it can be used in a business network. The specification requires more evaluations and a working implementations is necessary. But in our opinion I2RS is on a good way to become a practical standardized interface.

# Bibliography

[1] L. Wang, H. Ananthakrishnan, M. Chen, amit.dass@ericsson.com, S. Kini, and N. Bahadur, "A YANG Data Model for Routing Information Base (RIB)," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-i2rs-rib-data-model-05, March 2016. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-i2rs-rib-data-model-05.txt

[2] A. Atlas, T. Nadeau, and D. Ward, "Problem Statement for the Interface to the Routing System," Internet Requests for Comments, RFC Editor, RFC 7920, June 2016.

[3] A. Atlas, J. Halpern, S. Hares, D. Ward, and T. Nadeau, "An Architecture for the Interface to the Routing System," Internet Requests for Comments, RFC Editor, RFC 7921, June 2016.

[4] J. Haas and S. Hares, "I2RS Ephemeral State Requirements," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-i2rs-ephemeral-state-15, July 2016. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-i2rs-ephemeral-state-15.txt

[5] S. Hares, D. Migault, and J. M. Halpern, "I2RS Security Related Requirements," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-i2rs-protocol-security-requirements-06, May 2016. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-i2rs-protocol-security-requirements-06.txt

[6] J. Clarke, G. Salgueiro, and C. Pignataro, "Interface to the Routing System (I2RS) Traceability: Framework and Information Model," Internet Requests for Comments, RFC Editor, RFC 7922, June 2016.

[7] N. Bahadur, S. Kini, and J. Medved, "Routing Information Base Info Model," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-i2rs-rib-info-model-08, October 2015. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-i2rs-rib-info-model-08.txt

[8] M. Bjorklund, “YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF),” Internet Requests for Comments, RFC Editor, RFC 6020, October 2010. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6020.txt

[9] M. Boucadair and C. Jacquenet, “Software-Defined Networking: A Perspective from within a Service Provider Environment,” Internet Requests for Comments, RFC Editor, RFC 7149, March 2014. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7149.txt

[10] Open Networking Foundation, “Software-Defined Networking: The New Norm for Networks,” Open Networking Foundation, Palo Alto, CA, USA, White paper, Apr. 2012. [Online]. Available: http://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[11] “Platform Overview | OpenDaylight,” https://www.opendaylight.org/platform-overview/, (Accessed on 07/31/2016).

[12] “Downloads | OpenDaylight,” https://www.opendaylight.org/downloads, (Accessed on 07/31/2016).

[13] “YANG Tools:Main - OpenDaylight Project,” https://wiki.opendaylight.org/view/YANG_Tools:Main, (Accessed on 08/11/2016).

[14] “L2 Switch:Main - OpenDaylight Project,” https://wiki.opendaylight.org/view/L2_Switch:Main, (Accessed on 07/31/2016).

[15] “BGP LS PCEP:Main - OpenDaylight Project,” https://wiki.opendaylight.org/view/BGP_LS_PCEP:Main, (Accessed on 07/31/2016).

[16] “Common Open Research Emulator (CORE) | Networks and Communication Systems Branch,” http://www.nrl.navy.mil/itd/ncs/products/core, (Accessed on 08/08/2016).

[17] “Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet,” http://mininet.org/, (Accessed on 08/08/2016).

[18] S. Kini, S. Hares, L. Dunbar, A. Ghanwani, R. R. Krishnan, D. Bogdanovic, and R. White, “Filter-Based RIB Information Model,” Working Draft, IETF Secretariat, Internet-Draft draft-ietf-i2rs-fb-rib-info-model-00, June 2016. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-i2rs-fb-rib-info-model-00.txt

[19] E. Voit, A. Clemm, and A. G. Prieto, “Requirements for Subscription to YANG Datastores,” Internet Requests for Comments, RFC Editor, RFC 7923, June 2016.

[20] “YANG PUBSUB:Main - OpenDaylight Project,” https://wiki.opendaylight.org/view/YANG_PUBSUB:Main, (Accessed on 07/31/2016).

[21] S. Hares and R. White, "Software-Defined Networks and the Interface to the Routing System (I2RS)," *IEEE Internet Computing*, vol. 17, no. 4, pp. 84–88, Jul. 2013. [Online]. Available: http://dx.doi.org/10.1109/MIC.2013.76

[22] "OpenFlow - Open Networking Foundation," https://www.opennetworking.org/sdn-resources/openflow/57-sdn-resources/onf-specifications/openflow?layout=blog, (Accessed on 08/11/2016).

[23] "Mininet VM Images · mininet/mininet Wiki · GitHub," https://github.com/mininet/mininet/wiki/Mininet-VM-Images, (Accessed on 08/01/2016).

[24] "Downloads – Oracle VM VirtualBox," https://www.virtualbox.org/wiki/Downloads, (Accessed on 08/01/2016).

[25] "Download/Get Started with Mininet - Mininet," http://mininet.org/download/, (Accessed on 08/01/2016).

[26] "Quagga Software Routing Suite," http://www.nongnu.org/quagga/, (Accessed on 08/01/2016).

[27] "GitHub - edwinsc/mininet_ospf_bgp: A simple Mininet network running Quagga (OSPF and BGP)," https://github.com/edwinsc/mininet_ospf_bgp, (Accessed on 08/01/2016).

[28] "GitHub - CiscoDevNet/ydk-gen: Extensions to pyang for generating code from yang models." https://github.com/CiscoDevNet/ydk-gen, (Accessed on 08/03/2016).

[29] "OpenDaylight Controller:MD-SAL:Startup Project Archetype - OpenDaylight Project," https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Startup_Project_Archetype, (Accessed on 08/03/2016).

[30] "YANG Tools:YANG to Java Mapping - OpenDaylight Project," https://wiki.opendaylight.org/view/Yang_Tools:Code_Generation_Demo:YANG2JAVA_Mapping, (Accessed on 08/03/2016).

[31] "BGP LS PCEP:Beryllium Developer Guide - OpenDaylight Project," https://wiki.opendaylight.org/view/BGP_LS_PCEP:Beryllium_Developer_Guide#BGP_Developer_Guide, (Accessed on 08/10/2016).

[32] "yangcli-pro - YumaWorks," https://www.yumaworks.com/yangcli-pro/, (Accessed on 08/10/2016).

[33] L. Wang, H. Ananthakrishnan, M. Chen, amit.dass@ericsson.com, S. Kini, and N. Bahadur, "A YANG Data Model for Routing Information Base (RIB)," Internet Engineering Task Force, Internet-Draft draft-ietf-i2rs-rib-data-model-06, Jul. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-i2rs-rib-data-model-06

[34] S. Hares and amit.dass@ericsson.com, “I2RS protocol strawman,” Internet Engineering Task Force, Internet-Draft draft-hares-i2rs-protocol-strawman-03, Jul. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-hares-i2rs-protocol-strawman-03