BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Improving Link Virtualization in the CloudNets Network Virtualization Framework

Generalized Link Metadata and Transit Link Negotiation

vorgelegt von

Johannes Graßler

Matrikelnummer: 78 33 71

dem Fachbereich VI – Informatik und Medien –
der Beuth Hochschule für Technik Berlin vorgelegte Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science
im Studiengang
Medieninformatik

Tag der Abgabe: 28. August 2014

Betreuer
Dr. M. Steppat Beuth Hochschule für Technik
Dr. S. Schmid Technische Universität Berlin

Gutachter
Prof. Dr. M. von Löwis Beuth Hochschule für Technik

Kurzfassung

In dieser Arbeit beschreibe und implementiere ich das Transit Link Negotiation Protocol, ein Protokoll zum Aushandeln von Konfigurationsparametern für virtualisierte Netzwerkverbindungen. Dieses Protokoll ist für die Nutzung durch Netzwerkbetriebssysteme, wie die an der TU Berlin entwickelte Netzwerkvirtualisierungsarchitektur CloudNets vorgesehen. Es verbessert die Behandlung von *Transit-Links*, Verbindungen virtualisierter Netzwerke über Providergrenzen hinweg.
Vor meiner Arbeit wurden die Konfigurationsparameter von Transit-Links nicht durch die umsetzenden PIPs (Physical Infrastructure Provider) festgelegt, sondern durch den Betreiber eines in mehrere Teilnetze segmentierten virtualisierten Netzwerks. Diese Teilnetze werden auf mehrere PIPs verteilt. Dies ist problematisch, da beim Vorhandensein mehrerer Netzbetreiber der Namensraum verfügbarer Konfigurationsparameter für Transit-Links statisch zwischen diesen aufgeteilt werden muss, um Kollisionen zu vermeiden. Eine Lösung für dieses Problem ist die Delegation der Konfigurationsparametervergabe an die PIPs, die Transit-Links umsetzen. Das Transit Link Negotiation Protocol setzt diese Lösung um.
Diese Arbeit ist folgendermaßen aufgebaut: In Kapitel 1 gebe ich einen Überblick zum Stand der Forschung im Bereich der Netzwerkvirtualisierung und gehe dabei insbesondere auf die CloudNets-Architektur und die Problematik der Transit-Links ein. Kapitel 2 gibt einen Überblick über verschiedene Linkvirtualisierungsmechanismen und beschreibt für jeden dieser Mechanismen die Konfigurationsparameter, die einen damit realisierten Transit-Link ausmachen. In Kapitel 3 arbeite ich die Anforderungen an ein auf PIP-Ebene funktionierendes Aushandlungsprotokoll für Konfigurationsparameter heraus, und skizziere eine Protokollarchitektur, die diesen Anforderungen gerecht wird. In Kapitel 4 beschreibe ich das Transit Link Negotiation Protocol, meine Umsetzung dieser Protokollarchitektur und zeige einen möglichen Weg zur Integration in das CloudNets-Netzwerkbetriebssystem.

Abstract

In this thesis I propose and implement the Transit Link Negotiation Protocol, a link parameter negotiation protocol for use by the CloudNets network virtualization architecture devoleped at TU Berlin, as well as other network operating systems. My link negotiation protocol improves the handling of *transit links* links that interconnect virtual networks implementend by multiple providers across these providers' boundaries.
Currently, transit links' parameters are dictated to the Physical Infrastructure Providers (PIPs) implementing them by the virtual networks operator, who specifies the virtual network and divides it into partial networks interconnected by transit links. This situation is not ideal since it requires exclusively assigning a block of the available link parameter name space to each network operator. This potentially wastes a large portion of the available name space. One solution for this issue is devolving the authority to assign configuration parameters to the PIPs implementing transit links. The Transit Link Negotiation Protocol implements this solution.
This work is organized as follows: in Chapter 1 I sum up the state of the art in the field of network virtualization with a focus on the CloudNets architecture and the transit link problem complex. Chapter 2 gives an overview of various link virtualization technologies and discusses the parameters that define a virtual link for each of these technologies. Chapter 3 discusses the requirements and challenges of a link parameter negotiation protocol and outlines a high-level protocol architecture fulfilling these requirements. In Chapter 4 I describe the Transit Link Negotiation Protocol, my implementation of this architecture and provide a roadmap for integration into the CloudNets network operating system.

Erklärung

Ich versichere, dass ich diese Abschlussarbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Datum Johannes Graßler

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank the following people for their help in bringing this thesis about:
My advisors, Michael Steppat and Stefan Schmid who firmly but gently steered me towards a well-rounded end product. Anja Feldmann, who gave me the opportunity to work on the CloudNets prototype and turn it into what is now `cloudnets-framework`, and who encouraged me to resume studying computer science. Gregor Schaffrath, with whom I had the pleasure of collaborating on the CloudNets prototype and who asked hard questions about various aspects of my protocol design. Srivatsan Ravi, who introduced me to the finer points of distributed consensus. Tom Eichhorn, who is a fountain of wisdom on all things MPLS, and the tf-platform team with whom I have been exploring the OpenStack cloud platform for the past months, at Syseleven GmbH. Professor Sven Tschirley who published the Beuth Hochschule themed LaTeX template I used for this thesis.
Last but not least I would like to thank my parents who had the patience to support me despite my not always linear progress towards graduation and Olga, my wife, who put up with my absentmindedness, irritability and sloppiness over the past months and made this thesis a far more pleasant read than it used to be through numerous proofreading passes.

Chapter 1

Introduction

Network virtualization has been a topic of interest ever since Sincoskie and Cotton laid the groundwork[58] for what eventually evolved into the technology we today know as IEEE802.1Q or *VLANs*[1]. IEEE 802.1Q is a network virtualization protocol based on adding the numerical *VLAN tag* field to an Ethernet[36] frame's header that identifies the logical network it belongs to. Using VLANs multiple logical networks can share the same physical network.

Since the introduction of VLAN tagging a range of network virtualization technologies have appeared on the stage. These technologies are in turn leveraged by *network operating systems* or *cloud operating systems*[2]. So far deployments of these network operating systems exist in isolation, insofar as there is no standard way of interconnecting virtual networks hosted by different cloud providers, apart from connecting both to the public Internet and using it as a communication channel. It would be desirable to have a standardized way of establishing *transit links* between networks that have similar characteristics to these networks' internal links, e.g. bandwith or latency guarantees. First, this requires a reliable interconnection between cloud providers, of course. Second, it requires a way to distinguish logical links sharing this interconnection. This in turns requires a mechanism to reliably synchronize the configuration parameters identifying a logical links, such as VLAN tags, between all involved parties. This synchronization is the problem I am going to solve in this work.

I will use the example of the CloudNets network virtualization architecture[53] to illustrate the problem and engineer a solution. It is a particularly good example, since its model of federated roles already takes the challenges of establishing virtual networks across provider boundaries into account, even though its prototype implementation `cloudnets-framework` [62] currently lacks the mechanisms to scalably overcome these challenges. I will analyze the requirements of such a mechanism and develop the software components required to implement it.

In this introduction, I will first give a broad overview of the network virtualization field in general and the CloudNets architecture in particular. Finally, I will discuss the scalability and interoperability problems the current implementation of link virtualization in `cloudnets-framework` exhibits to motivate my work.

---

[1] Virtual Local Area Networks[38]

[2] Both are somewhat ambiguous terms: usually they refer to management and orchestration systems that abstract the management of servers and their interconnecting networks to varying degrees. This abstraction ranges from simple virtual or physical server provisioning and VLAN assignment to translating service specifications into arbitrarily complex deployments of interrelated servers including automatically configuring the services running on them.

1.1 Terms and Definitions

I shall use the following terms as defined here throughout this work.

Global Link Identifier This is a globally unique identifier assigned to each *Transit Link* by the entity requesting this link.

Infrastructure Provider, Physical Infrastructure Provider, PIP I use these terms synonymously to refer to any entity that operates physical or virtualized infrastructure capable of hosting virtual networks.

Logical Link, Virtual Link, VLink I use these terms synonymously to refer to any logical links that interconnect a virtualized network's hosts.

Link Domain Two or more infrastructure providers may peer with each either directly using dedicated connections, Internet exchange points or even public networks such as the Internet. I treat all these peering methods equally by referring to them as *link domains*: a link domain is any venue that may be used to establish logical links crossing provider boundaries and that provides a single shared name space, such as VLAN tags, for identifying these logical links.

Local Link Identifier I use this as a shorthand to refer to the set of attributes (such as VLAN tags, MPLS labels, tunnel endpoints) required to identify a logical link in the scope of a link domain. This only includes data that need to be configured prior to establishing the link - dynamically created parameters, such as those automatically negotiated by the parties to the link using routing protocols or similiar mechanisms, are not part of the local link identifier.

Network Provider, Virtual Network Provider, VNP I use these terms synonymously to refer to any entity that uses the resources of one or multiple *Physical Infrastructure Providers* to assemble virtual networks.

Transit Link This refers to a *Logical Link* that extends across two or more Physical Infrastructure Providers' networks.

Virtual Node, VNode, Virtual Host, VM I use these terms synonymously to refer to any virtual machine that is part of a virtualized network.

1.2 Background and State of the Art

In this section I will give an overview of the history of network virtualization in research and industry, putting my work into context.
While link virtualization technologies allowing for multiple logical networks on a shared physical infrastructure have been in existence for over 20 years[58], whole network virtualization is still comparatively young. Two of its early milestones in academia are the GENI[23] (2006) and OFELIA[61] (2010) projects. These projects had the goal of building testbeds for networking research that would allow multiple researchers to specify arbitrary experimental network topologies to be implemented on a shared physical network. They were one of the first formal attempts at virtual network orchestration and introduced the notion of a single specification describing the properties of both a virtual network's

nodes and their interconnecting network links. On the heels of these projects followed the CloudNets project[53]. Its initial goal was the development of a network virtualization architecture as an enabler for innovation in networking by allowing new networks to co-exist with existing ones on the same infrastructure. Among other results the project yielded the virtual network resource description language FleRD[52] and the prototype implementation of the CloudNets Network Virtualization architecture[62], which I will cover in detail in the next section. Finally Ericsson's *Network Embedded Cloud*[3] is the first effort at tackling the wide-area virtualization problem.
In industry, Google is the most prominent company to implement network virtualization: it introduced a software-defined network architecture called B4, which began operating in early 2010 and is used to serve production traffic. Urs Hoelzle, Google's Senior Vice President for Technical, Infrastructure publicly announced[28] the existence of B4 at the Open Networking Summit in 2012[3]. Aside from Google, there have been various efforts at establishing industry standards for network virtualization. The most important of these is probably the OpenStack[44] cloud management platform's Neutron[41] service, which offers virtualized internal networking for OpenStack based clouds.
Most of the architectures and technologies mentioned so far share a common blind spot: they do not incorporate wide-area network virtualization across two or more cloud provider's boundaries. The notable exceptions are the CloudNets project's federated hierarchy of roles and Ericsson's Network Embedded Cloud. The former looks at the problem from an architectural perspective and identifies business roles that might have an interest in creating virtualized networks that span multiple cloud providers. The latter is primarily concerned with identifying enabling technologies such as interfaces for controlling components implementing wide-area link virtualization (e.g. switches or routers).
Finally, in late 2013 another initiative with an interest in virtual network orchestration was started: The European Union FP7[9] project UNIFY[13]. UNIFY is a consortium of industry (Deutsche Telekom, Ericsson, Intel and Telecom Italia among others) and research (Budapest University of Technology and Economics, Politecnico di Torino, Technische Universität Berlin, Universidad del Pais Vasco) partners, aiming to create a unified production environment incorporating both datacenter and carrier networks. The primary focus of this unified architecture is services' needs: In a UNIFY environment, a service provider would specify its services' needs in a formalized way, submit this description to the production environment's operators and get a fully provisioned virtual network fullfilling these needs in return.

1.3 CloudNets Network Virtualization Architecture

The CloudNet architecture's origins lie with a cooperation project of Technische Universität Berlin, Deutsche Telekom Laboratories, University of Karlsruhe, and Lancaster University, among others[53]. This project proposed to create a network virtualization ecosystem of federated business roles (See table 1.1, and fig. 1.1).
The vision behind this ecosystem of roles was partially realized in the CloudNets network virtualization prototype, `cloudnets-framework` [62]. In the following, I would like to introduce some of the central concepts and mechanisms of the CloudNets architecture, along with some key implementation details of `cloudnets-framework` .

[3]Since Urs Hoelzle's original presentation slides are fairly terse, the reader may also refer to the SIGCOMM paper on the architecture published in 2013[29].

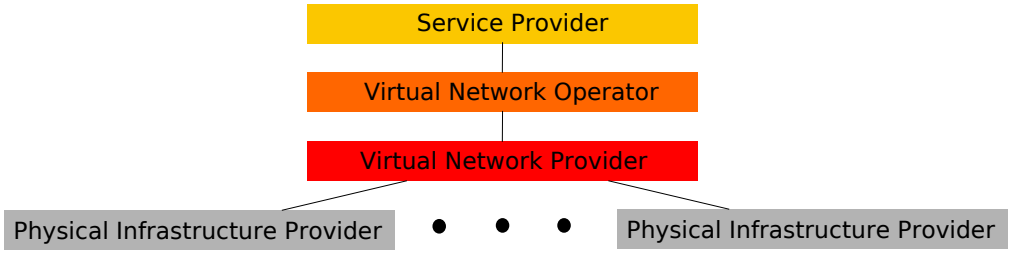

Figure 1.1: Hierarchy tree of the CloudNets architecture's roles (from [53]).

Role	**Function**
SP (Service Provider)	Provides a service, such as a web page and communicates this service's high-level requirements and properties (availability, load estimates) to a VNO.
VNO (Virtual Network Operator)	Develops a virtual network's topology from the SP's description and tasks one or more Virtual Network Providers with this network's implementation.
VNP (Virtual Network Provider)	Partitions the virtual network description received from the VNO and maps the partial networks to one or more Physical Infrastructure providers, who implement the virtual networks thus described. Once all partial virtual networks are implemented, administrative access to their network elements (nodes and virtualized switches) is granted to the VNO requesting the network.
PIP (Physical Infrastructure Provider)	The PIP role operates a substrate of physical infrastructure, e.g. VM hosts and switches, on which it implements virtual networks requested by a VNP or customer. Multiple virtualized networks can co-exist on this substrate since the PIP isolates its tenants from each other.

Table 1.1: Roles of the CloudNets architecture, from PIP to VNP.

1.3.1 Resource Description Language and Topology Graphs

The central concept of `cloudnets-framework` is the *topology graph*, a network description in the resource description language FleRD[52]. These graphs consist of the following main data structures:

- *Network Elements (NEs):* these objects represent both virtual or physical machines and the links between them.
- *Network Interfaces (NIs):* Network Elements can have an arbitrary number of Network Interfaces. They model Network Elements' interconnections as doubly linked pointers between Network Interfaces.
- *Resources* and *Features*: These objects represent countable resources and free-form textual attributes of Network Elements. They are modelled as an arbitrary number of data structures attached to Network Elements.

FleRD exists in two forms:

1. Each role maintains a MySQL[14] database in which FleRD objects are stored as database tables and accessed through an ActiveRecord[27] based, object-relational interface.
2. For communication with other roles FleRD objects or whole graphs can be serialized to the textual YAML[59] format for transmission.

For further information on see [52] for a high-level overview and [25] for comprehensive reference documentation on the implementation and semantics of FleRD objects.

1.3.2 Virtual Network Mapping

Each role in the CloudNets architecture maps[4] *Overlay (OL) Graphs* describing virtual networks to a *Substrate Graph* or *Underlay (UL) Graph* describing the infrastructure they are to be implemented on[5].
In the case of the PIP the substrate graph describes physical infrastructure, i.e. physical servers interconnected by a physical network. All other roles' substrate graphs consist of instances of subordinate roles with their (known) interconnections, i.e. a VNP's substrate graph consists of PIPs and the peering links they chose to reveal to the VNP in question. Mappings are recorded in a special kind of FleRD graph the *Mapping Layer (ML) Graph*. Figure 1.3.2 shows these various kinds of graphs as they pass through or are created in the course of the mapping process. An in-depth explanation of the mapping process can be found in the *Life of a CloudNet Section* of [26].

1.3.3 Embedding Plugins

Once mapping is completed, the *Embedding Component* implements the virtual network as described by the mapping layer graph. This is a plugin-based subsystem that looks up the hosting substrate Network Elements for each of the VNet's Network Elements and invokes the appropriate embedding plugin based on the substrate network element's type

[4]Mappings are computed by formulating and solving a mixed integer problem [51, pp. 47].

[5]There is an intermediate step: OL graphs as they arrive from the customer/higher-level role (OL0 graphs) can be incomplete. This leaves the choice on the omitted information (e.g. CPU architecture or hypervisor type of VNodes) to the implementing role. Once made, these choices are recorded in the OL1 graph, which is then considered in the mapping process.

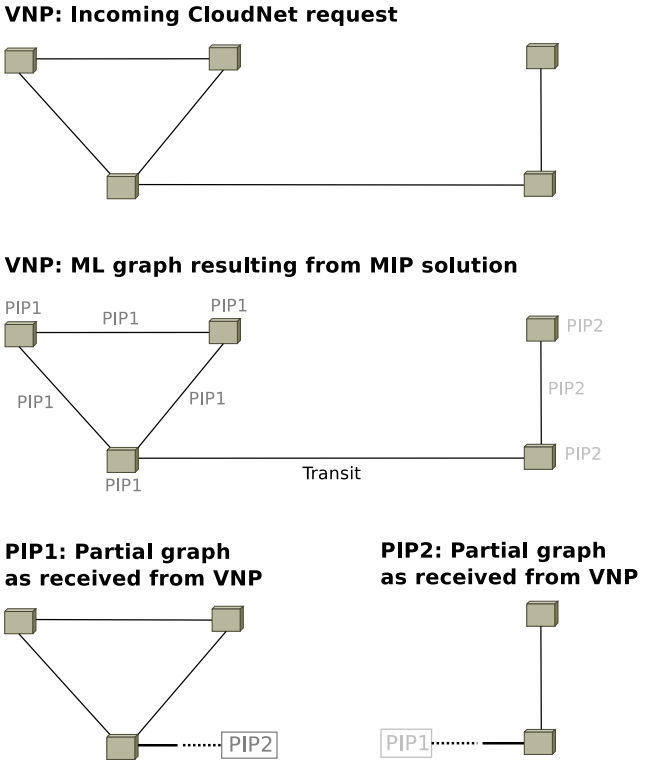

Figure 1.2: Life cycle of a virtual network from a VNP's perspective (previously published in [26]).

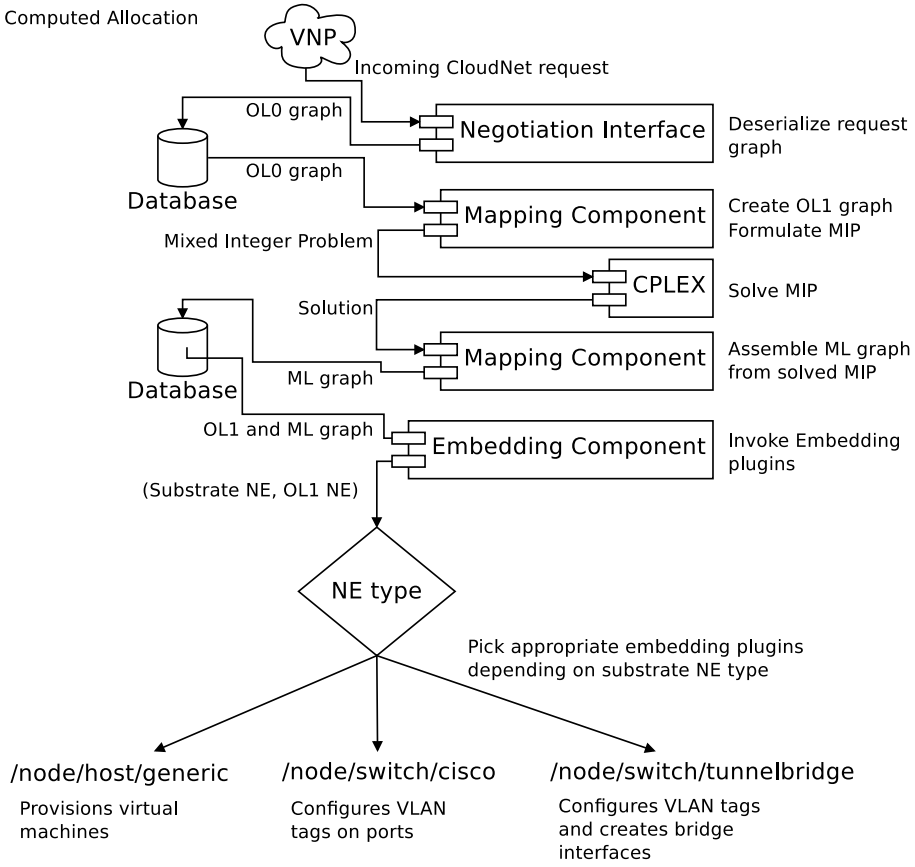

Figure 1.3: Life cycle of a virtual network from a PIP's perspective (previously published in [26]).

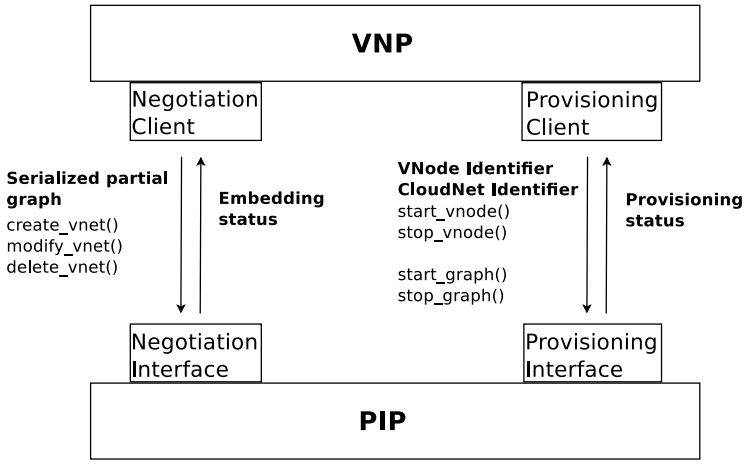

Figure 1.4: Overview of the negotiation and provisioning interfaces between a PIP and a VNP.

for each <Substrate Network Element, Overlay Network Element> tuple. The bottom portion of figure 1.3.2 shows a selection of embedding plugins. Note that there are no embedding plugins for link type Network Elements. This is due to the fact that virtual links are always instantiated through configuration of the nodes on either end of their constitutent segments.

1.3.4 Negotiation and Provisioning Interfaces

Finally, the glue that enables communication between roles in the CloudNets ecosystem consists of two XMLRPC interfaces exported by each role to roles further up the chain: The *Negotiation Interface* and the *Provisioning Interface*. The former interface provides access to the role's mapping subsystem. Its methods accept overlay graphs in FleRD's YAML format and pass them to the mapping subsystem to compute allocations for new VNets or updated allocations in the case of VNet modification. The latter comes into play once the mapping process is finished: It encapsulates the provisioning methods of the role's embedding plugins. These include starting and stopping network elements, among others. Figure 1.3.3 shows a schematic overview of both interfaces. Refer to [24] and [24] for detailed reference documentation on the Negotation and Provisioning interface, respectively.

1.4 Link Negotiation as Implemented in `cloudnets-framework`

At present, there is no link parameter negotiation in `cloudnets-framework` . All PIPs and VNPs participating in a Link Domain share the available VLAN name space as follows:

1. Each VNP is assigned a range of VLANs it is allowed to use for transit links. The VNPs' VLAN ranges do not overlap, i.e. VNPs are guaranteed exclusive use of their VLAN ranges.

2. In the course of the mapping process a VNP assigns unique VLANs from this VLAN range to all transit links (links crossing PIP boundaries) occuring in a graph.

3. All transit links are annotated with the chosen VLAN tag in the partial graphs sent to a VNet's implementing PIPs.

4. The PIPs then assign the VLANs thus communicated by the VNP to each transit link.

1.5 Drawbacks of the Current Implementation

While very simple to implement — and entirely sufficient for a proof-of-concept implementation — this approach would raise a number of problems if used in production and at scale, among them the following.

1.5.1 Namespace exhaustion

First and foremost, the current practice of pre-allocated VLAN ranges is not very scalable. Given a number of 396[6] VNPs establishing transit links through a link domain this approach would leave each VNP with an amount of 10 transit links it could establish at any given point in time. Since some VNPs will establish less than 10 transit links, while others might establish more (given more VLAN tags), this approach is highly wasteful.

1.5.2 Abuse potential

Currently there is no entity ensuring either PIPs or VNPs play by the rules. This permits the following abuse scenarios:

- Both PIPs and VNPs may configure their transit interfaces or specify their transit links with the VLAN tags of VLinks they VLANs they are not authorized to receive. This scenario can be mitigated by modelling the transit zone between PIPs as separate transit PIPs with the sole authority to configure the switch gear in between. This mitigation approach essentially amounts to introducing the Link Authority implemented in this work.
- VNPs may set transit VLANs they know to be from a competitor's assigned VLAN range in order to eavesdrop on or interfere with said competitor's VNets' traffic.

1.6 Improving Link Negotiation

In this thesis I propose to improve link negotiation by implementing a negotiation protocol for use by VNPs and PIPs, that assigns the available name space efficiently and tackles the security issues of the current approach. To this end I will first survey existing link virtualization technologies to determine the configuration parameters the protocol needs to synchronize (Chapter 2). Then I will determine the requirements and challenges of link parameter negotiation at both the PIP and VNP level and develop the high-level architecture and semantics of the protocol (Chapter 3). Finally, I will describe my protocol implementation, the Transit Link Negotiation Protocol and compile a list of tasks involved in integrating it in the `cloudnets-framework` network virtualization system.

[6] 396 is the number of peering partners that have recently been observed in a study of a large European IXP[2].

Chapter 2

Survey of Link Virtualization Technologies

I have identified three link virtualization technologies that can be used to implement subsets or all of what can be expressed in terms of the FleRD resource description language[52]: IEEE 802.1Q, The MPLS (Multi Protocol Label Switching) based Virtual Private LAN Service (VPLS) and OpenFlow based link virtualization. In this chapter I will examine these technologies with a focus on the configuration parameters required to establish virtual links using them.

2.1 IEEE 802.1Q (VLANs)

The simplest approach to link virtualization is IEEE 802.1Q[38], an extension to the Ethernet[36] protocol, also known as VLANs (Virtual Local Area Networks). Conceptually it is very simple. It is based on a 12-bit header added to Ethernet frames, the so-called *VLAN tag*. An Ethernet switch implementing IEEE 802.1Q will then base its forwarding decisions on this VLAN tag, separating Ethernet broadcast domains from each other by VLAN tag. In practice, this is usually implemented by associating sets of switch ports with a common VLAN tag, thus dividing the switch up into multiple independent broadcast domains.

2.1.1 VLANs as used by `cloudnets-framework`

In `cloudnets-framework` VLANs are currently used to separate VLinks from each other: each VLink is assigned a unique VLAN tag. It uses the software switch OpenVSwitch on VM hosts to set this VLAN tag on all VNode network interfaces participating in this VLink. If desired, `cloudnets-framework` can configure the (physical) switch interconnecting substrate nodes to limit propagation of a VLink's VLAN to only the substrate nodes hosting VNodes participating in this VLink.

2.1.2 Configuration parameters

In terms of configuration parameters VLANs are exceptionally simple. Purely topological information ("Which waypoints does a VLink pass through?") aside, the only configuration parameter is a VLAN tag that needs to be the same on all involved virtual machines and that the interconnecting switch needs to permit on all ports. The topological information (the aforementioned waypoints) is taken care of by FleRD in the current implementation: once a VLAN tag is chosen for a given VLink all endpoints and intermediate waypoints are annotated with a Feature object containing this VLAN tag.

2.1.3 Number of available VLinks

Mathematically there is a total of $2^{12} = 4096$ possible VLAN tags. In practice, there are just 4094 usable VLANs (and hence individual VLinks) since VLAN tag 0 and VLAN tag 4096 are reserved values[38, p. 150]. This rather small amount of available logical networks may may be problematic in link domains with a large number of customers.

2.2 MPLS/VPLS

Another approach to link virtualization is the MPLS (Multi Protocol Label Switching) protocol[48] that can be used to encapsulate Ethernet traffic, among other things. For the purposes of link virtualization by `cloudnets-framework` I want to single out VPLS (Virtual Private Lan Service) protocol based on MPLS and the BGP (Border Gateway Protocol) routing protocol[46], since it shares a key property with IEEE 802.1Q: it supports the point-to-multipoint links, that FleRD's half-duplex links allow for.

2.2.1 A short primer on MPLS

MPLS inserts a variable length header between[1] a packet's layer two and layer three headers. This header consists of one or more *label stack entries*, each of which holds a 20 bit*label value*. This implements the *MPLS label stack* [48, p. 7], the central element of the MPLS protocol: As a packet enters a MPLS network, a label is pushed onto[2] the packet's label stack. Henceforth each router/switch in the MPLS network will base its forwarding decisions (and possibly label manipulations) solely on the packet's MPLS label. Finally, the MPLS label is removed[3] as it reaches its destination, commonly a router/switch at the edge of the MPLS network.

2.2.2 VPLS: Building on MPLS

The VPLS (Virtual Private LAN Service) protocol described in RFC 4761[30] uses MPLS to implement virtual links between an arbitrary number of customers/sites, each connecting to the virtual topology through a *Provider Edge Router* (henceforth referred to as PE routers or edge routers). PE routers exchange information about VPLS instances using the Border Gateway Protocol, a routing protocol [46].
Creation of a VPLS instance takes place in two stages: First, a unique *Route Target Identifier* for a VPLS instance is chosen[30, pp. 7], and each PE router is assigned a unique *Virtual Edge Identifier* or VE Identifier[30, pp. 11]. The VE Identifier is a two-byte unsigned integer, i.e. a number in the range $[1 - 16384]$.
The second stage takes place automatically: All edge routers participating in this VPLS instance advertise membership in the Route Target community with the instance's Route Target Identifier to their BGP peers[30, p. 8]. In a second step, each edge router assigns a unique *demultiplexor* (which is a MPLS label) for the VPLS instance to each of its peers using the VPLS protocol's label block negotiation scheme[30, pp. 8]. This scheme is based on advertisement of a label base B and label offset O. Each receiving router with VE ID V computes its demultiplexor/label for traffic sent to the advertising router using the formula $B + V - O$[30, p. 10].
Edeg routers will use these unique demultiplexors in forwarding VPLS traffic to each other. Thus each edge router will know (a) which MPLS label to use to send a packet to a

[1] Due to this insertion between other headers it is often referred to as *shim header*, e.g. in [48, p. 24].

[2] Note that this allows for labels already being on the stack, e.g. from a neighbouring MPLS network.

[3] Also referred to as "popped"[48, p. 13]

Figure 2.1: Sample VPLS environment (graphical representation of Fig. 1 from [30]).

given destination while (b) the packet's receiving edge router will know where the packet originated permitting it to map the packet's origin MAC address to the originating edge router for future use.

2.2.3 Configuration parameters

If it is to be employed as a link virtualization technology by `cloudnets-framework` VPLS requires two configuration parameters[4]:

1. A unique (in the scope of the link domain) numerical Route Target Identifier for each virtual link. According to RFC 4360[50] this can be any 6 byte[5] integer. This identifier is chosen as the virtual link is first established and needs to be communicated to each infrastructure provider participating in the link.

2. A unique (in the scope of the link domain) Virtual Edge Identifier, i.e. a 16 bit integer for each link endpoint. Each infrastructure provider participating in the virtual link is assigned a Virtual Edge Identifier.

2.2.4 Number of available VLinks

The number of total VLinks per link domain largely depends on the number of endpoints per link. This is best illustrated by working out the number of MPLS labels required by a virtual link with n participants (where n is a non-zero, positive integer). For the purpose of the following calculation I will assume the trivial case of just one MPLS hop per link domain:
In this case there is a total of $2^{20} - 2^8 = 1048320$ MPLS labels available per link domain[6].
To establish a link with n endpoints, each participating provider needs to assign $n - 1$ unique MPLS labels, one for each other provider involved in the link.
Hence, a total of $n * (n - 1)$ labels is required per link. Adjusting for the reserved MPLS labels, and assuming n ist constant for all virtual links we can compute upper bounds $L(n)$ on the number of available virtual links for various values of n (see table 2.1) as follows:

$$\begin{aligned} L(n) = & \quad \frac{2^{20}-2^8}{n(n-1)} \\ = & \quad \frac{2^8(2^{12}-1)}{n(n-1)} \end{aligned}$$

Starting from links of 17 participants, VPLS performs worse than IEEE 802.1Q in terms of total available virtual links. Since this would require a large number of links with a high number of endpoints this is unlikely to arise in practice. Should this limitation become a problem it could possibly be mitigated by leveraging the MPLS protocol's label stack: one could divide a link domain's providers into subnets much like the IP protocol's prefixes[22, pp. 4]: each subnet would then consist of multiple providers using a gateway to

[4]This assumes that all involved entities use BGP to negotiate the remaining parameters among themselves as described in RFC 4761[30], of course, and that there is full mesh connectivity between PE routers.

[5]While RFC 4761 mentions a 8 byte field, only 6 bytes are usable for a Route Target Identifier since the two highest-order bytes are required to denote the BGP community type 'Route Target'[50, p. 1 and and 5].

[6]The 256 missing values are due to the fact that the first 256 MPLS labels are reserved values[60].

n	**number of virtual links**
2	524160
3	174720
4	87360
5	52416
6	34944
7	24960
8	18720
9	14560
10	11648
11	9530
12	7941
13	6720
14	5760
15	4992
16	4368
17	3854
18	3425
19	3065
20	2758

Table 2.1: Upper bounds on available VLinks, assuming a constant number of participants n across all links.

connect to other subnets. This gateway would then examine the VPLS labels in outgoing packets' label stack to determine the subnet to route the packet to. It would then add a label to the packet's label stack, ensuring it gets forwarded to this subnet. The receiving subnet's gateway would then examine the original VPLS label on the stack to determine the packet's destination provider. This way, labels used by VPLS would only have to be unique in the scope of the involved subnets' scope, but not globally. This would require the link domain to participate in all VLinks' BGP communication, greatly increasing its complexity. Hence I will leave this as a lead for future research to pursue, should the discussed label exhaustion become a problem.

2.3 OpenFlow Based Link Virtualization

As the third and final technology in my survey I want to consider the OpenFlow software defined networking architecture. Originally proposed in a 2008 whitepaper[37], OpenFlow itself is not a link virtualization technology, but has been used to build link virtualization technologies, among them the VeRTIGO architecture I am going to base the OpenFlow related parts of my work on.

2.3.1 A short primer on OpenFlow

Before I go into detail on the VeRTIGO architecture I would like to briefly introduce the key concepts of the OpenFlow architecture. The 2008 whitepaper introducing it[37] lists three key elements of OpenFlow, or rather its central device, the OpenFlow switch:

- A *flow table* consisting of *flow entries*. The flow table entries are matching rules for packet headers with associated actions and govern the switch's forwarding decisions.

- A *secure channel* connecting the switch to a *controller* that is consulted for forwarding decisions not covered by the switch's flow table.
- The *OpenFlow Protocol*, a standardized bidirectional protocol for controller/switch interaction.

The switch will forward packets much like a traditional switch as long as packets are matched by a rule in its forwarding table. OpenFlow comes into play where that is not the case: packets not matched by flow table entries can be forwarded to the controller. The controller, typically implemented in software on a commodity server, will then make a forwarding decision and can additionally modify the switch's flow table to allow the switch to directly handle packets of this kind in the future.

2.3.2 FlowVisor and Vertigo: OpenFlow based link virtualization

Not long after OpenFlow appeared on the stage, Sherwood et al. leveraged it in their *FlowVisor* system[57] to partition OpenFlow networks into what they refer to as *slices*. These slices are defined as a set of OpenFlow flow table entries.
In the FlowVisor system, the OpenFlow switches are configured to use FlowVisor as a controller. Each slice is then assigned a dedicated controller responsible for making the slice's forwarding decisions. FlowVisor then acts as a transparent proxy between the OpenFlow switches constituting this slice and its assigned controller, i.e. OpenFlow protocol messages are filtered such that only messages relevant to a slice reach its controller and vice-versa.
Building on FlowVisor, VeRTIGO[16] extended FlowVisor with the notion of *virtual links*. A virtual link is defined in the scope of a FlowVisor slice and can consist of an arbitrary number of physical or logical segments. Of these segments only the endpoints will be visible to the associated slices' controller, with forwarding between these endpoints being handled by VeRTIGO.

2.3.3 Configuration parameters

VeRTIGO's virtual links are defined in terms of OpenFlow datapath IDs[7], along with switch ports. A virtual link consists of multiple DPID/Port pairs[17].

2.3.4 Number of available VLinks

While theoretically there is no limit on available virtual links (a single physical port may terminate multiple virtual links[16, p. 26]) on the VeRTIGO side, there is on the FlowVisor side: although there may be multiple virtual links terminating on a single physical port, any two of these virtual links need to be defined in the scope of mutually distinct slices. FlowVisor, in turn, needs a mechanism to identify any packet's associated slice. For this purpose it can use the following identifiers:

1. Various packet header fields, among them source and destination IP and MAC addresses and VLAN tags[56].
2. The physical ports the packet ingresses from.

These limitations stem from OpenFlow's packet header based flow matching capabilities[43, p. 60] and are thus a limitation all OpenFlow based link virtualization technologies share.

[7] A data path ID (DPID) is an identifier for an OpenFlow switch instance, consisting of the switch's MAC address and a 16 bit value that is up to the switch's implementer [43, pp. 71]

2.3.5 Summary

Due to its limitations, OpenFlow based link virtualization is mainly useful in the scenarios originally envisioned by FlowVisor and VeRTIGO:

- Hiding details of the physical network's topology[16, p. 25].
- Segregating networks on a shared physical network on the basis of arbitrary packet headers that need not neccessarily be VLAN tags[57, p. 4].
- Logically partitioning a physical topology on the basis of switch ports.

It is not useful in the virtualization of layer 2 links as required by the current implementation of `cloudnets-framework` . The existing implementations of FlowVisor and Vertigo would add very little value on top of either IEEE 802.1Q[38] or MPLS[48], since they would mandate use of these protocol's header fields to allow for layer 2 link virtualization that does not restrict the values of IP header fields for any virtual network.
While it is conceivable to combine certain packet headers (for instance, the VLAN tag and the MPLS label) to leverage OpenFlow's packet matching abilities, the currently existing implementations lack this capability. Developing a feasible scheme for employing combined packet headers in a link virtualization scenario is well beyond the scope of this thesis. Hence I will not include OpenFlow based virtualization technologies in my link negotiation protocol's design considerations[8].

[8] I will leave room for it, though: The protocol is designed to be extensible. Hence future link virtualization technologies, such as an extended OpenFlow specification, can be included as they emerge.

Chapter 3

Protocol Architecture

In this chapter I will give a high-level overview of the Transit Link Negotiation Protocol and discuss the choices underlying its architecture. The chapter is organized as follows: In section 3.1 I will list the main functional requirements the protocol needs to fulfill. These requirements will later be used as true north to evaluate my architectural and design desicions against, in both this and the following chapter. In section 3.2 I will discuss my decision to use a centralized topology (as opposed to a peer-to-peer topology) for interconnecting the entities using the Transit Link Negotiation Protocol. Section 3.3 provides an overview of the parties to the protocol. The remaining sections deal with the protocol's semantics and state to be transmitted and stored. They form the basis of the implementation discussed in the next chapter.

3.1 Requirements

The Transit Link Negotiation Protocol needs to fulfill the following main requirements:

1. It needs to be able to reliably synchronize a list of available and occupied local link identifiers between all providers participating in a link domain. There must be no disagreement about this list, since this may cause disruption of established transit links or connections between unrelated transit links.

2. It needs to be robust against malicious providers trying to sabotage logical links. This protocol may be used by competing entities, some of whom may not be above trying to disrupt or eavesdrop on their competitors' customers' communications. Hence it should guard against such abuse as to the extent possible.

3. To remain usable in the future the protocol needs to be extensible. New requirements may become evident or new link virtualization technologies may emerge, both of which may require protocol changes.

With these requirements set down, I shall now develop a protocol design able to accomodate them.

3.2 Type of protocol: Peer-to-peer vs. Centralized Oracle

The first question in designing the Transit Link Negotiation Protocol is whether to organize it in a centralized or peer-to-peer fashion. My first tendency was to use a peer-to-peer protocol, since this obviates the need for a trusted central entity maintaining a repository of local link identifiers. This appproach is fraught with problems, though, and I ended up opting for the centralized approach. Here is my rationale for arriving at this decision:

Fundamentally, fulfilling the requirements outlined in the previous section in a peer-to-peer protocol is equivalent to achieving distributed consensus, a problem theoretical computer science has been working on for years. One of the earliest approaches to tackling this problem is Leslie Lamport's Paxos algorithm[33]. While this algorithm is rather hard to understand (and consequently hard to implement), even by the author's own admission[32], it has been in existence for quite some time, and there are various reference implementations that could be used for the protocol's purposes. It lacks one key property, though: it is resilient to random failures (due to entities dropping out of the communication randomly, for instance), but it does not exhibit *byzantine fault tolerance*[34], i.e. resistance against deliberately malicious behaviour by one or more participating entities. Hence it cannot fulfill requirement 2.
There have been several attempts[1, 6] to introduce byzantine fault tolerance into Paxos[1]. The most promising of these is Practical Byzantine Fault Tolerance (PBFT), but it has been shown to be less than practical by Chondros et al "On the practicality of practical Byzantine fault tolerance"[7]: the article's authors surveyed existing implementations of PBFT and identified the hurdles that would be involved in using PBFT in a real-world system, using an electronic voting system as an example of such a system. They conclude that "[using] existing PBFT implementations [requires] significant engineering effort" and caution against "unclear performance ramifications"[7, p. 12].
Since this work is primarily about implementing a link parameter negotiation protocol and not about improving an existing implementation of PBFT to the point where it can be integrated into a real-world system, "significant engineering" would be well beyond the scope of a three-month thesis. Hence I opted for a centralized approach, organized as follows:

- For each link domain the participating entities agree on a neutral third party (in the case of an internet exchange point (IXP) this might be the exchange's operator, financed by the IXP's membership fees, for instance).
- The neutral third party runs a service that maintains an authoritative list of local link identifiers for the link domain.
- Whenever a participating provider needs to establish or remove a logical link across provider boundaries it communicates its wishes to the third party and receives an authoritative response.

The centralized approach trades the transparency and prefect auditability of a distributed consensus protocol for greatly reduced complexity. While one may still introduce provisions for auditability, there will always be a potential for undetected abuse by the central authority.
Nonetheless I chose the centralized approach, since I believe avoiding the "significant engineering" Chondros et al allude to trumps the transparency gains that would come at the expense of a much more complex protocol. Also, the Link Authority is likely to take the shape of an an independent corporate entity financed by membership fees. Hence the participants in a Link Domain would still "vote" to agree on local link identifiers, but indirectly by assigning the mandate for picking them to a Link Authority a majority of participants agrees on. Last but not least, the centralized approach has been used successfully before. One prominent example is the Internet's Domain Name System. It is based on IANA acting as the world's definitive authority on the system's root zone[47]. In

[1]While originally regarded as a different algorithm, The Practical Byzantine Fault Tolerance[6] algorithm described by Castro et al. has been proven to be equivalent to a modified Paxos Algorithm by Lamport[31].

reminiscence of IANA's name[2] I shall refer to the protocol's neutral third party as *Link Authority* for the remainder of this work.
Once a central authority is in place, the functional requirements outlined in the last section can be fulfilled as follows:

1. Reliable synchronization of local link identifiers is a non-issue if the Link Authority is the one authoritative source of such identifiers.

2. Since the Link Authority is the one choke point all information passes through, it can implement authentication and authorization, ensuring that PIPs only have access to local link identifiers they are authorized to know. Additionally, it may enforce local link identifiers if it controls the switch ports PIPs connect through.

3. In principle, extensibility is possible for both the peer-to-peer and the centralized approach. That being said, the provisions for extensibility are more of an implementation detail and less of an architectural concern. Refer to section 3.6.1 for a birds-eye view of these provisions and to section 4.2.6 for a detailed discussion of their implementation in the Transit Link Negotiation Protocol.

3.3 Communicating entities

In this section I will list the entities involved in the link negotiation protocol and their high-level goals and responisibilities.

3.3.1 Link Authority

The link authority maintains an authoritative database of link metadata for each virtual link established through a link domain. It is the central element in fulfilling the function of reliable link parameter syncronization. The link authority is the sole point of contact for:

- Requesting a new virtual link between two or more PIPs connected across its link domain.

- Requesting configuration parameters for participating in an existing virtual link.

- Modifying the set of PIPs allowed to participate in an existing virtual link.

- Decommissioning existing virtual links between two or more PIPs.

3.3.2 Physical Infrastructure Provider (PIP)

Physical infrastructure providers maintain peering links with one or more other PIPs through one or more link domains[3]. If a VNP requests virtual networks that involve transit links from them, they in turn request new links or the configuration parameters required to join existing links from the link authority.

[2]Internet Assigned Numbers Authority.

[3]While there may be PIPs without any peering links whatsoever, they are irrelevant as far as this protocol is concerned, since they cannot establish virtual links to other providers. Hence they have no link parameter negotiation needs.

3.3.3 Virtual Network Provider (VNP)

The VNP initially defines virtual links to be established between PIPs it requests virtual networks from. It decides on creating new links, deleting existing links and administrating the set of PIPs authorized to participate in existing transit links it owns. The VNP communicates its decisions to the link authority responsible for the affected link domain and receives authentication tokens to pass on to the PIPs.

3.4 Authentication

I already mentioned authentication tokens in the previous section. These tokens are used on both the VNP level and the PIP level, to authenticate modifications to an existing transit link and to authenticate the retrieval of configuration parameters required to join the link, respectively. I identified two possible approaches to implementing such authentication tokens:

3.4.1 Public-key cryptography

This approach requires VNPs and PIPs to generate cryptographic certificates. They then optionally present these certificates to a certificate authority vouching for their membership in either group for signing (this has the added benefit of being able to restrict the circle of entities permitted to interact with a link authority). In their interactions with link authories they sign all protocol messages with this their public key. The link authority turn uses this signature to verify that the sender of a message is indeed who they claim to be and accepts the protocol message if its internal records authorize the sender to perform the protocol step in question.
This approach is probably preferable in a production scenario, if difficult to implement. As an added benefit it would allow for confidential, authenticated communication proxied through PIPs, no longer requiring the VNP to have direct access to any link authorities.

3.4.2 Shared secrets

In this considerably simpler approach the link authority or VNP would generate two kinds of shared secret, both communicated to the VNP upon link creation. The first grants the VNP administrative access to a virtual link, allowing deletion and modification of its set of members. The second is passed on to PIPs as part of the partial graphs containing the link in question and allows them to join the virtual link.
In this work I opted for the shared secret approach, since a cryptographically sound implementation of the first approach would require a considerable amount of engineering, worthy of a thesis all by itself.

3.5 Link virtualization technologies and their synchronization requirements

Building on my survey in chapter 2 I will now compile a list of logical link parameters the Link Authority needs to keep track of to fulfill providers' synchronization needs. Much like chapter 2, this section will consider VPLS and IEEE802.1Q separately.

3.5.1 IEEE 802.1Q (VLANs)

There are two configuration parameters required to fully define a transit link implemented using IEEE802.1Q:

1. The VLAN tag identifying the link on all switch ports it is to be forwarded through.

2. The set of switch ports the VLAN is to be forwarded through, i.e. all the switch ports connected to the PIPs participating in the transit link.

3.5.2 VPLS

The set of configuration parameters for VPLS is a little more complex. First, there are the obvious configuration parameters already mentioned in 2:

1. The Route Target Identifier identifying the link in the BGP messages exchanged between PIPs.

2. The set of Virtual Edge Identifiers, one for each PIP participating in the transit link.

Additionally the PIPs need to be able to exchange BGP messages, hence they need

3. A list of the involved PIPs' BGP speaker IP addresses.

There are two possible approaches to distributing this information:

1. Distribute it ad-hoc as transit links are established.

2. Upon joining a link domain a PIP's BGP speaker address is distributed to all other member PIPs and all existing PIPs establish BGP peering relationships with the new PIP.

Option (1) will create large amounts of redundant, distributed state that needs to be maintained when membership changes: a PIP may participate in multiple transit links, hence it will occur in other PIPs' lists of BGP peers multiple times.
Option (2) is simpler, both in terms of protocol complexity and state to be maintained. This is due to every PIPs' ability to establish a BGP peering relationship with every other PIP connected to a link domain: this wayPIPs participating in a transit link can exchange the BGP messages required to negotiate the link's component pseudo wires without setting up new BGP peerings first.
I chose a hybrid approach that amounts to option 2 in the protocol's implementation: while the data model allows for per-link BGP speaker addresses, the Transit Link Negotiation Protocol will set this attribute to a default per-PIP value that is configured out-of-band as a PIP joins the link domain.
I took the liberty of this simplified approach based on the assumption that BGP peerings between all link domain members exist and are static. Thus a PIP joining or leaving a link domain should be a fairly rare event. After all it usually requires physically connecting the PIP to the link domain.

3.6 Protocol Steps

In this section I will formalize all protocol steps and give a high level overview of protocol messages' contents. Please refer to Appendix A for an in-depth specification of protocol messages.

3.6.1 Protocol Version Handshake

Initiating party:	PIP or VNP
Responding party:	Link Authority
Data sent by initiating party:	Request for supported protocol versions
Data sent by responding party:	List of supported protocol versions

To allow for extensibility, protocol communication needs to be preceded by a version handshake. This version handshake ensures both parties in any protocol exchange speak the same Transit Link Negotiation Protocol version. The protocol version handshake must be performed successfully before any other protocol messages are exchanged. The version handshake takes place as follows:

1. The party initiating the communication requests a list of supported protocol versions from the responding party.
2. The responding party responds with its list of supported protocol versions.
3. The initiating party choses the highest protocol version it supports from this list. It must signal this protocol version to the responding party as part of all further protocol messages.

This process will terminate with the highest Transit Link Negotiation Protocol version supported by both parties if at least one common protocol version exists. If there is no common protocol version no other protocol messages may be exchanged until a successful protocol version handshake. The responding party must ignore or reject any attempts to use unsupported protocol versions.

3.6.2 VNP level protocol messages

These protocol steps are exchanges between VNPs and link authorities. They cover commissioning, decomissioning and modification of transit links.

Create Link

Initiating party:	VNP
Responding party:	Link Authority
Data sent by initiating party:	Unique identifier for this VNP List of PIPs allowed to connect
Data sent by responding party:	Link status (acknowledgement or refusal) Link identifier (unique in this link domain's scope) List of access secrets for PIPs to use Shared secret for link administration

This is the first actual protocol step. A VNP must perform this step after it has finished mapping a VNet graph to its substrate graph of PIPs and peering links, and before passing any of the resulting partial graphs to PIPs.
For each transit link resulting from the mapping process a link creation message is sent to the link authority responsible for the link domain it crosses. This message consists of:

- The VNP's *globally unique identifier* (e.g. a fully qualified DNS[40] domain name).
- A *list* of the link's prospective member PIPs' globally unique identifiers (e.g. their DNS domain names).

The link authority may either reserve link parameters as required and acknowledge link creation or refuse if link parameters cannot be reserved (e.g. if there are not enough VLANs or MPLS labels available to create the link). In the latter case the VNP may either compute a new mapping that avoids the link authority in question or retry link creation until it succeeds. In the former case the link authority responds with:

- A *status code* that indicates successful link creation.
- A *link identifier*. This identifier identifies this link in all future protocol steps, both from the VNP and the link's member PIPs.
- A *shared secret* authorizing the VNP for administrative access to the link in future interactions.
- A *list* of *PIP identifier/access secret* tuples (the link's member PIPs may later use these access secrets to join the link).

Add PIPs to Link

Initiating party:	VNP
Responding party:	Link Authority
Data sent by initiating party:	Link identifier Shared secret for link administration List of new PIPs allowed to connect
Data sent by responding party:	Link modification status (acknowledgement or refusal) List of shared secrets for new PIPs to use.

Once a link is established, a VNP may send this protocol message to grant additional PIPs permission to join the link. It must contain the following information:

- The link's identifier received at link creation (see section 3.6.2).
- The link's *shared secret* for administrative access (see section 3.6.2).
- A *list* of PIPs to add to the link.

The link authority responds with a *status code* indicating success or failure and a list of *PIP identifier/access secret* tuples assigning the newly added PIPs the shared secrets they need to join the link. Since the PIPs already participating in the link may need to take action to add the new PIPs to the link (see 3.5.2 for a scenario where this is required).

Remove PIP from Link

Initiating party:	VNP
Responding party:	Link Authority
Data sent by initiating party:	Link identifier Shared secret for link administration List of PIPs to remove from link
Data sent by responding party:	Link modification status (acknowledgement or refusal)

Once a link is established, a VNP may send this protocol message to remove PIPs from link. It must contain the following information:

- The link's identifier received at link creation (see 3.6.2).

- The link's *shared secret* for administrative access (see 3.6.2).
- A *list* of PIPs to remove from the link.

The link authority responds with a *status code* indicating success or failure. It must purge the affected PIPs' membership records take the steps neccessary to deny these PIPs access to the link in question, if this is possible. The VNP must notify the removed PIPs that they are no longer part of the link by issuing either a modification or deletion request for the partial graph containing the link. It must also notify the PIPs remaining on the link, since they may need to take action to remove the new PIPs from the link (See 3.5.2 for a scenario where this is required.).

Remove Link

Initiating party:	VNP
Responding party:	Link Authority
Data sent by initiating party:	Link identifier
	Shared secret for link administration
Data sent by responding party:	Link removal status (acknowledgement or refusal)

The VNP must send this protocol message if a link is to be decommissioned. It must first notify the PIPs that are parties to the link. The PIPs must in turn cleanly decomission the link at their own endpoints, i.e. remove all configuration implementing the link on their side. Once all PIPs have confirmed local link decomissioning, the VNP sends the link removal protocol message. It must contain the following information:

- The link's identifier received at link creation (see 3.6.2).
- The link's *shared secret* for administrative access (see 3.6.2).

The link authority responds with a *status code* indicating success or failure. It must take the steps neccessary to deny all involved PIPs access to the link in question, if this is possible.

3.6.3 PIP level protocol messages

These protocol steps are exchanges between PIPs and link authorities. They cover link participation and exchange of the required participation.

Join Link

Initiating party:	PIP
Responding party:	Link Authority
Data sent by initiating party:	Link identifier
	Globally unique PIP identifier
	Access secret for link participation
Data sent by responding party:	Link join status (success or failure)
	Local link identifier

This step takes place for each PIP after a transit link has been created by a VNP and after the PIP has received its access secret for this link. Its purpose is communicating all required configuration parameters for the link to the PIP. The PIP authenticates itself with its globally unique identifier and the link's access secret it received from the VNP.

If authentication fails, a join status indicating failure is returned and the PIP may not participate in the link. If it is successful, the link authority must take the steps neccessary (e.g. enable switch ports or modify access control lists) to grant the PIP access to the link and returns a *join status* indicating success and a *local link identifier* (this is the set of configuration parameters the PIP needs to configure the link on its own side).

3.7 State

In this section I will outline the state kept by the parties involved in the Transit Link Negotiation Protocol. It forms the blueprint of the database schema detailed in section 4.2.1 is based on. Unlike section 4.2.1, it develops the state to be kept by protocol party, rather than by individual data objects. Hence I recommend it to readers interested in an overview of which state is kept where, since 4.2.1 is mainly concerned with the link authority's database.

3.7.1 State Kept by the Link Authority

The link authority generates and keeps most of the state associated with transit links. Regardless of link type (VPLS or VLANs) there are common metadata, which I will list first. Aside from these common metadata both link virtualization protocols require protocol specific state.
For a quick overview of most state kept by the Link Authority refer to figure B.4 in Appendix B.

Common State

The following information is required for all transit links:

- A unique (in the scope of the link domain) *link identifier*.
- A list of PIPs participating in the link.
- A list of secrets for authenticating PIPs joining the link, each assigned to a PIP.
- A secret granting the VNP who created the link administrative access.

State kept for VPLS Based Links

The following additional information is required for VPLS based transit links:

- The Route Target Identifier identifying the link in the BGP messages exchanged between PIPs.
- The set of Virtual Edge Identifiers, one for each PIP participating in the transit link.

State kept for VLAN Based Links

The following additional information is required for VLAN based transit links:

- The VLAN tag identifying the link on all switch ports it is to be forwarded through.
- The set of switch ports the VLAN is to be forwarded through, i.e. all the switch ports connected to the PIPs participating in the transit link.

3.7.2 State kept by the VNP

The VNP needs to keep all the state it receives from the link authority and pass it on to a transit link's member PIPs. The state kept by the VNP is a subset of the link authority's state and most notably excludes link virtualization protocol specific state such as VLAN tags, since PIPs receive this information directly from the link authority. It contains the following information for each transit link:

- The unique (in the scope of the link domain) *link identifier*.
- The list of secrets for authenticating PIPs joining the link, each assigned to a PIP.
- The secret granting the VNP administrative access to the link.

3.7.3 State kept by the PIP

The PIP needs to maintain both administrative and link virtualization protocol specific state for each transit link.

Administrative state

- The link's unique (in the scope of the link domain) *link identifier*.
- The secret for authenticating this PIP with the link authority when joining the link.

State kept for VPLS Based Links

The PIP needs to store the following information for each VPLS based transit link:

- The Route Target Identifier identifying the link in the BGP messages exchanged between PIPs.
- The Virtual Edge Identifier assigned to this PIP.

State kept for VLAN Based Links

The PIP needs to store the following information for each VLAN based transit link:

- The VLAN tag identifying the link on all switch ports it is to be forwarded through.

Note that it is not neccessary for the PIP to store the switch port this virtual link passes through: This information is already available from the PIP's substrate graph.

Chapter 4

Protocol Implementation

In this chapter I will describe my reference implementation of the Transit Link Negotiation Protocol in detail. I will start out with an overview of its components and their macro structure in section 4.1. Sections 4.2 and 4.3 are dedicated to the protocol's core component, the Link Authority and the client library communicating with it. Finally, section 4.5 discusses the technical challenges of integrating the protocol's reference with the existing `cloudnets-framework` code base and outlines a possible approach to integration.

4.1 Overview

The protocol's core consists of two Ruby[11] gems[1]: the `tlneg` gem which implements the Link Authority (i.e. the server side of the protocol) and the `tlneg-client` gem providing a client-side implementation of the Transit Link Negotiation Protocol[2]. On the periphery there are a three command-line executables:

- `tlnegadm` A utility for manipulating the link authority's database directly
- `tlneg` A command-line client for the Transit Link Negotiation Protocol using the `tlneg-client` gem
- `tlnegd` The link authority's server daemon

You will find an list of all files making up my reference implementation of the Transit Link Negotiation Protocol in table B.1 and an overview diagram of all components in figure B.3 (Appendix B).

4.2 Implementation of the Link Authority

The Link Authority is the most important component of the Transit Link Negotiation Protocol. It keeps all protocol state and passes subsets of this state to PIPs and VNPs as appropriate. In this section I will describe its inner workings. Since its chief task is maintaining state, I will start off with a description of its underlying data model (subsection 4.2.1) and the object-relational[42] abstraction used to access it(section 4.2.2). Next, I will discuss the HTTP[21] and JSON[15] based REST[3] interface I developed for transmitting protocol messages.

[1] *Gems*[10] are the Ruby programming language's native package management format.

[2] While in an ideal environment, a single gem might have been desirable in order to keep implementation of client and server close together, the realities of the software ecosystem I was working in made this aproach appear less than prudent. See section 4.4 for a discussion of the issues involved.

[3] Representational State Transfer[19].

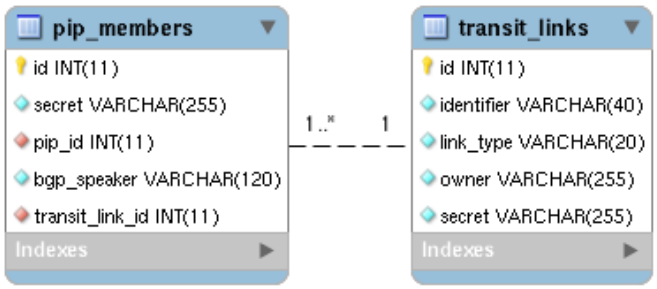

Figure 4.1: Central entities of the Link Authority's database schema: Transit links and the PIPs participating in them.

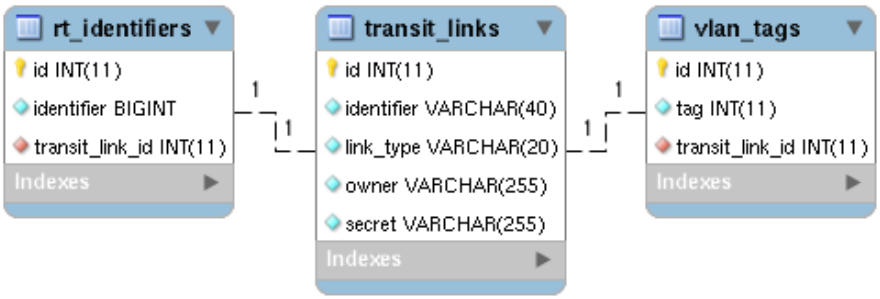

Figure 4.2: Per-link metadata in Transit Link Negotiation Protocol version 1.0: VLAN Tags for VLAN based link virtualization and Route Target Identifiers for VPLS based link virtualization.

4.2.1 Data Model

I chose to use a MySQL[14] database for persisting the link authority's state. Figure B in the appendix shows this database's Entity-Relationship diagram. Here, I will give a quick tour of this schema and its most important features, especially as far as extensibility is concerned.

The two central elements of the schema are the `transit_links` and `pip_members` tables (see figure 4.1). They represent transit links and the PIPs participating in them, respectively. Both hold meta data about transit links and their members that are independent of link virtualization technologies, e.g. link type, ownership and authentication credentials for Transit Link Negotiation Protocol protocol steps. Beyond storing these metadata, they serve as attachment points for link virtualization technology specific metadata: such metadata reference the transit links they belong to if they occur on a per-link basis (figure 4.2), or the link members they are are assigned to (figure 4.3).

Due to these central entities being referenced from technology specific entities on the schema's periphery, extending the data model with information required by new link virtualization technologies is straightforward: one simply creates the entities required and — depending on whether they occur per link or per link member — references either the `transit_links` or the `pip_members` table with a foreign key attribute.

The only element still missing from this picture are the `pips` and `switch_ports` tables. Strictly speaking, they are not required by the protocol[4], but may in the future be used to enforce the Link Authority's decisions: They hold information on which PIPs are members of its link domain, and the switch ports they connect through. The link authority uses

[4]Which is why they are populated manually and out-of-band (see subsection 4.2.8).

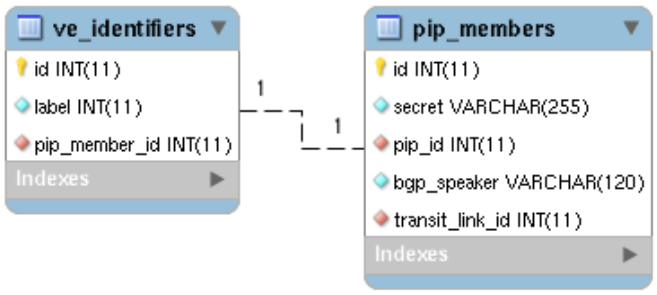

Figure 4.3: Link member specific metadata in Transit Link Negotiation Protocol version 1.0: VE Identifiers for VPLS based link virtualization.

the `pips` table as a template for creating PIP members.

4.2.2 Database Abstraction

Database tables are accessed through the ActiveRecord[27] object-relational framework. This means that each database table is encapsulated by a class representing this table. Table rows are instances of this class, with the rows fields being represented by the class' public attributes. Search queries can be performed through a range of static wrapper methods that generate SQL queries and return instances of the class. Furthermore, there are instance methods that dereference foreign key relations and return instances of the referenced type. These methods are automatically generated by the ActiveRecord framework.

Figure 4.2.2 shows these classes, omitting the automatically generated methods (interested readers may refer to the ActiveRecord documentation which is part of the Ruby on Rails API documentation [49]).

For the most part, ORM classes map to the corresponding database tables bijectively. The classes representing dynamically generated link configuration parameters (see right column of figure 4.2.2) are an exception, though: Since VLAN tags, MPLS labels and Route Target Identifiers need to be unique in the scope of a link domain they contain lookup and generation methods that are executed by the ActiveRecord constructor hook `before_create`. These methods create link parameters that are guaranteed to be unique (or raise exceptions if the namespace is exhausted).

4.2.3 Transport Protocol

Instead of starting protocol development at OSI layer[5] 5 and implementing everything beyond TCP[6] myself I decided to use the established HTTP[7] protocol for transmitting Transit Link Negotiation Protocol messages. This allowed me to use the proven mechanisms of the HTTP protocol in a range of situations where I would have had to implement analogues otherwise.

[5] [63, pp. 429]

[6] Transmission Control Protocol[45]

[7] Hyper Text Transfer Protocol[21]

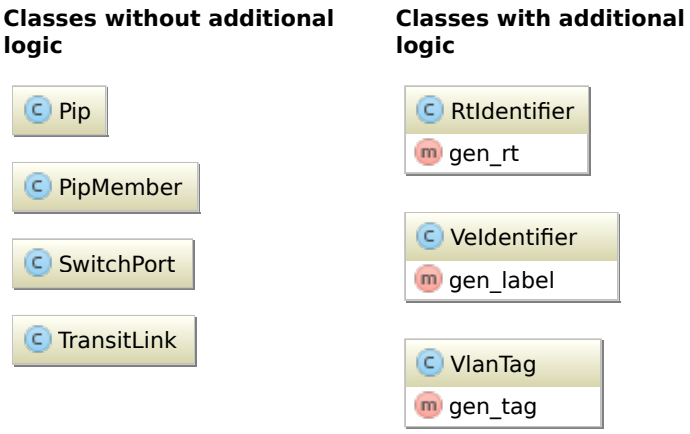

Figure 4.4: Object-relational classes used by the link authority. The right column shows classes without logic of their own (i.e. they simply provide access to the database tables behind them), the right column shows classes with processing logic for creating unique link configuration parameters.

4.2.4 Remote Procedure Call Architecture

In order to transfer protocol messages through HTTP I used Sinatra[39], a Ruby-based framework. This framework implements the REST[8] architecture originally proposed by Roy Fielding[20, pp. 76]. The REST architecture is a set of constraints for network communications, with the primary goal[9] of avoiding server-side session state in communication. In the special case of a HTTP based REST protocol all information (Transit Link Negotiation Protocol is such a protocol) sent from the client to the server in any communication is in one of two forms:

1. Part of the URI[10] identifying the resource on the server being accessed.
2. The data a client sends as header field or payload of a HTTP request.

Beyond this information a REST compliant protocol must require no additional server-side state related to an individual request, since this would constitute context. The Sinatra framework implements REST as follows: A developer can define *routes*, which are paths relative to a web server's root directory. HTTP requests for these paths are passed to a handler method defined for the path. The handler method then processes the request and returns a response to the requesting HTTP client.
In my reference implementation of the Transit Link Negotiation Protocol I implemented such handler methods for the protocol's steps and the version handshake. These handler methods are accessed through standardized paths on the web server hosting the Link Authority. Upon link creation the Link Authority generates a UUID[11] identifying the link. This UUID is later used as part of the URI for all protocol steps that act on specific links. Table 4.1 shows an example list of URIs for a link authority hosted on the server `example.com`.
Clients use HTTP GET[21, p. 53] requests for retrieving the server's list of supported protocol versions and POST requests for all further protocol steps. The body of these

[8] Representational State Transfer[19].

[9] The REST architecture contains further constraints, such as cacheability, but I consider statelessness its most important property.

[10] Uniform Resource Identifier[4].

[11] Universally Unique IDentifier[35].

URI	HTTP Method	Protocol step
`http://example.com/tlneg/versions`	GET	Version Handshake
VNP Level steps		
`http://example.com/tlneg/1.0/link/create`	POST	Create Link
`http://example.com/tlneg/1.0/link/<ID>/addpips`	POST	Add PIPs
`http://example.com/tlneg/1.0/link/<ID>/removepips`	POST	Remove PIPs
`http://example.com/tlneg/1.0/link/<ID>/delete`	POST	Delete Link
PIP Level steps		
`http://example.com/tlneg/1.0/link/<ID>/join`	POST	Join Link

Table 4.1: Transit Link Negotiation Protocol version 1.0 URIs for a Link Authority running on the host `example.com`. "<ID>" is a placeholder for a particular link's UUID.

POST requests contains a serialized representation of the data structures sent by the client. The server responds with a serialized data structure of its own, containing the response specified for this protocol step. Refer to section 3.6 or Appendix A for details on the information sent/received in each protocol step.

4.2.5 Data Serialization

On both the client side protocol state is representend in memory as Ruby data structures. The subset of this state that is sent in protocol messages can be expressed in terms of Ruby's Hash and Array data structures and elementary types, all of which offer a `to_json()` method for easy serialization to the JSON[15] format. A detailed specification of this JSON based protocol payload can be found in appendix A.

4.2.6 Protocol Version Handshake

The protocol version handshake specified in subsection 3.6.1 is implemented as follows:

1. A client sends a HTTP GET request for the link authority's supported versions list at `/tlneg/versions`.
2. The server responds with a list of protocol versions it supports.
3. The client picks the highest version supported by itself from this list and generates a base URI containing the server's identifier for this protocol version.
4. Subsequently, the client will use this base URI for all protocol interactions, thus ensuring that it uses the highest common protocol version.

Since the Transit Link Negotiation Protocol is HTTP based, this way of encoding the protocol version in the URI has the added benefit of automatic errors messages being generated whenever a client attempts to use a protocol version not supported by the server: any request for an unsupported protocol version will result in an HTTP error code of 404 (File not Found[21, p. 66]), since the server won't have any routes configured for the protocol version in question.

4.2.7 Extensibility

Protocol versions are implemented in a plugin-based manner. There is a class named `Registry`, that classes implementing protocol versions must inherit. This class is notified

of existing protocol versions through Ruby's `inherited()` mechanism: `inherited` is a static method invoked with the inheriting class' type as argument for each class inheriting the top-level class.
Aside from inheriting from `Registry`, classes implementing new protocol versions must keep to the following contract:

1. They must have a static attribute `@@version` that contains their version number. This version number must be of the Gem::Version type, since this type defines comparison methods for version numbers.

2. Their constructor must create Sinatra routes for all of the protocol version's steps. It must take a `prefix` prefix parameter, a `server` parameter and a `logger` parameter. `prefix` is a path to prefix all created routes with, `server` is the instance of `Sinatra::Base` to register the routes with and logger is an instance of Ruby's `Logger`[8] class to be used for access and error logging.

Apart from complying with this contract, protocol version implementers are free to design their protocol version as they choose. In the case of Transit Link Negotiation Protocol version 1.0 I chose the lean architecture visible in figure B.2:
The `Main` class keeps the `Tlneg::Protocol::V10` the contract outlined above. Additionally, it contains methods for authentication and basic sanity checks of parameters supplied by clients. All protocol logic is handled by dedicated processing classes[12], one per protocol step (bottom portion of the figure).

4.2.8 Command-Line Programs

There are two command-line programs using the `tlneg` gem:

- `tlnegadm` is an administration utility for the Link Authority's database. Its chief purpose is creation and deletion of PIP objects and their associated SwitchPort objects[13].

- `tlnegd` is the Link Authority executable that launches the Sinatra based web server exposing the protocols functionality to clients.

4.3 Transit Link Negotiation Protocol Client

Along with the Link Authority I developed a client library and command-line client for the Transit Link Negotiation Protocol, which I will briefly describe in this section.

4.3.1 Client Library

Just like the Link Authority, the client library communicating with it is a Ruby gem (`tlneg-client`). It shares the plugin based architecture of the `tlneg` gem (see subsection 4.2.7): there is a protocol version registry (class `Registry`). All protocol version classes must inherit this class and have a public class attribute `@@version` denoting the protocol version they implement.

[12]These classes' `process()` methods could have been integrated in `Tlneg::Protocol::V10::Main`, but I chose to place them in separate classes so as to keep the Main class as lean and as generic as possible. This way it can easily be reused for future protocol versions.

[13]While both these data structures are used by the protocol I consider their creation a manual provisioning operation outside the scope of the Transit Link Negotiation Protocol, much like physically connecting a new PIP to the link domain. Consequently they are created out of band using `tlnegadm`.

This is the whole contract I am going to prescribe for implementations of client libraries at this point: depending on how protocol semantics change in future protocol versions, the client library's northbound APIs need to change as well, with consequences for the software using these APIs. Hence, the lowest common denominator I am going to set down as a hard requirement is a well-defined mechanism that exposes the available protocol versions to this software. This is realized as follows.
The user-facing `Client` class' constructor queries the registered protocol versions from the `Registry` class, performs the version handshake with the remote it was supplied and finds the maximum common supported version. It then sets an instance of this protocol version class as its public `@handler` attribute. It is then up to the user to query this instance's protocol version and treat it accordingly.
In the interest of extensibility, I tried to keep the protocol version 1.0 implementation fairly lean. All of its methods expect a Ruby `Hash`[14] populated with the implemented protocol message's payload as their sole argument (i.e. it is keyed by the field names specified in Appendix A, and the fields thus keyed contain the contents described there). They perform basic sanity checks — mainly ensuring required fields are present and all others are absent — serialize the `Hash` to JSON and send it to the appropriate URI.

4.3.2 Command Line Client

For testing and debugging a Link Authority I implemented the command-line client `tlneg`. It uses the `tlneg-client` Gem and can perform all protocol steps of Transit Link Negotiation Protocol 1.0.

4.4 Release Engineering Considerations

The Transit Link Negotiation Protocol reference implementation is available packaged form as part of the `cloudnets-framework` release 1.1. This section describes some of the challenges and considerations behind packaging and integration.

4.4.1 Supporting different Ruby Versions

There is a range of major incompatibilities between Ruby versions 1.8.7 and 1.9.3. Gregory Brown details a few of them in his article *Writing Backward-Compatible Code: Appendix A - Ruby Best Practices*[5]. `cloudnets-framework` is affected as well: It requires Ruby 1.8.7, since it contains large amounts of legacy code that is incompatible with Ruby 1.9.3. This is problematic, since Ruby 1.8.7 has gone out of support. So as not to make the Transit Link Negotiation Protocol entirely dependent on Ruby 1.8.7 I decided to split its reference implementation into two Ruby gems.
This split allowed me to develop and test the `tlneg` gem implementing the Link Authority in a Ruby 1.9.3 environment, thus keeping it compatible with the Ruby version shipped by default with most systems these days. The client library implemented in the `tlneg-client` gem needs to be usable by `cloudnets-framework`. Thus I developed it to be compatible with a Ruby 1.8.7 environment.

4.4.2 Packaging and Distribution

For the time being I integrated `tlneg` and `tlneg-client` with the `cloudnets-framework` package. This makes it possible to use its build infrastructure for configuration file generation and installation. Since the protocol may be of interest outside the `cloudnets-framework`,

[14] A Ruby class implementing an associative array.

this may have to change eventually. As most of the code base is in the self contained `tlneg` and `tlneg-client` gems, creating packages independent of `cloudnets-framework` will be fairly straightforward, should the need ever arise.

4.5 Integration in `cloudnets-framework`

In this section I would like to briefly discuss how the Transit Link Negotiation Protocol could be integrated into `cloudnets-framework`. I stopped short of full integration into `cloudnets-framework` for two reasons.

First, `cloudnets-framework` is a proof-of-concept implementation of a network operating system. It served its purpose of proving the concepts it set out to prove, but it can not compete with the engineering efforts going into production grade network operating systems such as Open Stack[44] and is slowly falling into disuse. Hence, I think integration of the Transit Link Negotiation Protocol into `cloudnets-framework` would serve little puprose.

Second, I believe that an integration effort would have given the Transit Link Negotiation Protocol reference implementation with far less attention than it received, leaving it with rougher edges and little to no release engineering. This would — if to a lesser degree — represent the same kind of obstacle to adoption Chondros et al. criticized in their paper on the practicality of Practical Byzantine Fault Tolerance[7]. Since I am quite sure that transit link negotiation will be a problem increasingly arising with the adoption of cloud computing, and consequently logical networks that span multiple cloud providers, I wanted to make adoption as easy as possible. Thus I believe I was right to focus my efforts on ensuring the Transit Link Negotiation Protocol reference implementation is both usable and well-documented.

Since I may turn to be wrong in my assessment of `cloudnets-framework` adoption, I will now sketch out the steps likely involved in integrating Transit Link Negotiation Protocol with `cloudnets-framework`, easing the work for those who take on this task. I will only outline implementation of VLAN based link virtualization, since MPLS based link virtualization would require extensive modifications to `cloudnets-framework` that go well beyond using the Transit Link Negotiation Protocol to retrieve configuration parameters from a Link Authority.

A note to implementers: All paths given in this section are relative to the root of the `cloudnets-framework` source tree, unless mentioned otherwise.

4.5.1 Protocol Integration in the VNP Role

The VNP role needs to perform two tasks: provision and deprovision transit links, and communicate the credentials required to join these links to the PIPs involved. Since link establishment must precede communication with PIPs, the state created during link creation must be kept between these steps. In the following two sections I will sketch out how this could be integrated with minimum effort by leveraging existing `cloudnets-framework` mechanisms, and the FleRD resource description language[15].

[15] See section 1.3.1.

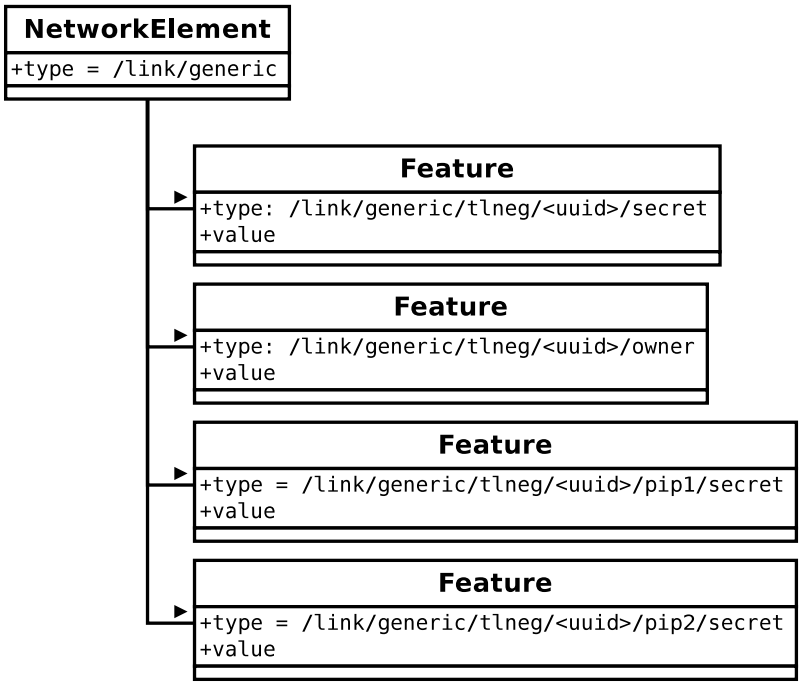

Figure 4.5: Transit Link metadata as a VNP might store them using the mechanisms provided by FleRD. Note that the `type` attribute is named `avp_attribute` in FleRD. I took the liberty of renaming it in this illustration for the sake of readability.

Embedding Plugin For Transit Links

First of all, transit links need to be created. To this end, an embedding plugin[16] for the network element type `/link/transit`[17] can be used. This plugin would get invoked for each virtual network element mapped to a network element of type `/link/transit`. It would then use the `tlneg-client` library to contact the link authority controlling the interconnection in question and request a new transit link connecting the PIPs participating in the transit link (the link authority's base URI can be stored as a FleRD Feature attached to the `/link/transit` network element in question. With the link created it would store its metadata as FleRD Features of the virtual network element mapped to the substrate's `/link/transit` network element.
To maintain backwards compatibility with the current static approach to link virtualization this plugin should only be invoked if it is configured in `/etc/cloudnets/define.rb`. It should run before a virtual network is split into partial graphs. Hence it is best to invoke it from the `create_vnet` function in `/cloudnets/lib/cloudnets/vnp/vnp_embed.rb`. When a virtual network is split into partial graphs, the appropriate credentials for the PIP this graph will be sent to (and only these credentials) need to be transferred to the partial graph in question. This is best accomplished in the `assemble_pgraphs()` function in `/cloudnets/lib/cloudnets/vnp/vnp_functions.rb`.

Keeping State in FleRD Graphs

Since a dedicated data structure for persisting and transmitting link metadata would entail significant effort (modifications to the FleRD database schema and ORM classes

[16] See section 1.3.3.

[17] In the VNP's substrate graph network elements of type `/link/transit` represent the interconnections between PIPs, and thus link domains.

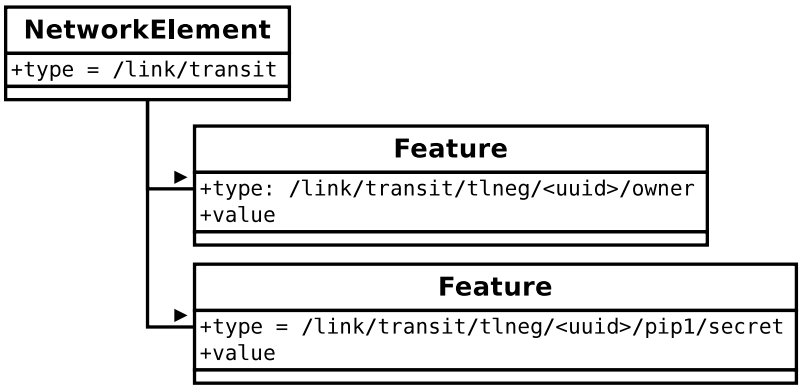

Figure 4.6: PIP specific Transit Link metadata as a VNP might pass them to a PIP involved in the link. Note that the `type` attribute is named `avp_attribute` in FleRD. I took the liberty of renaming it in this illustration for the sake of readability.

along with extensions to graph serialization and deserialization at the very least), it is probably best to leverage the FleRD resource description language's extensibility. This means storing link metadata using FleRD feature objects attached to network elements in two kinds of graphs:

- After link creation all link metadata are stored in Feature objects attached to `/link/transit` network elements of the VNP's Mapping Layer (ML0) graph (figure 4.5).
- When partial graphs (ML1 graphs) are created the metadata specific to the PIP a graph is meant for are attached to all `/link/transit` network elements occuring in this graph (figure 4.5.2). Currently this is how the VNP records VLAN tags to be used for transit links in partial graphs.

4.5.2 Protocol Integration in the PIP Role

Integration on the PIP level is fairly straightforward: in the course of the mapping process, the helper function `pick_vlan()` (in `cloudnets/lib/cloudnets/ml_functions.rb`) is invoked to assign a VLAN tag to each link. This function can be modified to use the `tlneg-client` library to issue a Join Link protocol message using the credentials attached to the transit link in question. It then returns the VLAN tag it receives from the link authority.

Chapter 5

Conclusion

The main contribution of this thesis is a protocol that allows for automatic negotiation of configuration parameters for transit links between providers. In the introduction I described the environment in which such transit links might exist: a virtualized network extending across cloud provider boundaries. Now I want to explore why it is in fact desirable for a service provider to have such a network at its disposal. The key reason lies in the very nature of cloud computing.

The cloud computing community uses the "Cattle versus Kittens" metaphor[18] as an analogy for comparing cloud computing with traditional ways of operating a network service. The servers and services in the latter case are likened to kittens: high maintenance pets with individual names and personalities that need attention, pampering and manual intervention at all points in their life cycle. On the other hand, Cloud based servers and the services running on them are likened to cattle: instead of names and personalities they have numbers and a well-defined task they will perform in a dull and reliable manner; both operation and provisioning are automated as much as possible. Cloud based services are elastic in the sense that the herd of servers can be extended — or culled — as needed, with little to no manual intervention on the operators' part.

In this elasticity lies the *raison d'être* for the Transit Link Negotiation Protocol: just like keeping cattle requires fences and herders, cloud services require a network to interconnect the virtual machines they run on. And much like the support infrastructure for cattle, this network infrastructure needs to grow and shrink in lockstep with the cloud service. This already works well as long as there is only one cloud provider involved. Modern control orchestration systems such as Open Stack offer both computing and network elasticity. Where elasticity breaks down is at the boundaries between cloud providers: a cloud operator intending to distribute its service across multiple cloud providers [1] today has the choice of interconnecting its partial clouds through either the public Internet or leased lines terminating at both cloud providers' network edges. Both of these options are unattractive. The Internet's best-effort packet delivery is unsuitable for the purpose of providing the sort of high-bandwidth, low-latency interconnection required for service failover between cloud providers. Leased lines on the other hand, are on the Kittens side of the spectrum: their setup typically requires phone calls to infrastructure providers, signed agreements faxed — or even mailed — back and forth, and manual provisioning steps by all involved parties.

The Transit Link Negotiation Protocol offers a way to make establishing such leased lines (or MPLS tunnels) just as elastic as growing or shrinking the cloud partitions they connect. It is not complete (more on that in the Outlook), and it may well replaced by a different protocol, but it is my firm conviction, that it or an analogous mechanism will in the future

[1] A prudent move, since large scale outages of cloud providers are not unheard of[54][55].

be used as the glue that binds cloud services together. There already have been efforts at creating such a mechanism in the past[3], there are currently ongoing efforts[13] and I am quite certain there will be more in the future. I take the fact that this thesis is not the first step in this direction (even though it is the only one with publicly available source code to the best of my knowledge) as a strong indicator of evolutionary pressure towards interoperable clouds. Hence I am quite sure they will soon be as common as isolated cloud providers without interconnections are today.

Outlook

In this thesis I implemented one of the components required for enabling connectivity between virtual networks hosted by different cloud providers. For this connectivity to become as commonplace as interior network virtualization others need to follow. In this section I shall trace out a possible route towards realizing this mechanism using the OpenStack[44] cloud operating system and put the Transit Link Negotiation Protocol into its context.
At the top level, cloud providers will have to provide a protocol that allows outside entities to request interconnections. I assume this will look similar to what OpenStack project's Neutron service currently performs for clouds' interior networking. The Neutron service currently allows for creating virtual Ethernet switches and virtual ports connecting these switches to virtual machines, referred to as Networks and Ports in OpenStack parlance[12]. The envisioned protocol would then allow for creating what I propose to call *Remote Ports*. These ports attach to Neutron networks on one side and to the transit link on the other side. Upon creation the Neutron network to connect to and the remote network to establish a transit link with are specified. On the transit link side the implementation would look up the appropriate link domain for the desired remote networks and use the Transit Link Negotiation Protocol to obtain configuration parameters for this link.
While this sounds like a fairly straightforward case of implementation at first, there is a range of points at which significant engineering is likely to be required:

1. Specifying the remote virtual network will require a globally unique identifier for a virtualized network. This identifier needs to be standardized and recognized by all cloud providers wishing to offer interconnections.

2. Cloud providers need to know which of the link domains they participate in, — or possibly a chain of other link domains if desired remote networks are not hosted with adjacent providers — to use for a given remote virtual network. This likely requires both a routing protocol for finding the destination cloud provider and possibly a signalling protocol for establishing chains of transit links.

3. There needs to be a mechanism that ensures only authorized transit links are established between virtualized networks. This likely requires a handshake protocol between cloud providers.

Appendix A

Protocol Specification

This is the protocol specification for the Transit Link Negotiation Protocol, version 1.0. It is meant as an implementer's reference. Thus it is deliberately terse and assumes familiarity with the protocol's purpose and semantics.

A.1 Protocol Parties

There are two parties to the protocol:

- The *Link Authority*. It operates a server responding to protocol messages. Henceforth referred to as Link Authority.
- An arbitrary number of clients. A *Client* sends protocol messages to a Link Authority.

These parties will be referenced by the name in italics in the following.

A.2 Transport Protocol

The transport protocol for all Transit Link Negotiation Protocol protocol messages is HTTP[21]. The protocol version handshake is a HTTP GET requests. All other protocol messages are HTTP POST requests with the protocol message payload sent by the Client in their bodies.

A.3 List of Protocol Steps

Transit Link Negotiation Protocol 1.0 defines the following protocol steps:

Name	**Description**
Version Handshake	Returns the Link Authority's list of supported protocol versions.
Create Link	Creates a new transit link.
Delete Link	Removes an existing transit link.
Add PIPs	Adds one or more PIP members to a transit link.
Remove PIPs	Removes one or more PIP members from a transit link.
Join Link	Supplies link configuration parameters to a transit link's member PIP.

In the following these protocol steps will be referred to by the names in the **Name** column of this table.

A.4 Protocol Paths

The Link Authority receives protocol messages for protocol steps at the following paths relative to its web server root directory:

Protocol Step	**Path**	HTTP Method
Version Handshake	`/tlneg/versions`	GET
Create Link	`/tlneg/1.0/link/create`	POST
Delete Link	`/tlneg/1.0/link/<ID>/delete`	POST
Add PIPs	`/tlneg/1.0/link/<ID>/addpips`	POST
Remove PIPs	`/tlneg/1.0/link/<ID>/removepips`	POST
Join Link	`/tlneg/1.0/link/<ID>/join`	POST

The placeholder `<ID>` in these paths represents the identifier of the specific transit link a protocol message acts upon.

A.5 Format of Protocol Messages

All protocol messages are JSON objects. Their contents are specified in the next section.

A.6 Protocol Messages

This section describes the following aspects of all protocol messages:

- Request contents.
- Response contents.
- Failure modes.

A.6.1 Failures

If a protocol step fails an error data structure is returned. The error data structure is a JSON object with the following attributes:

Attribute	**Type**	**Description**
`success`	Boolean	A value of `false`, indicating failure.
`error`	String	An error message describing the type of failure.

A.6.2 Version Handshake

This step must be performed before any further protocol exchanges. It requires no parameters and returns a response.
In response to this protocol step the Link Authority returns a list of protocol versions it supports. The Client must pick the highest version supported by itself from this list and use it in all future protocol exchanges.

Response Data Structure

A JSON array of Strings containing supported protocol versions. Example:

```
["1.0"]
```

A.6.3 Create Link

This step creates a new transit link. It requires parameters and returns a response. It is initiated by a VNP or other entity desiring to establish a new transit link.

Request Data Structure

The request data structure sent by the VNP or other link creator is a JSON object with the following attributes:

Attribute	Type	Description
`owner`	String	The Link's owner's name. Free-from but must be globally unique, i.e. a DNS domain name or E-Mail address.
`pips`	Array	Contains the names of PIPs (strings) that form the link's initial set of members. Must have two or more entries. All PIPs must be in the Link Authority's database of PIPs.

Response Data Structure

The response data structure is a JSON object with the following attributes if the operation is successful:

Attribute	Type	Description
`pip_members`	Object	A list of key-value pairs. Keys are the PIP names supplied in the request, values are the secrets individual PIPs must use to authenticate their own *Join Link* requests.
`identifer`	String	A UUID uniquely identifying the transit link just created
`success`	Boolean	The value `true`, indicating successful link creation.

Upon failure an eror data structure (see section A.6.1) is returned. Its `error` attribute will be one of the following:

Error	Cause
Parameter pips must be an array.	Bad `pips` parameter supplied in request.
Parameter pips must contain at least two entries.	Not enough PIPs supplied in request.
The following PIPs do not exist: <PIPs>	Nonexistent PIPs supplied as link members. <PIPs> will be a list of these nonexistent PIPs.
No more VLANs available.	VLAN tag range for this Link Domain exhausted.
No more VE Identifiers available.	VE Identifier range for this Link Domain exhausted.
No more Route Targets available.	Route Target Identifier range for this Link Domain exhausted.

A.6.4 Delete Link

This step deletes an existing transit link. It requires parameters and returns a response. It is initiated by a VNP or other entity owning a link.

Request Data Structure

The request sent by the VNP or link owner is a JSON object with the following attributes:

Attribute	Type	Description
`identifier`	String	The link's UUID.
`vnp_identifier`	String	The link's owner. Must be equal to the `owner` field used in the links *Create Link* step.
`secret`	String	The link's authentication secret, i.e. the `secret` attribute of the response data structure returned upon link creation.

Response Data Structure

The response data structure is a JSON object with the following attributes if the operation is successful:

Attribute	Type	Description
`success`	Boolean	The value `true`, indicating successful link deletion.

Upon failure an eror data structure (see section A.6.1) is returned. Its `error` attribute will be one of the following:

Error	Cause
Authentication failed.	Bad `vnp_identifier`, bad `secret` or nonexistent link.

A.6.5 Add PIPs

This step adds PIP members to an existing transit link. It requires parameters and returns a response. It is initiated by a VNP or other entity owning a link.

Request Data Structure

The request sent by the VNP or link owner is a JSON object with the following attributes:

Attribute	Type	Description
`identifier`	String	The link's UUID.
`vnp_identifier`	String	The link's owner. Must be equal to the `owner` field used in the links *Create Link* step.
`secret`	String	The link's authentication secret, i.e. the `secret` attribute of the response data structure returned upon link creation.
`pips`	Array	Contains the names of PIPs (strings) to add to the link. Must have at least one entry. All PIPs must be in the Link Authority's database of PIPs.

Response Data Structure

The response data structure is a JSON object with the following attributes if the operation is successful:

Attribute	Type	Description
`pip_members`	Object	A list of key-value pairs. Keys are the PIP names supplied in the request, values are the secrets individual PIPs must use to authenticate their own *Join Link* requests.
`success`	Boolean	The value `true`, indicating successful link creation.

Upon failure an eror data structure (see section A.6.1) is returned. Its `error` attribute will be one of the following:

Error	Cause
Parameter pips must be an array.	Bad `pips` parameter supplied in request.
Parameter pips must contain at least one entry.	Empty `pips` parameter supplied in request.
The following PIPs do not exist: <PIPs>	Nonexistent PIPs supplied as link members. <PIPs> will be a list of these nonexistent PIPs.
No more VE Identifiers available.	VE Identifier range for this Link Domain exhausted.
Authentication failed.	Bad `vnp_identifier`, bad `secret` or nonexistent link.

A.6.6 Remove PIPs

This step removes PIP members from an existing transit link. It requires parameters and returns a response. It is initiated by the VNP or other entity owning a link.

Request Data Structure

The request sent by the VNP or link owner is a JSON object with the following attributes:

Attribute	Type	Description
`identifier`	String	The link's UUID.
`vnp_identifier`	String	The link's owner. Must be equal to the `owner` field used in the links *Create Link* step.
`secret`	String	The link's authentication secret, i.e. the `secret` attribute of the response data structure returned upon link creation.
`pips`	Array	Contains the names of PIPs (strings) to remove from the link. Must have at least one entry. All PIPs must be in the Link Authority's database of PIPs and members of the link in question.

Response Data Structure

The response data structure is a JSON object with the following attributes if the operation is successful:

Attribute	Type	Description
`removed`	Array	A list of the PIPs that were removed from the link.
`success`	Boolean	The value `true`, indicating successful link creation.

Upon failure an eror data structure (see section A.6.1) is returned. Its `error` attribute will be one of the following:

Error	Cause
Parameter pips must be an array.	Bad `pips` parameter supplied in request.
Parameter pips must contain at least one entry.	Empty `pips` parameter supplied in request.
The following PIPs do not exist: <PIPs>	Nonexistent PIPs supplied as link members. <PIPs> will be a list of these nonexistent PIPs.
The following PIPs are not members of this link: <PIPs>	Non-member PIPs supplied as PIPs to be removed from the link. <PIPs> will be a list of these non-member PIPs.
Authentication failed.	Bad `vnp_identifier`, bad `secret` or nonexistent link.

A.6.7 Join Link

This step supplies a transit link's PIP members with the configuration parameters they need for link participation. It requires parameters and returns a response. It is initiated by each of the link's member PIPs.

Request Data Structure

The request sent by the PIP is a JSON object with the following attributes:

Attribute	Type	Description
`identifier`	String	The link's UUID.
`pip_identifier`	String	The PIP member's name. Must be equal to one of the PIP names used as key in the `pip_members` key-value list returned by this link's *Create Link operation*.
`secret`	String	The PIP member's authentication secret. Must be the value of this PIP's entry in the `pip_members` key-value list returned by this link's *Create Link operation*.

Response Data Structure

The response data structure is a JSON object with the following attributes if the operation is successful:

Attribute	Type	Description
`link_type`	String	The link's virtualization type. Can either be `vpls` or `vlan`.
`rt_identifier`	Integer	The Link's Route Target Identifier in the case of `vpls` type links.
`ve_identifier`	Integer	This PIP's Virtual Edge Identifier in the case of `vpls` type links.
`vlan_tag`	Integer	The Link's VLAN tag in the case of `vlan` type links.
`success`	Boolean	The value `true`, indicating successful link creation.

Upon failure an eror data structure (see section A.6.1) is returned. Its `error` attribute will be one of the following:

Error	Cause
Authentication failed.	Bad `pip_identifier`, bad `secret` or non-member PIP.

Appendix B

Illustrations and Tables

Here you will find illustrations and tables I deemed to large for the main text.

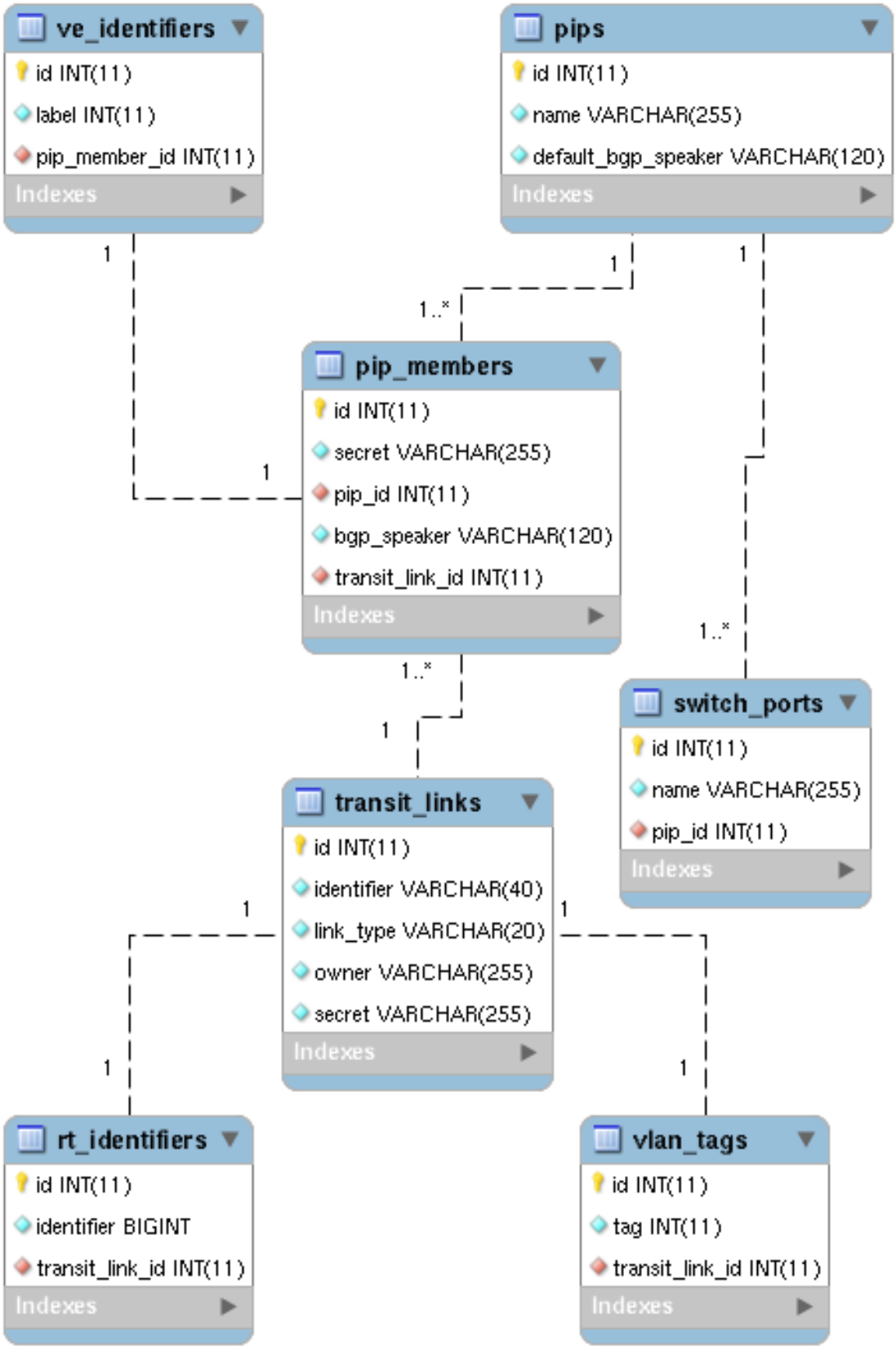

Figure B.1: ER diagram of the database used to maintain the Link Authority's state.

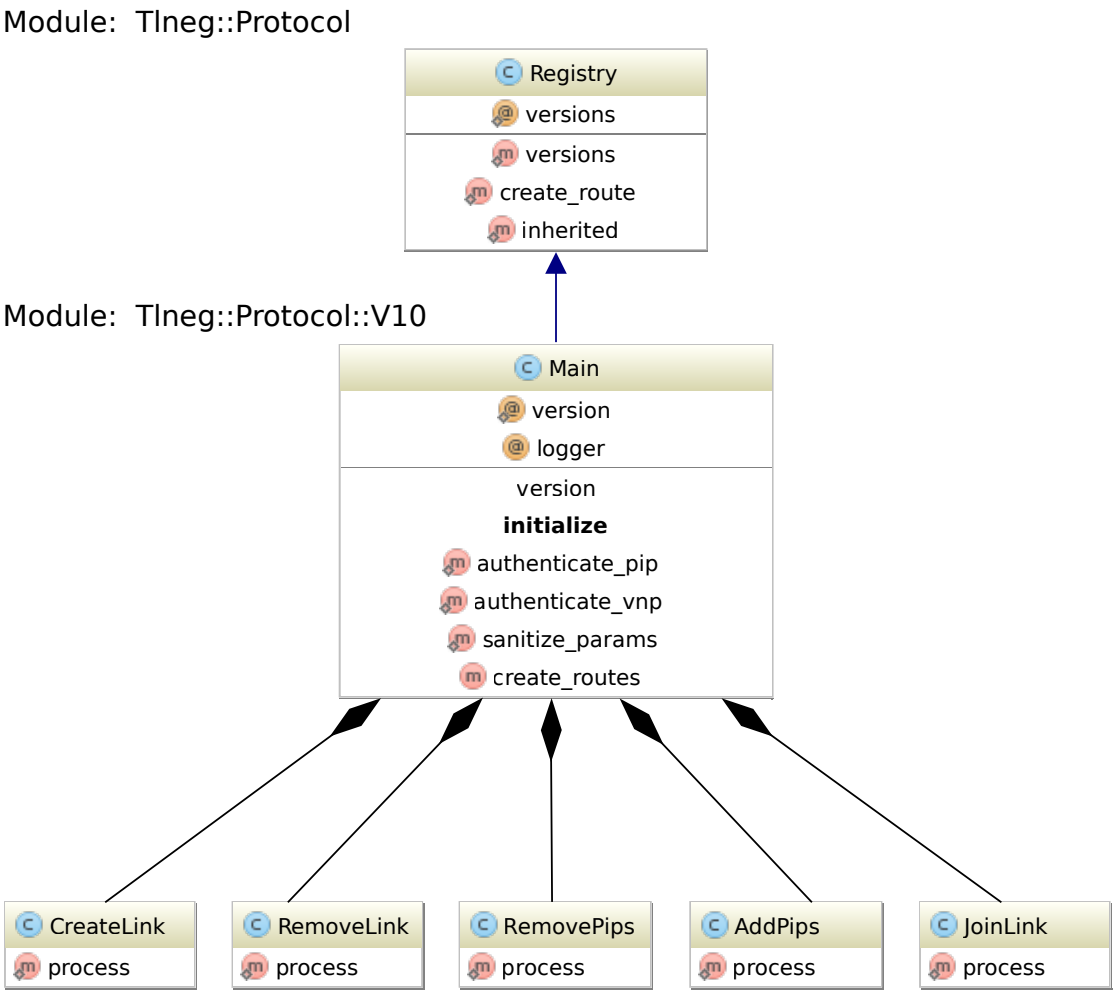

Figure B.2: Class diagram of the classes involved in the `tlneg` plugin infrastructure. The diagram shows the class hierarchy for protocol version 1.0.

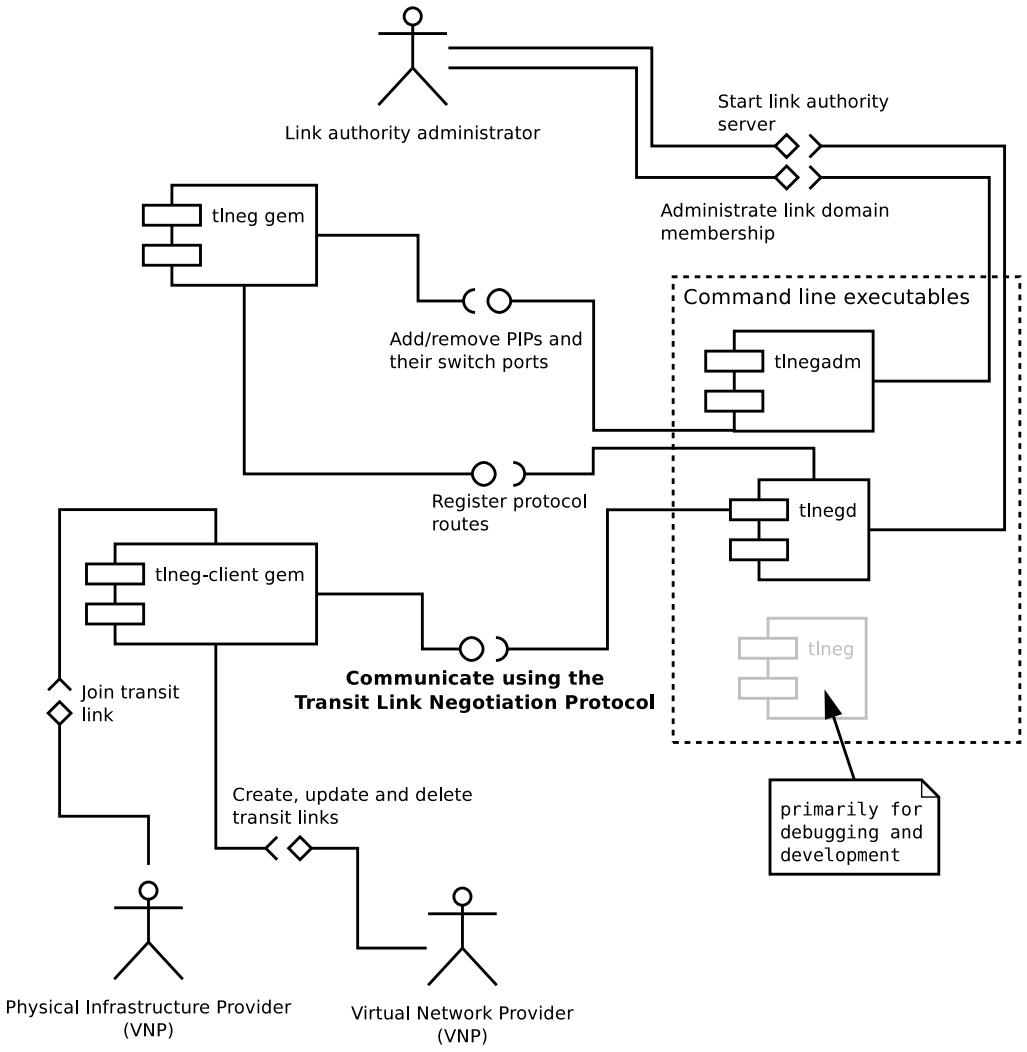

Figure B.3: Components of the Transit Link Negotiation Protocol reference implementation and their relationships. The relationships of the command line client `tlneg` have been omitted for clarity's sake since it is not normally part of the protocol's operation (it exists mainly for development and testing purposes).

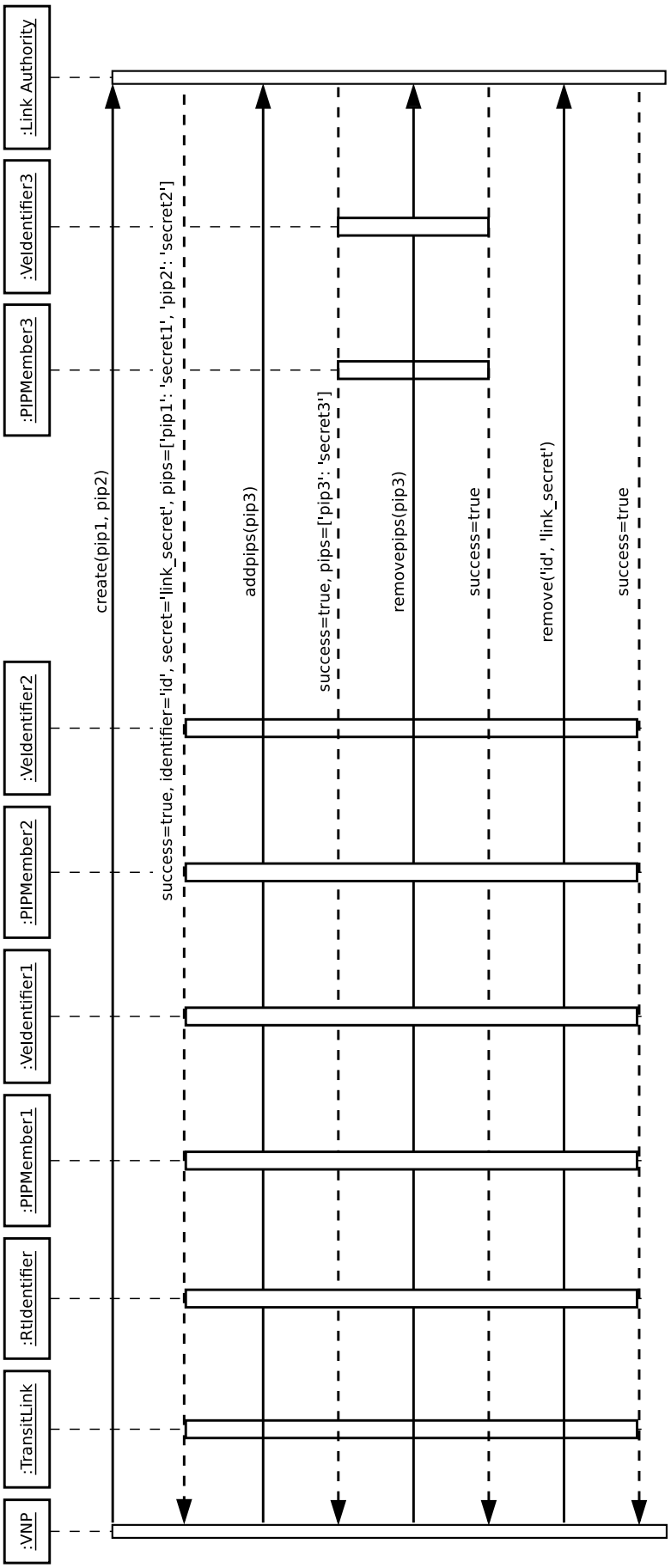

Figure B.4: Life cycle of a MPLS based transit link in the Link Authority's database. The PIPs' `join` protocol steps are not shown, since they do not affect database state.

Executables	
`bin/tlneg`	Command line client for the Link Paramater Negotiation Protocol
`bin/tlnegadm`	Command line interface to the link authority's database
`bin/tlnegd`	Link Authority Server executable
Client library (tlneg-client gem)	
`tlneg-client/lib/tlneg-client.rb`	Main file (pulls in the rest of the library's files).
`tlneg-client/lib/tlneg-client/client.rb`	Northbound interface of tlneg-client
`tlneg-client/lib/tlneg-client/exceptions.rb`	Exceptions raised by tlneg-client
`tlneg-client/lib/tlneg-client/protocol/registry.rb`	Registry for protocol version handlers
`tlneg-client/lib/tlneg-client/protocol/1.0/main.rb`	Handler class for protocol version 1.0
Link Authority library (tlneg gem)	
`tlneg/lib/tlneg.rb`	External dependencies, static configuration and constants
`tlneg/lib/tlneg/classes.rb`	Pulls in the classes making up the object-relational model.
`tlneg/lib/tlneg/config.rb`	Performs sanity checks for various configuration parameters
`tlneg/lib/tlneg/exceptions.rb`	Exceptions raised by the tlneg gem.
`tlneg/lib/tlneg/protocol/registry.rb`	Registry for implemented protocol versions
`tlneg/lib/tlneg/classes/Pip.rb`	Data model class representing PIPs (Physical Infrastructure Providers)
`tlneg/lib/tlneg/classes/RtIdentifier.rb`	Data model class representing Route Target Identifiers (VPLS)
`tlneg/lib/tlneg/classes/SwitchPort.rb`	Data model class representing a link domain's switch ports
`tlneg/lib/tlneg/classes/TransitLink.rb`	Central data model: represents transit links
`tlneg/lib/tlneg/classes/VlanTag.rb`	Data model class representing VLAN tags
`tlneg/lib/tlneg/protocol/1.0/main.rb`	Route registration and shared methods for protocol version 1.0
`tlneg/lib/tlneg/protocol/1.0/addpips.rb`	Implementation of the Add Pips protocol step
`tlneg/lib/tlneg/protocol/1.0/create.rb`	implementation of the Create Link protocol step
`tlneg/lib/tlneg/protocol/1.0/remove.rb`	Implementation of the Remove Link protocol step
`tlneg/lib/tlneg/protocol/1.0/removepips.rb`	Implementation of the Remove Pips protocol step
`tlneg/lib/tlneg/protocol/1.0/join.rb`	Implementation of the Join Link protocol step

Table B.1: File system locations of Transit Link Negotiation Protocol code relative to the `cloudnets-framework` source directory.

Appendix C

Source code

The source code of my Transit Link Negotiation Protocol reference implementation is available from `tlneg` branch of the `cloudnets-framework` Git repository at `https://projects.net.t-labs.tu-berlin.de/cloudnets-framework/cloudnets-src.git`.
Code written in the course of this work starts at commit `1e34807dc9fc6794b422b86a2b2e1400a0910b6d` and ends at commit `398bbe38d860e0dc93dddff3bf771851837162ed`.
A snapshot of the `tlneg` branch has been included on CD along with this thesis to serve as a copy of record. This CD also contains an electronic version of this thesis in PDF format.
Alternatively, the Transit Link Negotiation Protocol reference implementation is also available in packaged form at `https://www.net.t-labs.tu-berlin.de/~cloudnets-framework/releases/cloudnets-framework-1.1.0-pre.tar.bz2`.

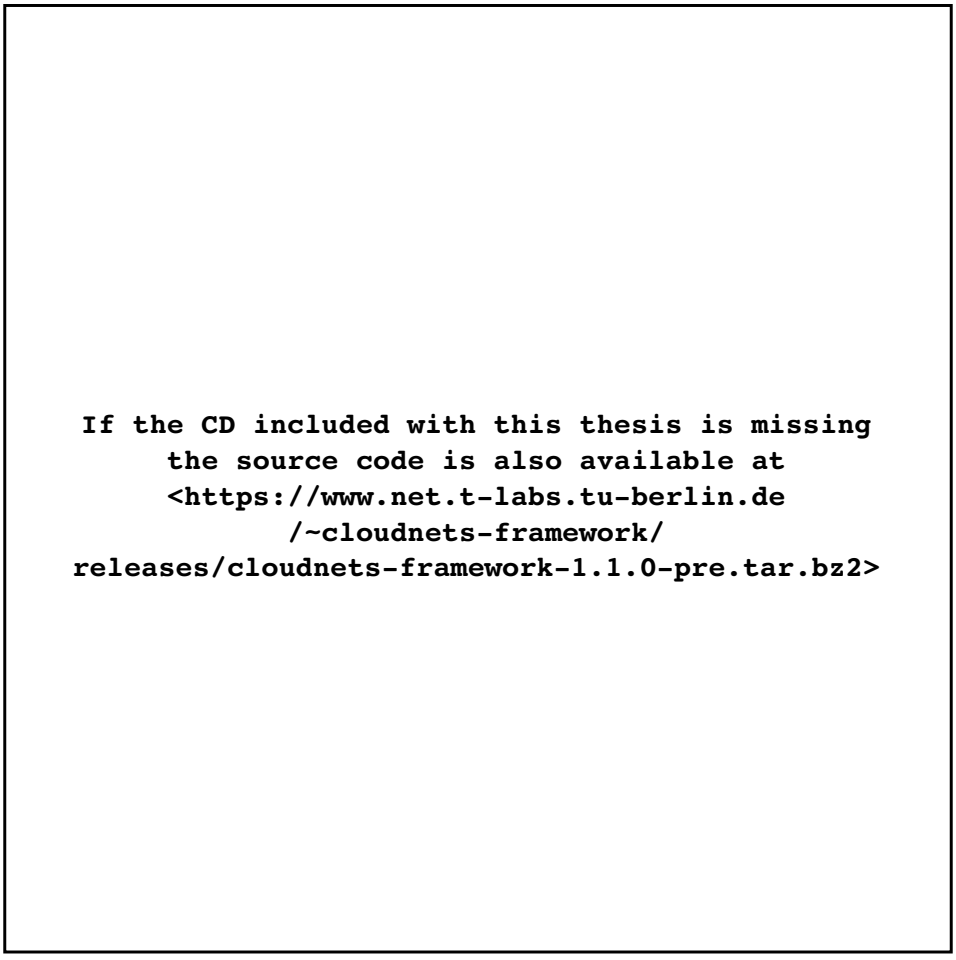

Figure C.1: CD with cource code and this thesis in electronic format.

Bibliography

[1] Ittai Abraham et al. "Byzantine disk paxos: optimal resilience with Byzantine shared memory". In: *Distributed Computing* 18.5 (2006), pp. 387–408.

[2] Bernhard Ager et al. "Anatomy of a large European IXP". In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM. 2012, pp. 163–174.

[3] David Allan, James Kempf, and Torbjörn Cagenius. "Enabling the network-embedded cloud". In: *Ericsson Review* (12/2012 2012). URL: `http://www.ericsson.com/res/thecompany/docs/publications/ericsson_review/2012/er-network-enabled-cloud.pdf` (visited on 08/26/2014).

[4] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Jan. 2005. URL: `http://www.rfc-editor.org/rfc/rfc3986.txt` (visited on 08/26/2014).

[5] Gregory Brown. *Writing Backward-Compatible Code: Appendix A - Ruby Best Practices*. URL: `http://www.oreillynet.com/pub/a/ruby/excerpts/ruby-best-practices/writing-backward-compatible.html` (visited on 08/26/2014).

[6] Miguel Castro and Barbara Liskov. "Practical Byzantine fault tolerance and proactive recovery". In: *ACM Transactions on Computer Systems (TOCS)* 20.4 (2002), pp. 398–461.

[7] Nikos Chondros, Konstantinos Kokordelis, and Mema Roussopoulos. "On the practicality of practical Byzantine fault tolerance". In: *Middleware 2012*. Springer, 2012, pp. 436–455.

[8] *Class: Logger (Ruby 1.9.3)*. The Ruby Community. URL: `http://www.ruby-doc.org/stdlib-1.9.3/libdoc/logger/rdoc/Logger.html` (visited on 08/26/2014).

[9] European Commission. *What is FP7? The basics*. URL: `http://ec.europa.eu/research/fp7/understanding/fp7inbrief/what-is_en.html` (visited on 08/26/2014).

[10] Ruby Community. *Ruby Gems*. URL: `http://rubygems.org/` (visited on 08/26/2014).

[11] The Ruby Community. *Ruby Programming Language*. URL: `https://www.ruby-lang.org/en/` (visited on 08/26/2014).

[12] OpenStack Consortium. *Neutron API v2 Specification: Concepts*. URL: `https://wiki.openstack.org/wiki/Neutron/APIv2-specification#Concepts` (visited on 08/26/2014).

[13] UNIFY Consortium. *UNIFY web page*. URL: `http://www.fp7-unify.eu/` (visited on 08/26/2014).

[14] Oracle Corporation. *MySQL web page*. URL: `http://libvirt.org/` (visited on 08/26/2014).

[15] Douglas Crockford. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7158. Mar. 2013. URL: http://www.rfc-editor.org/rfc/rfc7158.txt (visited on 08/26/2014).

[16] R. Doriguzzi Corin et al. "VeRTIGO: Network Virtualization and Beyond". In: (Oct. 2012), pp. 24–29. DOI: 10.1109/EWSDN.2012.19.

[17] Roberto Doriguzzi. *VeRTIGO README file*. URL: https://raw.githubusercontent.com/fp7-ofelia/VeRTIGO/master/README.vertigo (visited on 08/26/2014).

[18] Greg Ferro. *Cattle vs Kittens – On Cloud Platforms No One Hears the Kittens Dying*. URL: http://etherealmind.com/cattle-vs-kittens-on-cloud-platforms-no-one-hears-the-kittens-dying/ (visited on 08/26/2014).

[19] Roy T. Fielding and Richard N. Taylor. "Principled Design of the Modern Web Architecture". In: *ACM Trans. Internet Technol.* 2.2 (May 2002), pp. 115–150. ISSN: 1533-5399. DOI: 10.1145/514183.514185. URL: http://doi.acm.org/10.1145/514183.514185 (visited on 08/26/2014).

[20] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.

[21] Roy T. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. URL: http://www.rfc-editor.org/rfc/rfc2616.txt (visited on 08/26/2014).

[22] Vince Fuller and Tony Li. *The Internet Address Assignment and Aggregation Plan*. RFC 4632. Aug. 2006. URL: http://www.rfc-editor.org/rfc/rfc4632.txt (visited on 08/26/2014).

[23] "GENI Design Principles". In: *Computer* 39.9 (2006), pp. 102–105. ISSN: 0018-9162. DOI: 10.1109/MC.2006.307.

[24] Johannes Graßler. *CloudNets negotiation interface documentation 1.0.0*. Apr. 2014. URL: http://www.net.t-labs.tu-berlin.de/~cloudnets-framework/doc/1.0.0/negotiate-frontend/top-level-namespace.html (visited on 08/26/2014).

[25] Johannes Graßler. *FleRD API documentation 1.0.0*. Apr. 2014. URL: http://www.net.t-labs.tu-berlin.de/~cloudnets-framework/doc/1.0.0/flerd/index.html (visited on 08/26/2014).

[26] Johannes Grassler, Gregor Schaffrath, and Stefan Schmid. "A Federated CloudNet Architecture: The PIP and the VNP Role". In: *it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik* 55.4 (2013), pp. 155–162.

[27] David Heinemeier Hansson. *ActiveRecord web page*. URL: http://rubygems.org/gems/activerecord (visited on 08/26/2014).

[28] Urs Hölzle. *OpenFlow Google*. URL: http://www.opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf (visited on 08/26/2014).

[29] Sushant Jain et al. "B4: Experience with a globally-deployed software defined WAN". In: *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM. 2013, pp. 3–14.

[30] *Virtual Private LAN Service (VPLS) Using BGP for Auto-Discovery and Signaling*. RFC 4761. Jan. 2007. URL: http://www.rfc-editor.org/rfc/rfc4761.txt (visited on 08/26/2014).

[31] Leslie Lamport. "Byzantizing paxos by refinement". In: *Distributed Computing*. Springer, 2011, pp. 211–224.

[32] Leslie Lamport. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[33] Leslie Lamport. "The part-time parliament". In: *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998), pp. 133–169.

[34] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[35] Paul J. Leach, Michael. Mealling, and Rich. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. July 2005. URL: `http://www.rfc-editor.org/rfc/rfc4122.txt` (visited on 08/26/2014).

[36] *LOCAL AND METROPOLITAN AREA NETWORK STANDARDS*. Tech. rep. Institute of Electrical and Electronics Engineers (IEEE). URL: `http://standards.ieee.org/getieee802/download/802.3-2012.zip` (visited on 08/26/2014).

[37] Nick McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: `10.1145/1355734.1355746`. URL: `http://doi.acm.org/10.1145/1355734.1355746` (visited on 08/26/2014).

[38] *Media Access Control (MAC) Bridges and Virtual Bridge Local Area Networks*. Tech. rep. Institute of Electrical and Electronics Engineers (IEEE). URL: `http://standards.ieee.org/getieee802/download/802.1Q-2011.pdf` (visited on 08/26/2014).

[39] Blake Mizerany. *Sinatra*. URL: `http://www.sinatrarb.com/` (visited on 08/26/2014).

[40] Paul Mockapetris. *DOMAIN NAMES - CONCEPTS AND FACILITIES*. RFC 1034. Nov. 1987. URL: `http://tools.ietf.org/rfc/rfc1034.txt` (visited on 08/26/2014).

[41] *Neutron*. OpenStack Foundation. URL: `https://wiki.openstack.org/wiki/Neutron` (visited on 08/26/2014).

[42] Elizabeth J. O'Neil. "Object/relational mapping 2008: hibernate and the entity data model (edm)". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 1351–1356.

[43] *OpenFlow Switch Specification*. Tech. rep. 1.3.4. Open Networking Foundation. URL: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.4.pdf` (visited on 08/26/2014).

[44] *OpenStack Open Source Cloud Computing Software*. OpenStack Foundation. URL: `https://www.openstack.org` (visited on 08/26/2014).

[45] Jon Postel. *TRANSMISSION CONTROL PROTOCOL*. RFC 793. Sept. 1981. URL: `http://www.rfc-editor.org/rfc/rfc793.txt` (visited on 08/26/2014).

[46] *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. Jan. 2006. URL: `http://www.rfc-editor.org/rfc/rfc4271.txt` (visited on 08/26/2014).

[47] *Root Zone Management*. Internet Assigned Numbers Authority. URL: `http://www.iana.org/domains/root` (visited on 08/26/2014).

[48] Eric C. Rosen, Arun. Viswanathan, and Ross. Callon. *Multiprotocol Label Switching Architecture*. RFC 3031. Jan. 2001. URL: `http://www.rfc-editor.org/rfc/rfc3031.txt` (visited on 08/26/2014).

[49] *Ruby on Rails API*. URL: `http://api.rubyonrails.org/` (visited on 08/26/2014).

[50] Srihari R. Sangli, Dan Tappan, and Yakov Rekhter. *BGP Extended Communities Attribute*. RFC 4360. Feb. 2006. URL: `http://www.rfc-editor.org/rfc/rfc4360.txt` (visited on 08/26/2014).

[51] Gregor Schaffrath. "Virtual Network Management". PhD thesis. Technische Universität Berlin, 2012.

[52] Gregor Schaffrath et al. "A Resource Description Language with Vagueness Support for Multi-Provider Cloud Networks". In: *Proceedings of International Conference on Computer Communication Networks (ICCCN)*. 2012.

[53] Gregor Schaffrath et al. "Network Virtualization Architecture: Proposal and Initial Prototype". In: *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*. VISA '09. Barcelona, Spain: ACM, 2009, pp. 63–72. ISBN: 978-1-60558-595-6. DOI: 10.1145/1592648.1592659. URL: http://doi.acm.org/10.1145/1592648.1592659 (visited on 08/26/2014).

[54] Amazon Web Services. *Summary of the AWS Service Event in the US East Region*. July 2012. URL: http://aws.amazon.com/message/67457/ (visited on 08/26/2014).

[55] Amazon Web Services. *Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region*. Dec. 2012. URL: http://aws.amazon.com/message/680587/ (visited on 08/26/2014).

[56] Ali Al-Shabibi. *fvctl manual page*. URL: https://raw.githubusercontent.com/OPENNETWORKINGLAB/flowvisor/1.4-MAINT/doc/fvctl.1 (visited on 08/26/2014).

[57] Rob Sherwood et al. *Flowvisor: A network virtualization layer*. Technical Report. 2009. URL: http://archive.openflow.org/downloads/technicalreports/openflow-tr-2009-1-flowvisor.pdf (visited on 08/26/2014).

[58] David W. Sincoskie and Charles J. Cotton. "Extended bridge algorithms for large networks". In: *Network, IEEE* 2.1 (Jan. 1988), pp. 16–24. ISSN: 0890-8044. DOI: 10.1109/65.3233.

[59] YAML Web Site. *Clark C. Evans*. URL: http://yaml.org/ (visited on 08/26/2014).

[60] *Special-Purpose Multiprotocol Label Switching (MPLS) Label Values*. URL: http://www.iana.org/assignments/mpls-label-values/mpls-label-values.txt (visited on 08/26/2014).

[61] Marc Suñé et al. "Design and implementation of the OFELIA FP7 facility: The European OpenFlow testbed". In: *Computer Networks* (2014), pp. 132–150. DOI: http://dx.doi.org/10.1016/j.bjp.2013.10.015. URL: http://www.sciencedirect.com/science/article/pii/S1389128613004301 (visited on 08/26/2014).

[62] cloudnets-framework development team. *cloudnets-framework project page*. URL: https://projects.net.t-labs.tu-berlin.de/projects/cloudnets-framework/wiki (visited on 08/26/2014).

[63] Hubert Zimmermann. "OSI reference model–The ISO model of architecture for open systems interconnection". In: *Communications, IEEE Transactions on* 28.4 (1980), pp. 425–432.